



PowerGREP

Manual

Version 4.6.1 — 9 June 2014

Published by Just Great Software Co. Ltd.

Copyright © 2002–2014 Jan Goyvaerts. All rights reserved.

“PowerGREP” and “Just Great Software” are trademarks of Jan Goyvaerts

Table of Contents

How to Use PowerGREP 1

1. Introducing PowerGREP.....	3
2. Contact PowerGREP’s Developer and Publisher.....	4
3. Getting Started with PowerGREP.....	5
4. Mark Files for Searching.....	7
5. Define a Search Action.....	9
6. Interpret Search Results.....	12
7. Edit Files and Replace or Revert Individual Matches.....	14
8. Keyboard Shortcuts.....	15
9. Regular Expression Quick Start.....	16

PowerGREP Examples 21

1. Search Through File Names.....	23
2. Find Files Not Containing a Search Term.....	25
3. Find Email Addresses.....	26
4. How to Find Word Pairs.....	28
5. Boolean Operators “and”, “or”, and “not”.....	29
6. Find Two Words Near Each Other.....	32
7. Find Two or More Words on The Same Line.....	33
8. Search Through Printable Content in Word .docx Files.....	34
9. Search Through Printable Content in XPS Files.....	36
10. Search Through Printable Content in OpenDocument Format Files.....	37
11. Extract or Delete Lines Matching One or More Strings or Regexes.....	39
12. How to Delete Repeated Words.....	40
13. Add a Header and Footer to Files.....	41
14. Add Line Numbers.....	42
15. Collect Page Numbers.....	44
16. Update Copyright Years.....	45
17. Padding Replacements.....	47
18. Capitalize The First Letter of Each Word.....	48
19. Add Proper HTML <TITLE> Tags.....	49
20. Rename Files Based on HTML Title Tags.....	51
21. Replace HTML Tags.....	52
22. Replace HTML Attributes.....	53
23. Put Anchors Around URLs That Are Not Already Inside a Tag or Anchor.....	54
24. Replace in File Names and Contents.....	55
25. Replacing Named XML Entities.....	56
26. Fix Invalid Characters in XML.....	58
27. Search Through or Skip Source Code Comments and Strings.....	60
28. Convert Windows to UNIX Paths.....	62
29. Extract Data into a CSV File or Spreadsheet.....	64
30. Padding and Unpadding CSV Files.....	66
31. Collect a Numbered List.....	67

32. Collect a List of Header and Item Pairs.....	68
33. Collect Paragraphs (Split along Blank Lines).....	70
34. Apply an Extra Search-And-Replace to Target Files.....	71
35. Inspect Web Logs.....	72
36. Extract Google Search Terms from Web Logs.....	74
37. Split Web Logs by Date.....	75
38. Merge Web Logs by Date	77
39. Split Logs into Files with a Certain Number of Entries	78
40. Compile Indices of Files.....	80
41. Make Sections and Their Contents Consistent.....	81
42. Generate a PHP Navigation Bar.....	83
43. Include a PHP Navigation Bar.....	85

PowerGREP Reference87

1. PowerGREP Assistant.....	89
2. File Selector Reference	91
3. Import File Listings.....	96
4. File Selector Menu.....	98
5. Action Reference	105
6. Action Types	107
7. Search Terms and Options	114
8. Action Part: Filter Files.....	121
9. Action Part: File Sectioning	124
10. Main Part of The Action	128
11. Action Part: Extra Processing	130
12. Action Part: Context.....	131
13. Action Part: Collect Between	134
14. Action Part: Target and Backup Files	136
15. Action Parts and Named Capture.....	140
16. Action Menu.....	141
17. Sequence Reference.....	145
18. Sequence Menu.....	148
19. Library Reference	153
20. Library Menu.....	155
21. Results Reference	157
22. Results Menu.....	160
23. Editor Reference.....	168
24. Editor Menu.....	171
25. Undo History Reference	176
26. Undo History Menu.....	178
27. Change PowerGREP's Appearance	180
28. Share Experiences and Get Help on The User Forums	184
29. Forum RSS Feeds.....	188
30. File Selector Preferences	189
31. File Formats Preferences.....	192
32. Archive Formats Preferences	197
33. Action Preferences	201
34. Text Layout Configuration	204
35. Text Cursor Configuration.....	209

36. Text Encoding Preferences.....	212
37. Results Preferences.....	216
38. Editor Preferences.....	218
39. External Editors Preferences.....	221
40. General Preferences.....	223
41. Color Configuration.....	225
42. Match Placeholders.....	229
43. Path Placeholders.....	237
44. Command Line Parameters.....	240
45. XML Format of PowerGREP Files.....	247

Regular Expression Tutorial..... 249

1. Regular Expression Tutorial.....	251
2. Regex Tutorial Table of Contents.....	253
3. Literal Characters.....	256
4. First Look at How a Regex Engine Works Internally.....	258
5. Character Classes or Character Sets.....	260
6. The Dot Matches (Almost) Any Character.....	264
7. Start of String and End of String Anchors.....	266
8. Word Boundaries.....	270
9. Alternation with The Vertical Bar or Pipe Symbol.....	273
10. Optional Items.....	275
11. Repetition with Star and Plus.....	276
12. Use Round Brackets for Grouping.....	279
13. Named Capturing Groups.....	284
14. Unicode Regular Expressions.....	286
15. Regex Matching Modes.....	295
16. Possessive Quantifiers.....	297
17. Atomic Grouping.....	300
18. Lookahead and Lookbehind Zero-Width Assertions.....	302
19. Testing The Same Part of a String for More Than One Requirement.....	306
20. Continuing at The End of The Previous Match.....	308
21. If-Then-Else Conditionals in Regular Expressions.....	310
22. XML Schema Character Classes.....	313
23. POSIX Bracket Expressions.....	315
24. Adding Comments to Regular Expressions.....	319
25. Free-Spacing Regular Expressions.....	320

Regular Expression Examples..... 321

1. Sample Regular Expressions.....	323
2. Matching Floating Point Numbers with a Regular Expression.....	326
3. How to Find or Validate an Email Address.....	327
4. Matching a Valid Date.....	330
5. Finding or Verifying Credit Card Numbers.....	332
6. Matching Whole Lines of Text.....	334
7. Deleting Duplicate Lines From a File.....	335

9. Find Two Words Near Each Other	336
10. Runaway Regular Expressions: Catastrophic Backtracking.....	337
11. Repeating a Capturing Group vs. Capturing a Repeated Group.....	343
12. Mixing Unicode and 8-bit Character Codes.....	344

Regular Expression Reference..... 347

1. Basic Syntax Reference	349
2. Advanced Syntax Reference.....	354
3. Unicode Syntax Reference	358
4. Syntax Reference for Specific Regex Flavors.....	359
5. Regular Expression Flavor Comparison.....	361
6. Replacement Text Reference	372

Part 1

How to Use PowerGREP

1. Introducing PowerGREP

PowerGREP is a versatile and powerful text processing and search tool based on regular expressions. A regular expression is a pattern that describes the form of a piece of text. E.g. a regular expression could match a date or an email address. *Any* date or *any* email address that is, without specifying actual dates or actual email addresses. Your search patterns can be as specific or as general as you want. This makes PowerGREP much more flexible than a general search tool that only finds words and phrases (PowerGREP can do that too).

With PowerGREP you can use one or more such regular expressions to get lists of files, lists of search matches in files, search-and-replace through files, rename files, merge files, split files, etc. First read the “how to use PowerGREP” section to get a feel of the way PowerGREP works. Then check out the examples that seem interesting to you. All examples include step-by-step instructions. The examples don’t require prior experience with PowerGREP, but you’ll understand them better if you check out the “how to” section first.

Contents of This Manual

The PowerGREP manual consists of six parts:

1. How to use PowerGREP: General, step-by-step instructions on how to use PowerGREP’s various functionality. The most important options are explained.
2. PowerGREP Examples: Step-by-step instructions explaining how to perform specific tasks with PowerGREP. The examples cover most of PowerGREP’s functionality, giving you a good idea of PowerGREP’s capabilities.
3. PowerGREP Reference: Detailed information about all of PowerGREP’s capabilities. Each on-screen control and each menu item is explained in detail. Also explains how to configure PowerGREP, and sheds light on PowerGREP’s inner workings.
4. Regular Expression Tutorial: Detailed tutorial on regular expressions. All aspects of regular expressions are explained, from most common to most specialized.
5. Regular Expression Examples: Examples illustrating how to build a regular expression from scratch.
6. Regular Expression Reference: Brief reference of the various regular expression tokens.

2. Contact PowerGREP's Developer and Publisher

PowerGREP is developed and published by Just Great Software Co. Ltd.

For the latest information on PowerGREP, please visit the official web site at <http://www.powergrep.com/>.

Before requesting technical support, please use the Check New Version command in the Help menu to see if you are using the latest version of PowerGREP. We take pride in quickly fixing bugs and resolving problems in free minor updates. If you encounter a problem with PowerGREP, it is quite possible that we have already released a new version that no longer has this problem.

PowerGREP has a built-in Forum feature that allows you to easily communicate with other PowerGREP users. If you're having a technical problem with PowerGREP, you're likely not the only one. The problem may have already been discussed on the forums. So search there first and you may get an immediate answer. If you don't see your issue discussed, feel free to start a new conversation in the forum. Other PowerGREP users will soon chime in, probably even before a Just Great Software technical support person sees it.

However, if you have purchased PowerGREP, you are entitled to free technical support via email. The technical support only covers the installation and use of PowerGREP itself. In particular, technical support does not cover learning and using regular expressions. The online forum does have a group devoted to learning regular expressions though.

To request technical support, please use the Support and Feedback command in the Help menu. This command will show some basic information about your computer and your copy of PowerGREP. Please copy and paste this information into your email, as it will help us to respond more quickly to your inquiry. If the problem is that you are unable to run PowerGREP, and thus cannot access the Support and Feedback command, you can email support@powergrep.com. You can expect to receive a reply by the next business day. For instant gratification, try the forums.

If you have any comments about PowerGREP, good or bad, suggestions for improvements, please do not hesitate to send them to our technical support department. Or better yet: post them to the forum so other PowerGREP users can add their vote. While we cannot implement each and every user wish, we do take all feedback into account when developing new versions of our software. Customer feedback is an essential part of Just Great Software.

Where to Buy (More Copies of) PowerGREP

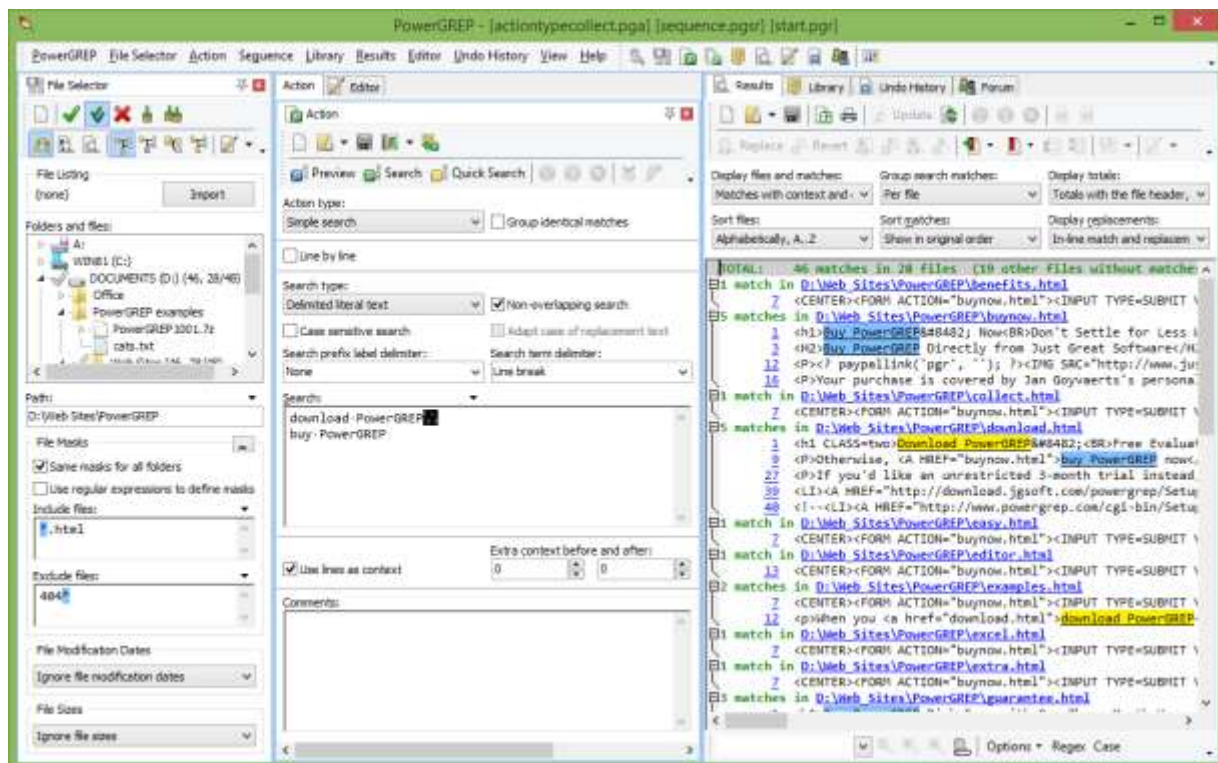
To buy a single user or site license to PowerGREP, please visit <http://www.powergrep.com/buynow.html> for a complete list of current purchasing options, and up to date pricing information. If you already have a license but want to expand it to more users, please go to <http://www.powergrep.com/multiuser.html>. If you have any questions about buying PowerGREP not answered on that page, please contact sales@powergrep.com.

3. Getting Started with PowerGREP

I don't exaggerate when I say that PowerGREP is the most powerful and versatile regular expression search and text processing tool available worldwide today. But that doesn't mean PowerGREP is complicated or difficult to use. While it will certainly take some practice to get the most out of PowerGREP, this "getting started" section will show you PowerGREP is surprisingly convenient to use.

1. You start with telling PowerGREP which files you want to work with. Click on a file or folder in the File Selector. Then select Include File or Folder in the File Selector menu to mark the file or folder to be searched through. Marking a folder is a quick way to work with all the files in that folder. Later I will show you how you can search through only certain files in a folder without marking them individually.
2. Then, you tell what PowerGREP should do with those files, by by defining an action on the Action panel. For a simple search, select **"simple search"** in the drop-down list labeled "action type" at the top of the Action panel. For a search-and-replace, select **"search-and-replace"** in the "action type" list. If you just want to search for some text, select "literal text" as the "search type". Enter the text you want to find in the search box.
3. Click the Preview button in the toolbar to start the search.
4. Inspect the search results on the Results panel. Double-click on a match to open the file in the editor and see its context. If you've previewed or executed a search-and-replace, you can make or revert some or all replacements via the Results and Editors menus or toolbars.

That's all it takes! PowerGREP's power and complexity remain hidden when you don't need it, making PowerGREP surprisingly easy to use.



4. Mark Files for Searching

The first step in running a search with PowerGREP is to select the files you want to search through in the File Selector.

1. If you already have a text file with a list of files and/or folders that you want search through, click the Import button to show the Import File Listings screen. There you can select one or more text files to read file listings from. Then you can skip ahead to step 8.

2. To include an individual file in the search, click on the file in the tree of files and folders, and then select the Include File or Folder item in the File Selector menu, or click the corresponding button on the File Selector toolbar. A green tick will appear next to the file.

3. To include a folder, and all the files in that folder, click on the folder and use the same Include File or Folder command. A green tick will appear next to the folder. Gray ticks will appear next to the files in the folder.

4. To include a folder, all the files in that folder, and all the files in all subfolders in that folder, click on the folder and then select Include Folder and Subfolders in the File Selector menu. A double green-blue tick will appear next to the folder. Gray ticks will appear next to the files. Double gray tick will appear next to the subfolders.

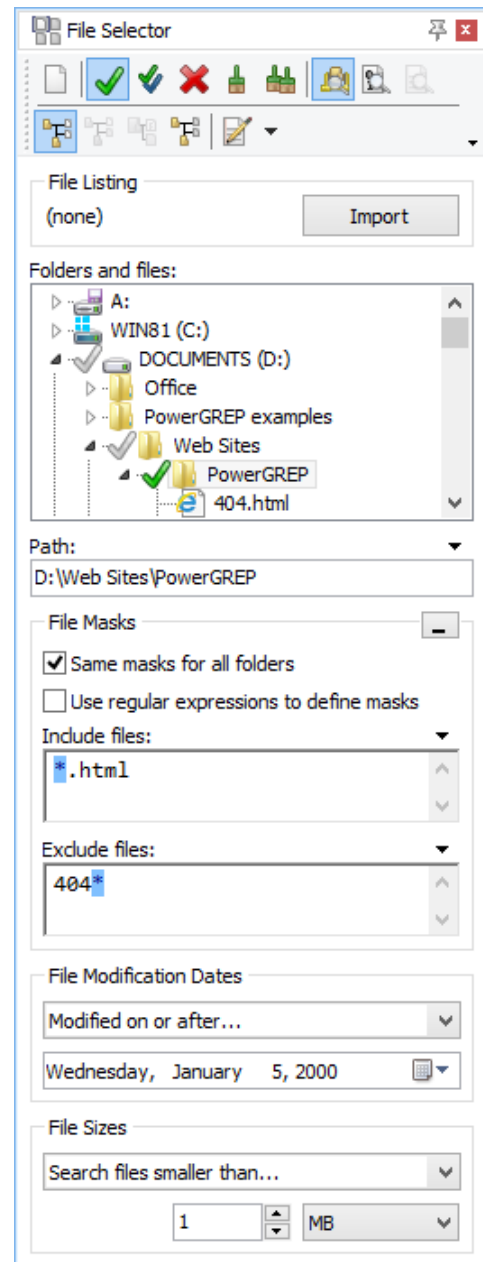
5. To exclude a file that has a gray tick because you included its folder, click on the file and select Exclude File or Folder from the menu.

6. To exclude folder that has a double gray tick because you included its parent folder, click on that folder and select Exclude File or Folder. Files and folders in that folder will be excluded as well.

7. If you change your mind about including or excluding a file or folder, click on it and then select Clear File or Folder. To remove all markings, select the Clear item in the File Selector menu.

8. If you want to search through files of particular types only, enter a semicolon-delimited list of file types in the “include files” box. E.g. enter *.txt;*.html to search through text files and HTML files only. To exclude certain types of files, enter their file types in the “exclude files” box. The File Selector reference explains the file masks you can use in the include files and exclude files boxes in full detail.

9. If you only want to search through recently modified files, select "modified during the past..." in the File Modification Dates section. Then you can enter the number of hours, days, weeks, months or years. Other date options allow you to restrict the search to files last modified during a certain date range.



10. Finally, if you only want to search through files of certain sizes, specify the sizes you want in the File Sizes section.

The next step is to define the action you want to run on the files you just marked.

After running the search, you can further narrow down the search results with the Mark Files with Search Results command or the Search Only through Files with Results option.

5. Define a Search Action

1. Start with selecting the kind of action you want to execute in the “action type” drop-down list in the upper left corner of the Action panel. As soon as you do so, the Action panel will rearrange itself slightly. Not all options are available for all action types.

- simple search: Display all search matches in each file, so you can inspect each search match and its context.
- search: Display all search matches, with additional options for file sectioning and target files.
- collect data: Collect a piece of text based on the search match using replacement text syntax. Create a new file with all the collected text, or create separate files for each source file.
- list files: Display a list of files matching, or not matching, the search criteria. The fastest search method. Allows you to copy, move, delete, zip, and unzip the listed files via the target file creation setting.
- rename files: Rename files or move files into different folders by searching and replacing through the file’s name or path.
- search-and-replace: Replace all search matches in each file, modifying either the original file or a copy of it.
- search-and-delete: Delete all search matches in each file, modifying either the original file or a copy of it.
- merge files: Collect the text of all files in which a search match is found into one or more new files.
- split files: Collect search matches into new files, using replacement text syntax to specify the target file for each search match.

2. Select the kind of search term you want to use from the “search type” list. Again, the Action panel will rearrange itself so you can enter that kind of search term. PowerGREP supports four kinds of search terms, which you can enter in three ways:

- Literal text: Any piece of text; words, phrases, whatever.
 - Regular expression: A pattern describing the form of the text you want to match. This is the most powerful way to search. Read the regular expression tutorial to learn everything about regular expressions.
 - Free-spacing regular expression: A regular expression that ignores spaces and comments in the pattern, allowing you to format it freely.
 - Binary data: Arbitrary data that you can enter in hexadecimal mode. Useful for searching through binary files.
-
- Single item: Enter one piece of text, one regular expression, or one chunk of binary data.
 - List of items: Separately enter as many search items as you want. Choose this method to key in multiple items in PowerGREP.
 - Delimited items: Enter multiple search items all together, delimited by whichever characters you want. Choose this method if you want to copy and paste an already delimited list of items into PowerGREP.

3. Toggle search options:

- Non-overlapping search: When searching for a list of items, turn on non-overlapping search to process each file only once, searching for all items at the same time. See the reference section in this book learn the implications of overlapping search.

- Case sensitive search: Turn on if the difference between uppercase and lowercase letters is significant.
- Adapt case of replacement text: Make the replacement text automatically all lowercase, all uppercase or all title case depending on the search match. Only available for search-and-replace and “collect data” actions.
- Dot matches newlines: When searching for a regular expression, make the dot match all characters, including line breaks.
- Whole words only: Only return search matches that consist of one or more complete words.
- List only files matching all terms: Display only files matching all search terms. Only available when the action type is “list files” or “search”, and the search type is a list.

4. When the action type is “search” or “collect data”, specify how matches should be collected.

- Group results for all files: Save only one output file with all the collected matches, rather than creating one file for each file searched through.
- Group identical matches: Collect identical matches only once in each output file (i.e. once for the whole action when grouping results for all files, otherwise once for each file searched through). If you turn off the grouping options, all matches are collected in the order they are found.
- Sort collected matches: When grouping matches, select to save them into the output file in alphabetic order, or sorted by the number of times each match was found.
- Minimum number of occurrences: When grouping matches, do not save matches that occur fewer times than you specify.

5. If you want to exclude some files from the action based on their contents, use the “filter files” option. An additional set of controls for entering search terms will appear.

6. Select a file sectioning option if you don’t want to search through entire files. An additional set of controls for entering search terms will appear. Select the “split along delimiters” sectioning type to make the main action (steps 1 through 4 above) process only those parts of each file *between* the matches of the sectioning search terms. Select “search for sections” to make the main action process the matches of the sectioning search terms. To really make use of “search for sections”, the sectioning search term should be a regular expression.

7. When sectioning a file, additional options affecting the main action (steps 1 through 4 above) are available.

- Match whole sections only: Only return search matches of the main part of the action that match whole sections.
- Collect/replace whole sections: When making replacements or collecting data, replace or collect the whole section, even if the main action search terms match the section only partially.
- Invert search results: Make the main action match sections in which the main search terms cannot be found. Only available in combination with "collect/replace whole sections". If the action type is “list files”, this option has a different meaning, since the main action does not involve individual search matches. In “list files” mode, inverting search results makes PowerGREP lists those files in which the main action’s search terms cannot be found.
- List only sections matching all items: Retain only matches from sections in which all the search terms of the main action can be found. Search matches found in sections that contain only some of the search terms are discarded.

8. Turn on “extra processing” if you want to apply an extra search-and-replace to the replacement text in a search-and-replace action, or the text to be collected in a “collect data” action. When you do so, an extra set

of controls for entering search terms will appear. This second search-and-replace will be executed on the replacement text or on the text to be collected, each time the main search finds a match.

9. If you plan to study the search results on the Results panel in PowerGREP, you can make things easier by collecting extra context before and/or after the match. Context is only used for display purposes on the Results panel.

10. Specify target file options. If the action type is “list files”, PowerGREP can save the file names of the files that are found into a target file. If the action type is “search-and-replace”, the target settings determine if the replacements are made in the files being searched through, or in copies of those files. When the action type is “collect data”, PowerGREP will save search matches or collected text to into either a single target file for the whole action, or into one file for each file searched through.

11. When creating target files, set the backup file options to make sure backup copies are made when files are overwritten. Backup copies are required to be able to undo action in the Undo History.

12. To test the action, click the Preview button in the Action toolbar. PowerGREP will execute the search without creating or overwriting any files, or doing anything else you might regret. Click the Execute button to execute the action for real. Click the Quick Execute button to save time when you don’t need full details of the search results.

When PowerGREP finishes running the search, a full report will appear on the Results panel. Unless you used the Quick Execute button, the results will be highly detailed.

If you want to add the action to a library or save it into an action file, enter a description of the action in the Comments field to help you remember the purpose of the action.

6. Interpret Search Results

When you have executed an action, detailed search results are available on the Results panel. Inspecting the results is quite straightforward.

1. Set the display options you want. Either turn on Automatic Update, or click the Update button on the toolbar after changing the options.

- Display files and matches: Select whether to display file names and/or individual search matches. Individual search matches can be shown with or without context.
- Group search matches: Group matches per file to display the matches of each file, using the name of the file as a header. Group unique matches to see identical matches only once per file, or only once for the whole result set.
- Display totals: Show totals for the whole operation before or after the results. When grouping per file, show totals for each file before or after the matches in that file. When grouping unique matches, toggle indicating the number of times each match was found.
- Sort files: When grouping per files, sort the files alphabetically by full path, or by number of search matches found in each file.
- Sort matches: Display matches in the order they were found (when not grouping unique matches), or sort them alphabetically (grouping or not), or by the number of times each unique match was found (grouping or not).
- Display replacements: For “search-and-replace” and “collect data” actions, show either the original search match, or the replacement or collected text, or both. When showing both, you can show both on the same line, or on separate lines. When showing them separately, and showing context, the context is shown twice.

2. Use the Font and Text Direction and Word Wrap items in the Results menu to control the appearance of text in the results.

3. Double-click on a match to open it in the file editor to inspect its context. When grouping unique matches, double-clicking a match shows the individual match results at the bottom of the Results panel. Double-click on an individual match to see its context in the file editor.

If you turned on “group identical matches” on the Action panel, then double-clicking on matches in the Results will highlight the collected text in the target file. If you did not create target files, double-clicking on a match has no effect, because individual match details were discarded.

4. If you previewed a search-and-replace action, you can replace search matches by selecting a block of text that includes the matches you want to replace and then pressing the Replace button on the Results toolbar. You can replace all matches in the file that the text cursor points to with the Make Replacements in This File item in the Results menu. You can replace all matches in all files with Make Replacements in All Files. The highlight color changes in the editor and in the results to indicate the match was replaced.

5. If you executed a search-and-replace, you can restore the original text by selecting a block of text that includes the replacements you want to cancel and then pressing the Revert button on the Results toolbar. You can also restore individually replaced matches (see step 4) this way. You can restore the original text of all matches in the file the cursor points to with the Revert Replacements in This File item in the Results menu. You can restore everything with Make Replacements in All Files.

When replacing or reverting matches on the Results panel on a file that you have open in the file editor, the matches are replaced or reverted in the Editor. Your changes aren't saved until you save the file in the Editor, either by clicking the Save button or choosing to save the file when prompted upon closing the file.

When replacing or reverting matches on the Results panel on files that you don't have open in the file editor, PowerGREP automatically saves the changes. These automatic saves can be undone via the Undo History just like files saved in the file editor.

7. Edit Files and Replace or Revert Individual Matches

1. First open the file you want to edit. The quickest ways are double-clicking on a match or a file name in the results, or right-clicking on a file in the File Selector and selecting Edit File from the context menu.
2. Use the Font and Text Direction, Word Wrap, Line Numbers and Auto Indent items in the Editor menu to adjust the editor to the way you want to edit the file you just opened.
3. The editor highlights search matches. There are only three situations in which matches aren't highlighted: you used Quick Execute, the action works on whole files (action type set to "list files" or "merge files"), or you turned on "group identical matches" in the action definition. In all three cases, PowerGREP does not retain information about individual matches.
4. To jump to the previous or next match in the file, use the Next Match and Previous Match buttons on the Editor toolbar.
5. If you previewed a search-and-replace action, you can replace a search match by double-clicking on it. You can replace multiple matches by selecting a block of text that includes them and then pressing the Replace button on the Editor toolbar. You can replace all matches in the file with the Make All Replacements item in the Editor menu. The matches are instantly replaced with the replacement text that you prepared in the action definition. The highlight color changes in the editor and in the results to indicate the match was replaced.
6. If you executed a search-and-replace, you can restore the original text by double-clicking on a match that was replaced. You can also restore individually replaced matches (see step 5) this way. You can restore the original text of multiple replacements by selecting a block of text that includes them and then pressing the Revert button on the Editor toolbar. You can revert all matches in the file with the Revert All Replacements item in the Editor menu.
7. To replace a match or a replacement with other text, simply edit the text like you would in any other text editor. If you type into the middle of a match or partially delete a match, PowerGREP adjusts the highlighting to keep the edited match highlighted. As long as part of the match is still highlighted, you can replace or revert the highlighted text as explained in steps 5 and 6.

8. Keyboard Shortcuts

All frequently used PowerGREP commands have keyboard shortcuts associated with them. The key combinations are indicated next to the menu items in the main menu. The fly-over hints that appear when you hover the mouse over a toolbar button also indicate keyboard shortcuts.

Some key combinations are associated with only a single command. Pressing such a key combination invokes the command, regardless of where you are in PowerGREP. E.g. F9 is only associated with the Action|Preview menu item. At any time, pressing F9 will start a preview of the action.

Other key combinations are associated with multiple commands. All commands that share a given keyboard shortcut perform conceptually the same task, but in a different area. Which command is executed when you press the keys depends on which panel has keyboard focus. E.g. Ctrl+P is associated with Results|Print and Editor|Print. If you press Ctrl+P while inspecting the results, PowerGREP will print the results. If you press Ctrl+P while editing a file in the editor, PowerGREP will print the file you're editing. If you press Ctrl+P while selecting files or editing the action definition, nothing will happen.

See the editor reference for a list of key combinations you can use to edit text in multi-line text boxes in PowerGREP. In addition to the editor box on the Editor panel, all boxes for search terms on the Action panel are full-featured text editing controls. So is the results display, except that it is read-only.

Keyboard Navigation

Press the Tab key on the keyboard to walk through all controls presently visible in PowerGREP. Press Shift+Tab to walk backwards. To enter a tab character into the search terms, press Ctrl+Tab.

You can quickly move the keyboard focus to a particular panel by pressing the panel's keyboard shortcut as indicated in the View menu. E.g. press Alt+2 to activate the File Selector, Alt+3 for the Action panel, and Alt+6 for the Results. These keyboard shortcuts, and the associated menu items, activate the panel whether it was already visible or not, making it visible if necessary. If you like to use the keyboard rather than the mouse, memorizing the Alt+1 through Alt+9 key combinations will greatly speed up your work with PowerGREP.

Selecting Files and Folders with The Keyboard

Instead of navigating the folder tree, you can directly type in a path in the Path field just below the folder tree in the File Selector. The tree will automatically follow you as you type. You can also paste in a path from the clipboard.

To include the path you entered in the search, press Ctrl+I (Include File or Folder) or Shift+Ctrl+I on the keyboard. To include multiple paths, type in the first path and press (Shift+)Ctrl+I. The text in the Path field will become selected, so you can immediately type in the second path, replacing the first. Press (Shift+)Ctrl+I again to include the second path. To start over, press Ctrl+N to clear the file selection.

9. Regular Expression Quick Start

This quick start will quickly get you up to speed with regular expressions. Obviously, this brief introduction cannot explain everything there is to know about regular expressions. For detailed information, consult the regular expression tutorial. Each topic in the quick start corresponds with a topic in the tutorial, so you can easily go back and forth between the two.

Text Patterns and Matches

A regular expression, or regex for short, is a pattern describing a certain amount of text. In this book, regular expressions are printed between guillemots: «regex».

This first example is actually a perfectly valid regex. It is the most basic pattern, simply matching the literal text „regex”. Matches are indicated by double quotation marks, with the left one at the base of the line.

I will use the term “string” to indicate the text that I am applying the regular expression to. I will indicate strings using regular double quotes.

Literal Characters

The most basic regular expression consists of a single literal character, e.g.: «a». It will match the first occurrence of that character in the string. If the string is “Jack is a boy”, it will match the „a” after the “J”.

This regex can match the second „a” too. It will only do so when you tell the regex engine to start searching through the string after the first match. In a text editor, you can do so by using its “Find Next” or “Search Forward” function. In a programming language, there is usually a separate function that you can call to continue searching through the string after the previous match.

Eleven characters with special meanings: the opening square bracket «[», the backslash «\», the caret «^», the dollar sign «\$», the period or dot «.», the vertical bar or pipe symbol «|», the question mark «?», the asterisk or star «*», the plus sign «+», the opening round bracket «(» and the closing round bracket «)». These special characters are often called “metacharacters”.

If you want to use any of these characters as a literal in a regex, you need to escape them with a backslash. If you want to match „1+1=2”, the correct regex is «1\\+1=2». Otherwise, the plus sign will have a special meaning.

Character Classes or Character Sets

A “character class” matches only one out of several characters. To match an a or an e, use «[ae]». You could use this in «gr[ae]y» to match either „gray” or „grey”. A character class matches only a single character. «gr[ae]y» will not match “graay”, “graey” or any such thing. The order of the characters inside a character class does not matter.

You can use a hyphen inside a character class to specify a range of characters. «`[0-9]`» matches a *single* digit between 0 and 9. You can use more than one range. «`[0-9a-fA-F]`» matches a single hexadecimal digit, case insensitively. You can combine ranges and single characters. «`[0-9a-fxA-FX]`» matches a hexadecimal digit or the letter X.

Typing a caret after the opening square bracket will negate the character class. The result is that the character class will match any character that is *not* in the character class. «`q[^\x]`» matches „qu” in “quest ion”. It does *not* match “Iraq” since there is no character after the q for the negated character class to match.

Shorthand Character Classes

«`\d`» matches a single character that is a digit, «`\w`» matches a “word character” (alphanumeric characters plus underscore), and «`\s`» matches a whitespace character (includes tabs and line breaks). The actual characters matched by the shorthands depends on the software you’re using. Usually, non-English letters and numbers are included.

Non-Printable Characters

You can use special character sequences to put non-printable characters in your regular expression. Use «`\t`» to match a tab character (ASCII 0x09), «`\r`» for carriage return (0x0D) and «`\n`» for line feed (0x0A). More exotic non-printables are «`\a`» (bell, 0x07), «`\e`» (escape, 0x1B), «`\f`» (form feed, 0x0C) and «`\v`» (vertical tab, 0x0B). Remember that Windows text files use “`\r\n`” to terminate lines, while UNIX text files use “`\n`”.

Use «`\xFF`» to match a specific character by its hexadecimal index in the character set. E.g. «`\xA9`» matches the copyright symbol in the Latin-1 character set.

If your regular expression engine supports Unicode, use «`\uFFFF`» to insert a Unicode character. E.g. «`\u20AC`» matches the euro currency sign.

All non-printable characters can be used directly in the regular expression, or as part of a character class.

The Dot Matches (Almost) Any Character

The dot matches a single character, except line break characters. It is short for «`[^\n]`» (UNIX regex flavors) or «`[^\r\n]`» (Windows regex flavors). Most regex engines have a “dot matches all” or “single line” mode that makes the dot match any single character, including line breaks.

«`gr.y`» matches „gray”, „grey”, „gr%y”, etc. Use the dot sparingly. Often, a character class or negated character class is faster and more precise.

Anchors

Anchors do not match any characters. They match a position. «[^]» matches at the start of the string, and «^{\$}» matches at the end of the string. Most regex engines have a “multi-line” mode that makes «[^]» match after any line break, and «^{\$}» before any line break. E.g. «[^]b» matches only the first „b” in “bob”.

«^{\b}» matches at a word boundary. A word boundary is a position between a character that can be matched by «^{\w}» and a character that cannot be matched by «^{\w}». «^{\b}» also matches at the start and/or end of the string if the first and/or last characters in the string are word characters. «^{\B}» matches at every position where «^{\b}» cannot match.

Alternation

Alternation is the regular expression equivalent of “or”. «^{cat|dog}» will match „cat” in “About cats and dogs”. If the regex is applied again, it will match „dog”. You can add as many alternatives as you want, e.g.: «^{cat|dog|mouse|fish}».

Repetition

The question mark makes the preceding token in the regular expression optional. E.g.: «^{colou?r}» matches „colour” or „color”.

The asterisk or star tells the engine to attempt to match the preceding token zero or more times. The plus tells the engine to attempt to match the preceding token once or more. «^{<[A-Za-z][A-Za-z0-9]*>}» matches an HTML tag without any attributes. «^{<[A-Za-z0-9]+>}» is easier to write but matches invalid tags such as „<1>”.

Use curly braces to specify a specific amount of repetition. Use «^{\b[1-9][0-9]{3}\b}» to match a number between 1000 and 9999. «^{\b[1-9][0-9]{2,4}\b}» matches a number between 100 and 99999.

Greedy and Lazy Repetition

The repetition operators or quantifiers are greedy. They will expand the match as far as they can, and only give back if they must to satisfy the remainder of the regex. The regex «^{<.+>}» will match „first” in “This is a first test”.

Place a question mark after the quantifier to make it lazy. «^{<.+?>}» will match „” in the above string.

A better solution is to follow my advice to use the dot sparingly. Use «^{<[^<>+>}» to quickly match an HTML tag without regard to attributes. The negated character class is more specific than the dot, which helps the regex engine find matches quickly.

Grouping and Backreferences

Place round brackets around multiple tokens to group them together. You can then apply a quantifier to the group. E.g. «Set(Value)?» matches „Set” or „SetValue”.

Round brackets create a capturing group. The above example has one group. After the match, group number one will contain nothing if „Set” was matched or „Value” if „SetValue” was matched. How to access the group’s contents depends on the software or programming language you’re using. Group zero always contains the entire regex match.

Use the special syntax «Set(?:Value)?» to group tokens without creating a capturing group. This is more efficient if you don’t plan to use the group’s contents. Do not confuse the question mark in the non-capturing group syntax with the quantifier.

Unicode Properties

«\p{L}» matches a single character that has a given Unicode property. L stands for letter. «\P{L}» matches a single character that does not have the given Unicode property. You can find a complete list of Unicode properties in the tutorial.

Lookaround

Lookaround is a special kind of group. The tokens inside the group are matched normally, but then the regex engine makes the group give up its match and keeps only the result. Lookaround matches a position, just like anchors. It does not expand the regex match.

«q(?=u)» matches the „q” in “question”, but not in “Iraq”. This is positive lookahead. The «u» is not part of the overall regex match. The lookahead matches at each position in the string before a “u”.

«q(?!u)» matches „q” in “Iraq” but not in “question”. This is negative lookahead. The tokens inside the lookahead are attempted, their match is discarded, and the result is inverted.

To look backwards, use lookbehind. «(?<=a)b» matches the „b” in “abc”. This is positive lookbehind. «(?<!a)b» fails to match “abc”.

You can use a full-fledged regular expression inside the lookahead. Most regular expression engines only allow literal characters and alternation inside lookbehind, since they cannot apply regular expressions backwards.

Part 2

PowerGREP Examples

1. Search Through File Names

Normally, the File Selector is used to determine which files are included in the action, and the search terms on the Action panel are used to search through the contents of those files. But if you want to run a quick search through file names or file paths, you can set the “action type” on the Action panel to “file name search”. Then the search terms in the main part of the action are used to search through the names of the files rather than their contents.

1. Clear the file selection.
2. Click on the folder that contains the files you want to search through. Then select Include File or Folder or Include Folder and Subfolders from the File Selector menu.
3. Repeat step 2 if you want to search through the files in multiple folders.
4. Start with a fresh action.
5. Set the action type to “file name search”.
6. Enter one or more search terms to look for in the file names.
7. Click the Preview button to run the action.

The Results panel will show a list of files of which the names contain one or more of the search terms from step 6. If you want to get a list of files *not* having any of the search terms in their names, turn on the “invert results” checkbox on the Action panel after setting the action type to “file name search”.

Using The File Selector to Search Through File Names

For complex operations where you want to use an action type other than “file name search” to manipulate the contents of the files, you can use the File Selector to include or exclude certain files by searching through their names. This example uses the “list files” action type without the search term to demonstrate how searching through file names using the File Selector works.

1. Clear the file selection.
2. Click on the folder that contains the files you want to search through. Then select Include File or Folder or Include Folder and Subfolders from the File Selector menu.
3. Repeat step 2 if you want to search through the files in multiple folders.
4. Turn on “use regular expressions to define masks”, even if you want to search for a simple word or phrase.
5. Enter your search terms in the “include files” box, delimited by semicolons.
6. Start with a fresh action.
7. Set the action type to “list files”.
8. Leave the Search box blank.
9. Click the Preview button to run the action.

The Results panel will show a list of files of which the names contain one or more of the search terms from step 5.

If you want to get a list of files *not* having any of the search terms in their names, enter the search terms from step 5 in the “exclude files” box instead. If you enter search terms in both boxes, you will get a list of files having one or more search terms from “include files”, and none of the search terms from “exclude files” in their names.

To search for different file names in different folders, turn off “same masks for all folders”. Then click on a folder to specify “include files” and “exclude files” for that folder only. Repeat for all other folders you marked in step 2.

How to Search through Both Names and Contents

To search through both the names of the files, and their contents, go through steps 1 through 6 above to make the file selector search through the file names, and then proceed as follows:

7. Leave the action type as “simple search”.
8. On the Action panel, enter the search terms you want to search for through the contents of the files.
9. Click the Preview button to run the action.

PowerGREP will then search through the contents of those files of which the names contain one or more of the search terms from step 5. A file will only be listed in the results if both its name matches the terms of step 5, and its contents match the terms of step 8.

PowerGREP cannot produce a list of files that contain a search term in their name, or contain the search term in their contents, but do not contain the search term in both name and contents. You will need to run two searches. One where you enter the search terms in the “include files” box and leave the search terms on the Action page blank, and another where you leave “include files” blank and enter the terms on the Action page.

2. Find Files Not Containing a Search Term

You can easily get a list of files *not* containing a particular search term by running a “list files” action and turning on the “invert search results” option.

1. Clear the file selection.
2. Click on the folder that contains the files you want to search through. Then select Include File or Folder or Include Folder and Subfolders from the File Selector menu.
3. Repeat step 2 if you want to search through the files in multiple folders.
4. Start with a fresh action.
5. Set the action type to “list files”.
6. Enter the search terms that the files should not contain.
7. Turn on the “invert search results” option.
8. Click the Preview button to run the action.

The Results panel will show a list of files that do not contain the search terms. If you entered a list of search terms, only files that do not contain any of the search terms will be listed.

3. Find Email Addresses

Searching for email addresses is easy with PowerGREP, and a very good example of the benefit of regular expressions. Instead of searching for a particular email address, with a regular expression you can search for *any* email address. If you forget somebody's address, simply search your correspondence. Getting a list of address of everybody you've communicated with is just as easy.

Finding a Particular, Unspecified Email Address

1. Select the files you want to search through in the File Selector.
2. Start with a fresh action.
3. Leave the action type as "simple search". Leave the search type as "regular expression".
4. In the search box, enter the regular expression `«\b[A-Z0-9._%+~]+@[A-Z0-9.-]+\.[A-Z]{2,6}\b»` and make sure to leave "case sensitive search" off.
5. Click the Preview button to run the action.

When the action completes running, you will get the list of email addresses on the Results panel. If the list is long, you can sort out the addresses as follows:

6. Select "per unique match" from the "group search matches" list, and click the Update button. Each email address now appears only once in the list.
7. Select "matches with context" from the "display files and matches" list. This will affect the way the details are displayed in the next step.
8. Double-click on an email address to see in which files it occurs. The details will appear in the bottom part of the Results panel.
9. Double-click on an address in the details to open the document it was found in. Check the document to see if it is the address you want.
10. If not, switch back the Results panel and repeat from step 8.

How to Get a List of all Email Addresses

When you follow the above steps, you will get a list of all addresses in step 6. If you want to save the results into a file, you can either copy-and-paste the results from step 6 above into a text editor, or you can make PowerGREP do this for you with the steps below.

1. Select the files you want to search through in the File Selector.
2. Start with a fresh action.
3. Set the action type to "collect data". Leave the search type as "regular expression".
4. In the search box, enter the regular expression `«\b[A-Z0-9._%+~]+@[A-Z0-9.-]+\.[A-Z]{2,6}\b»` and make sure to leave "case sensitive search" off.
5. Select "save results into a single file" in the target file creation list.
6. Click the ellipsis (...) button next to "target file location", and select the file you want to save the results into.
7. If you want a comma-delimited list of addresses for use with email software, set "between collected text" to "same text between all matches". Then type in a comma in the box that appears below the "between collected text" drop-down list.
8. Click the Collect button to run the action.

This will produce the same results as in step 6 in the first method, with one difference: if you double-click an address in the results, PowerGREP will open the target file in the editor rather than the file the email address was found in.

4. How to Find Word Pairs

This example illustrates the use of lookaround in regular expressions. In the discussion below, the file being searched through contains the four words “one two three four”.

Matching two consecutive words with a regular expression is easy: `«\w+\s+\w+»`. But when you try this regex in a collect data action, PowerGREP will find only two pairs: „one two” and „three four”. The middle pair “two three” is missing. The reason is that when PowerGREP finds a search match, it continues searching at the end of the match. After matching „one two”, PowerGREP continues at the space after “two”.

The solution is to use lookahead for the second word. Lookahead applies the regex match as usual, but does not actually expand the match result to the text matched by the lookahead. When you collect data with `«\w+\s+(?=\w+)»` PowerGREP will find all three pairs, but collect only “one ”, „two ” and „three ”, trailing spaces included.

To also collect the text matched by the lookahead, we need to use a capturing group. This does not change the nature of the lookahead. To make the output prettier, we’ll also capture the first word. That allows us to collect both words separated by just one space, rather than by whatever was matched by `«\s+»`.

When we search for `«(\w+)\s+(?=(\w+))»` and collect “\1 \2” the results will list all 3 word pairs: “one two”, “two three” and “three four”. You may need to select “replacement only” in the “display replacements” list on the Results panel to remove the regex match from the results and show the collected pairs only.

You can take this example as far as you want. Search for `«(\w+)\s+(?=(\w+)\s+(\w+))»` and collect “\1 \2 \3” to gather word triplets.

5. Boolean Operators “and”, “or”, and “not”

Many search tools use the boolean operators "and", "or", and "not". Searching for “term1 and term2 and term3” results in a list of files in which all three search terms can be found. Searching for “term1 or term2 or term3” gives a list of files in which at least one of the three search terms occurs. Searching for “term1 and not term2” lists files that contain term1 but not term2, while "(term1 or term2) and not (term3 or term4)" lists files that contain term1 or term2 or both, but not term3 and not term4.

PowerGREP does not use boolean operators, but does offer similar functionality.

PowerGREP’s Implicit “or”

When you specify multiple search terms, PowerGREP automatically implies an “or” operator between them. That is, you will get a list of files containing one or more of the search terms.

1. Select the files you want to search through in the File Selector.
2. Start with a fresh action.
3. Set the search type to “list of literal text”.
4. Make sure “list only files matching all terms” is *off* to imply “or” between the search terms.
5. Enter the first search term. Click on the green plus button to add additional search terms to the list.
6. Click the Preview button to run the action.

List Only Files Matching All Terms

If you turn on the option "list only files matching all terms", PowerGREP will give you a list of files containing each search term at least once, as if you used a boolean “and” operator between the search terms. Files containing some but not all of the search terms will not be displayed in the results.

1. Select the files you want to search through in the File Selector.
2. Start with a fresh action.
3. Set the search type to “list of literal text”.
4. Make sure “list only files matching all terms” is *on* to imply “and” between the search terms.
5. Enter the first search term. Click on the green plus button to add additional search terms to the list.
6. Click the Preview button to run the action.

This action is available in the PowerGREP.pgl library as “List files containing all search terms”.

List Only Lines Matching All Terms

With the option "list only sections matching all items", you can find lines or any other kind of section containing each search term at least once, as if you used a boolean “and” operator between the search terms. Lines or sections containing some but not all of the search terms will not be displayed in the results. This option only appears when using file sectioning.

1. Select the files you want to search through in the File Selector.

2. Start with a fresh action.
3. Set the search type to “list of literal text”.
4. Enter the first search term. Click on the green plus button to add additional search terms to the list.
5. In the “file sectioning” list, select “line by line”.
6. Turn on the “list only section matching all terms” option that appears after choosing line by line sectioning. This implies “and” between the search terms.
7. Click the Preview button to run the action.

This action is available in the PowerGREP.pgl library as “Collect lines containing all search terms”.

Combining “and” and “or”

The “list only files matching all terms” and “list only sections matching all items” options are all-or-nothing options. When both are off, “or” is implied between all search terms. When either is on, “and” is implied between all search terms, at the file level or the section level.

For a combination of “and” and “or”, you will need to use regular expressions. Turn on “list only files/sections matching all items” to imply “and” between the regular expressions, as in the above examples. Then use the alternation regex operator to combine multiple search terms into a single regular expression. Alternation is the regex-equivalent of “or”.

E.g. for the boolean query “(Jack or John) and (Sue or Mary or Grace)”, you would need two regular expressions, as follows:

1. Select the files you want to search through in the File Selector.
2. Start with a fresh action.
3. Set the search type to “list of regular expressions”.
4. Make sure “list only files matching all terms” is *on* to imply “and” between the regular expressions.
5. Enter the first regular expression «Jack|John». If your search terms contain non-alphanumeric characters, make sure to escape characters that have a special meaning in regular expressions.
6. Click on the green plus button to add the regular expression «Sue|Mary|Grace».
7. Click the Preview button to run the action.

Files or Lines Not Matching Your Terms

A boolean search for “not a” or for “not a and not b” gets you a list of files that don’t contain your search term(s). In PowerGREP you can do this with the “invert search results” checkbox in the file sectioning part of the action. If the action type is list files, this option inverts the list of files. You get the list of files that don’t contain any of the search terms anywhere.

1. Select the files you want to search through in the File Selector.
2. Start with a fresh action.
3. Set the action type to “list files”.
4. Turn on the “invert results” option.
5. Make sure “list only files matching all terms” is *off* to imply “or” between the search terms. We’re searching for “not (a or b)” to exclude both a and b.
6. Set the search type to “list of literal text”.

7. Enter the first search term that the files must not contain. Click on the green plus button to add additional search terms to the list.
8. Click the Preview button to run the action.

For all other action types, the option is only available when dividing your files into sections using PowerGREP's file sectioning feature. Inverting the results then gives you a list of sections that don't contain the search terms. You can get all lines not containing any of your search terms as follows:

1. Select the files you want to search through in the File Selector.
2. Start with a fresh action.
3. Set the search type to "list of literal text".
4. Enter the first search term that the files must not contain. Click on the green plus button to add additional search terms to the list.
5. In the "file sectioning" list, select "line by line".
6. Turn on the "invert results" option that appears after choosing line by line sectioning.
7. Make sure "list only files matching all terms" is *off* to imply "or" between the search terms. We're searching for "not (a or b)" to exclude both a and b.
8. Click the Preview button to run the action.

Find Some Terms, Exclude Other Terms

You can emulate the boolean combination "and not" using PowerGREP's ability to use a second set of search terms to filter files prior to the actual search. The boolean query "(term1 or term2) and not (term3 or term4)" gets you a list of files that contain term1 or term2 or both, but not term3 and not term4. In PowerGREP, "term1 or term2" is the main search and "not (term3 or term4)" is the filtering condition. PowerGREP first searches for "term3 or term4". If those can't be found in a file, then PowerGREP searches for "term1 or term2".

1. Select the files you want to search through in the File Selector.
2. Start with a fresh action.
3. Set the action type to "list files".
4. Set "filter files" to "disallow any terms to match".
5. Set the search type for the file filter to "list of literal text".
6. Enter the first search term that the files must not contain. Click on the green plus button to add additional search terms to the list.
7. Set the search type for main part of the action to "list of literal text".
8. Enter the first search term that the files should contain. Click on the green plus button to add additional search terms to the list.
9. Click the Preview button to run the action.

PowerGREP supports only one "and not". If you have "term1 and not term2 and term3 and not term4" you need to rewrite the boolean query first. First, put all the negated terms together: "term1 and term3 and not term2 and not term4". Then use boolean algebra to put the "not" outside parentheses so only one "not" remains: "term1 and term3 and not (term2 or term4)". Now you can create a PowerGREP action for this using the steps above. To make the file match "term1 and term3", turn on "list only files matching all terms".

6. Find Two Words Near Each Other

Some search tools that use boolean operators also have a special operator called "near". Searching for "term1 near term2" finds all occurrences of term1 and term2 that occur within a certain "distance" from each other. The distance is a number of words. The actual number depends on the search tool, and is often configurable.

PowerGREP does not use the "near" operator. You can perform the same task with the proper regular expression.

Emulating "near" with a Regular Expression

With regular expressions you can describe almost any text pattern, including a pattern that matches two words near each other. This pattern is relatively simple, consisting of three parts: the first word, a certain number of unspecified words, and the second word. An unspecified word can be matched with the shorthand character class «\w+». The spaces and other characters between the words can be matched with «\W+» (uppercase W this time).

The complete regular expression becomes «\bword1\W+(?:\w+\W+){1,6}word2\b». The quantifier «{1,6}» makes the regex require at least one word between "word1" and "word2", and allow at most six words.

If the words may also occur in reverse order, we need to specify the opposite pattern as well: «\b(?:word1\W+(?:\w+\W+){1,6}word2|word2\W+(?:\w+\W+){1,6}word1)\b»

Two actions with these regular expressions are available in the PowerGREP.pgl library as "Find two words near each other (ordered)" and "Find two words near each other (unordered)".

7. Find Two or More Words on The Same Line

PowerGREP's file sectioning feature makes it trivial to find words that occur on the same line, or in any other kind of file section.

1. Select the files you want to search through in the File Selector.
2. Start with a fresh action.
3. Leave the action type as "simple search".
4. Enter two or more words as the search terms. Make sure to specify each word as a separate search term, by setting the search type to "delimited literal text" or "list of literal text". If you enter two or more words as a single search term, PowerGREP will search for that exact phrase, which is not what we want now.
5. Turn on "line by line".
6. Turn on the option "list only sections matching all items". This option only appears when you've entered multiple search terms. It tells PowerGREP to only display matches from sections (lines, in this case) in which all the words we're searching for can be found.
7. Click the Preview button to run the action.

This action is available in the PowerGREP.pgl library as "Find two or more words on the same line".

8. Search Through Printable Content in Word .docx Files

PowerGREP can search and even replace through Microsoft Office Open XML files (*.docx). These files are technically .zip archives containing one or more XML files and other supports files (such as image files). Though technically .zip archives, these files are considered documents and are searched through even when the Search through Archives option in the File Selector menu is turned off.

By default, PowerGREP uses the IFilter that is installed by Office 2007 to convert DOCX files to plain text. The advantage is that the search results show the document's text similar to how it appears in Word, except without formatting. The disadvantage is that the IFilter's conversion process is one-way. Thus PowerGREP cannot modify .docx files when using the IFilter.

If you don't have Office 2007, or if you turn off the option to use the IFilter for "zipped documents" in the archive formats preferences then PowerGREP displays and searches through the raw XML. Both the XML tags and the actual content of the document will be searched through. In this mode, PowerGREP can make replacements in the file, though you'll need to be careful not to upset the XML structure. The examples below are one way to take care of that.

You can use PowerGREP's file sectioning feature to search only through specific parts of a file, such as only the body text, as described in this example.

1. Clear the file selection.
2. Click on the folder that contains the files you want to search through. Then select Include File or Folder or Include Folder and Subfolders from the File Selector menu.
3. Enter *.docx in the "include files" box in the File Selector.
4. Start with a fresh action.
5. Set the action type to "search".
6. Enter the search terms that you want to find.
7. Select "search and collect sections" from the "file sectioning" list. Leave the section search type as "regular expression".
8. In the Section Search box, enter the regular expression «<w:t>([^\<]++)</w:t>». This regular expression matches a pair of <w:t> and </w:t> XML tags, and the text between them. In Word .docx files, all printable text is stored between such tags. Turn on "case sensitive search" in the file sectioning for better performance. XML tags are case sensitive.
9. In the Section Collect box, enter the backreference "\1" to restrict the main action to the contents of the <w:t> tag.
10. Click the Preview button to run the action.

Note that in .docx files, paragraphs with mixed formatting (bold, italics, etc.) are broken up into multiple <w:t> tags, one for each block of text with contiguous formatting. This means that the PowerGREP action above will process each contiguously formatted part of the paragraph in separate sections. The action will not find any search terms that span across sections.

This action is available in the PowerGREP.pgl library as "Office: Search printable text in Word .docx files (without IFilter)".

Searching and Replacing Through .docx Files

If you're going to use the above steps to do a search-and-replace rather than a search, you have to keep in mind that sharp brackets and the ampersand have a special meaning in XML. If your replacement text contains any of these, you need to use named entities `<`, `>`, and `&`.

One way to do this is to use PowerGREP's "extra processing" feature. Turn on this checkbox on the Action panel. Set its search type to "delimited literal text", set the "extra term delimiter" to a comma and the "extra pair delimiter" to an equals sign. Then you can type or paste `<=< ; ,>=> ; ,&=&` into the "extra processing search" box. Make sure "non-overlapping search" is turned on.

This example is available in the PowerGREP.pgl library as "Office: Search-and-replace printable text in Word .docx files (without IFilter)".

9. Search Through Printable Content in XPS Files

PowerGREP can search and even replace through XML Paper Specification files (*.xps). These files are technically .zip archives containing one or more XML files and other supports files (such as image files). Though technically .zip archives, these files are considered documents and are searched through even when the Search through Archives option in the File Selector menu is turned off.

By default, PowerGREP uses the IFilter that is included with Windows Vista and Windows 7 to convert XPS files to plain text. The advantage is that the search results show the document's text similar to how it would appear on a printout, except without formatting. The disadvantage is that the IFilter's conversion process is one-way. Thus PowerGREP cannot modify .xps files when using the IFilter.

If you have an older version of Windows without XPS support, or if you turn off the option to use the IFilter for "zipped documents" in the archive formats preferences then PowerGREP displays and searches through the raw XML. Both the XML tags and the actual content of the document will be searched through. In this mode, PowerGREP can make replacements in the file, though you'll need to be careful not to upset the XML structure. The example below is one way to take care of that.

You can use PowerGREP's file sectioning feature to search only through specific parts of a file, such as only the body text, as described in this example.

1. Clear the file selection.
2. Click on the folder that contains the files you want to search through. Then select Include File or Folder or Include Folder and Subfolders from the File Selector menu.
3. Enter *.xps in the "include files" box in the File Selector.
4. Start with a fresh action.
5. Set the action type to "search".
6. Enter the search terms that you want to find.
7. Select "search and collect sections" from the "file sectioning" list. Leave the section search type as "regular expression".
8. In the Section Search box, enter the regular expression «UnicodeString="([^\"]++)"». This regular expression matches a UnicodeString XML attribute and its double-quoted value. In XPS files, all printable text is stored in UnicodeString attributes. Turn on "case sensitive search" in the file sectioning for better performance. XML attribute names are case sensitive.
9. In the Section Collect box, enter the backreference "\1" to restrict the main action to the contents of the UnicodeString attributes.
10. Click the Preview button to run the action.

Note that in XPS files, paragraphs that span multiple lines in the printout are broken up into multiple XML tags, one per line. Lines with mixed formatting (bold, italics, etc.) are also broken up into multiple tags, one for each bit of text with contiguous formatting. This is because XPS is designed as a printer format rather than an editable or searchable format.

The above means that the PowerGREP action above will process each contiguously formatted part of each line in separate sections. The action will not find any search terms that span across sections.

10. Search Through Printable Content in OpenDocument Format Files

PowerGREP can search and even replace through OpenOffice files and LibreOffice files using the OpenDocument Format. These include database files (*.odb), chart files (*.odc), formula files (*.odf), graphics files (*.odg), image files (*.odi), presentation files (*.odp), spreadsheets (*.ods) and text documents (*.odt and *.odm). These files are technically .zip archives containing one or more XML files and other supports files (such as image files). Though technically .zip archives, these files are considered documents and are searched through even when the Search through Archives option in the File Selector menu is turned off.

By default, PowerGREP uses the IFilter that is included with OpenOffice and LibreOffice to convert ODF files to plain text. The advantage is that the search results show the document's text similar to how it would appear in OpenOffice, except without formatting. The disadvantage is that the IFilter's conversion process is one-way. Thus PowerGREP cannot modify ODF files when using the IFilter.

If you don't have one of these office suites installed or if you turn off the option to use the IFilter for "zipped documents" in the archive formats preferences then PowerGREP displays and searches through the raw XML. Both the XML tags and the actual content of the document will be searched through. In this mode, PowerGREP can make replacements in the file, though you'll need to be careful not to upset the XML structure.

You can use PowerGREP's file sectioning feature to search only through specific parts of a file, such as only the body text, as described in this example. When doing a search-and-replace through these files, you'll need to be careful not to upset the XML structure.

1. Clear the file selection.
2. Click on the folder that contains the files you want to search through. Then select Include File or Folder or Include Folder and Subfolders from the File Selector menu.
3. Enter *.odb;*.odc;*.odf;*.odg;*.odi;*.odp;*.ods;*.odt;*.odm in the "include files" box in the File Selector to search through all OpenOffice files. Enter fewer extensions if you only want to search through certain types of files.
4. Start with a fresh action.
5. Set the action type to "search".
6. Enter the search terms that you want to find.
7. Select "search for sections" from the "file sectioning" list. Leave the section search type as "regular expression".
8. In the Section Search box, enter the regular expression «<text:p[^<>]*+>. *?</text:p>». This regular expression matches a pair of <text:p> and </text:p> XML tags, and the text between them. In OpenDocument Format files, all printable text is stored between such tags. One tag holds one paragraph of text. Turn on "case sensitive search" in the file sectioning for better performance. XML tags are case sensitive.
9. Click the Preview button to run the action.

Note that in OpenDocument Format files, paragraphs with mixed formatting (bold, italics, etc.) will have extra formatting tags inside them. If you follow the above steps, PowerGREP will search the document one paragraph at the time, including the paragraph tag itself and any formatting tags inside it.

If you do a search-and-replace, it's important to make sure that the replacement text consists of a valid piece of XML. One way to do this is to use file sectioning as described above, and to make sure that your search-and-replace does not touch the XML tags (codes between sharp brackets) in the sections that are found.

11. Extract or Delete Lines Matching One or More Strings or Regexes

PowerGREP's file sectioning feature makes it trivial to extract and delete lines, or any other kind of file section. Follow these steps to extract all lines matching at least one of the search terms:

1. Select the files you want to search through in the File Selector.
2. Start with a fresh action.
3. Set the action type to "search".
4. In the "file sectioning" list, select "line by line".
5. Turn on the option "collect/replace whole sections". This makes sure lines will be extracted as a whole.
6. Enter one or more search terms.
7. Click the Preview button to run the action.

To extract the lines into new files, set the target type to "save one file for each searched file". Set "between collected text" to "Line break". Then click the Collect button.

To delete the lines from the original files instead, set the action type to "search-and-delete" in step 3 above. If you set the file sectioning type to "line by line", matching lines will be blanked out, but the number of lines in the file will remain the same. If you set it to "line by line (including line breaks)", then the lines will be completely deleted, and the number of lines in the file will shrink.

Extract or Delete Lines Matching All Search Terms

The above steps will extract or delete any line that contains at least one of the search terms. If a line must contain all search terms in order to be extracted or deleted, turn on the option "list only sections matching all items". This option becomes visible after you set the file sectioning type to "line by line".

For more complex combinations of search terms, see the example about boolean operators.

Extract or Delete Lines Matching None of The Search Terms

To extract or delete those lines that do not contain any of the search terms, turn on the "invert search results" option in the file sectioning. When using a list of search terms, make sure "list only sections matching all terms" is off.

Extract or Delete Lines Not Matching All of The Search Terms

If you turn on both "invert search results" and "list only sections matching all terms" then you will extract or delete all lines that contain none of the search terms as well as all lines that contain some but not all of the search terms.

12. How to Delete Repeated Words

Repeated words, such as “the the”, are a common error that’s easy to overlook when editing text documents. PowerGREP makes such mistakes easy to find and correct.

1. Select the files you want to search through in the File Selector.
2. Start with a fresh action.
3. Set the action type to “search-and-replace”. Leave the search type as “regular expression”.
4. In the Search box, enter the regular expression `«\b(\w+)(\s+\1\b)+»` and make sure to leave “case sensitive search” off. This regex matches a word and its repetitions. The word is stored into a capturing group. The word boundaries make sure we don’t match partial words, such as “he” in “the helmet”.
5. Enter “\1” as the replacement text. This backreference will be substituted with the contents of the first capturing group, in this case the repeated word.
6. Set the target and backup file options as you like them.
7. Click the Preview button to run the action. PowerGREP will find all repeated words, but will not actually replace them.
8. On the Results panel, see if “per file” is selected in “group search matches”. If not, select it and click the Update button.
9. Double-click on one of the files in the results to open it in PowerGREP’s editor.
10. In the editor, use Next Match and Make Replacement to delete repeated words. The editor is a full-featured text editor. You can edit the file in any way you want. PowerGREP automatically keeps track of the search matches (i.e. repeated words) while you edit.
11. Save the file in the editor, and repeat from step 9 to edit all other files.

PowerGREP’s full-featured built-in editor makes it very easy to decide for each individual search match whether to replace it. You don’t have to click Yes/No for each search match in the order that PowerGREP finds them, like most other search-and-replace tools force you to.

You can also work the other way around. In step 7, click the Replace button to delete all repeated words. In step 12, use the Revert button to undo individual search matches.

13. Add a Header and Footer to Files

With a search-and-replace action, you can just as easily insert new information into files as you can replace information. The difference is that rather than specifying a search term to be replaced, you use a regular expression that matches a position in the file. Anchor and lookaround tokens are two ways of matching a position rather than actual text with a regular expression. Another way is to simply use the backreference `\0` to reinsert the search match into the replacement text.

This example uses the anchor method. The navigation bar example uses the backreference method.

1. Select the files you want to add the header and/or footer to in the File Selector.
2. Start with a fresh action.
3. Set the action type to “search-and-replace”. Set the search type to “list of regular expressions”.
4. In the Search box, enter the regular expression `«\A»`. This regular expression matches the position at the very start of the file.
5. In the Replacement box, enter the header text you want to insert.
6. Click the green plus button to the left of the Search box to add a second step to the action.
7. Enter `«\z»` in the Search box to make the second step match at the very end of the file.
8. Enter the footer text in the Replacement box.
9. Set the target and backup file options as you like them.
10. Click the Preview button to run a test.
11. If all looks well, click the Replace button to actually add the header and footer.

14. Add Line Numbers

With a search-and-replace action, you can just as easily insert new information into files as you can replace information. The difference is that rather than specifying a search term to be replaced, you use a regular expression that matches a position in the file. Anchor and lookahead tokens are two ways of matching a position rather than actual text with a regular expression. Another way is to simply use the backreference `\0` to reinsert the search match into the replacement text.

This example uses an anchor to match the start of the line, and the `%MATCHFILEN%` placeholder to insert the line numbers.

1. Select the files you want to add line numbers to in the File Selector.
2. Start with a fresh action.
3. Set the action type to “search-and-replace”. Set the search type to “regular expression”.
4. Enter `«^»` as the search term. This regular expression matches the start of any line in the file, including blank lines.
5. Enter `“%MATCHFILEN%. ”` as the replacement text. `%MATCHFILEN%` is a placeholder for the sequence number of the match being replaced in the current file. Since the regular expression we’re using matches each line, `%MATCHFILEN%` is essentially the line number.
6. Set the target and backup file options as you like them.
7. Click the Preview button to run a test.
8. If all looks well, click the Replace button to actually add the line numbers.

Add Line Numbers to Non-Blank Lines

If you only want to add numbers to lines that aren’t blank, you can follow the same steps as above, using the regular expression `«^(?=[\t]*+\S)»`. This regular expression uses positive lookahead to check if the line the caret matched at isn’t blank. It does this by skipping leading whitespace with a possessive character class, followed by `«\S»` to check if there are any further characters on the line.

Since blank lines are not matched at all, they are not included in the `%MATCHFILEN%` count. The non-blank lines will be numbered without gaps in the numbering. The numbering will restart at 1 for each file. To give all lines a unique number throughout the file, use `%MATCHN%` instead of `%MATCHFILEN%`.

Add Numbers to Lists in a File

Making the numbering restart at one for each block of non-blank lines is also possible, if slightly more complicated. This can be useful if you have files containing several lists with one item on each line, and the lists are separated by one or more blank lines. This action effectively turns all of the lists into independently numbered lists. The first 4 steps are the same as in the first example.

5. Enter `“%MATCHSECTIONN%. ”` as the replacement text. `%MATCHSECTIONN%` is a placeholder for the sequence number of the match being replaced in the current section. In this case, a section is one list of items, as we’ll define in the following steps.
6. Select “split along delimiters” in the “file sectioning” list. We will split up the file so we can number each of the lists in the file independently.

7. In the “section search” box, enter the regular expression «`\r\n(?: [\t]*+\r\n)++`». This regular expression matches a line break, followed by one or more line breaks. Each of the following line breaks can optionally be preceded by some whitespace. Effectively, this regular expression matches one or more blank lines, including the leading and trailing line breaks. Since we’re using the “split along delimiters” sectioning type, the files will be split up along blank lines. The main action will only search through the non-blank lines.
8. Set the target and backup file options as you like them.
9. Click the Preview button to run a test.
10. If all looks well, click the Replace button to actually create the numbered lists.

15. Collect Page Numbers

PowerGREP deals with plain text files. Plain text files consist of unformatted text, so there's no real concept of a page. Still, plain text files can contain page breaks represented by ASCII character 12 decimal. Some text editors, such as EditPad Pro and PowerGREP's built-in editor, allow page breaks to be inserted by pressing Ctrl+Enter and show them as horizontal lines.

PowerGREP's built-in decoder that converts PDF files into plain text (so PowerGREP can search through them) also inserts page breaks that match the page transitions in the original PDF. You can make PowerGREP search for these page breaks to determine the page numbers. In this example we'll do this to get search results that indicate on which page each search match was found. We'll use the "file sectioning" feature to split the file into one section per page. The main search then processed the PDF one page at a time, with the section number being the page number.

1. Select the PDF files you want to search through in the File Selector.
2. Start with a fresh action.
3. Set the action type to "collect data".
4. Set "file sectioning" to "split along delimiters".
5. To use each page break as the delimiter to divide the file into sections (pages), we need to set the search term for the file sectioning to a page break. There are two ways to do this. Choose whichever way you find more comfortable.
 - o Set the "search type" to "literal text". Click on the "section search" box and then press Ctrl+Enter. A horizontal line representing the page break appears.
 - o Set the "search type" to "regular expression" and type in the regex «\x0C» into the "section search" box. This regular expression matches ASCII character 12 which is the page break character.
6. Specify your search term(s) in the main part of the action.
7. In the collect box, use the match placeholder "%SECTIONN%" as a placeholder for the page number. E.g. "%MATCH% on page %SECTIONN%" collects "found me on page 7" when the main part of the action finds "found me" in the 7th section (page).
8. Click the Search button to run the search.

You can find this action in the PowerGREP.pgl standard library as "Collect page numbers".

16. Update Copyright Years

At the start of every year, you have to update the year in the copyright statements on your web site and other published materials. If you forget this, your web site will look outdated.

There are several reasons why this seemingly trivial task can be quite tedious:

1. You probably have a lot of files to update, with copyright statements in different places. So you want to automate it.
2. You cannot just search and replace the year number, because historic dates should not be updated.
3. Different files may have a different style of copyright statement, such as a © HTML element rather than the © character.
4. You may have forgotten to update some statements in the past. You want to make sure to update those now.
5. The first year in the copyright statement will be different for different projects. This year should not be changed.

In PowerGREP, you can solve this problem easily:

1. Select the files you want to search through in the File Selector.
2. Open the PowerGREP.pgl library file included with PowerGREP. You can find it in the folder where PowerGREP is installed, c:\Program Files\JGsoft\PowerGREP4 by default.
3. Select “Update copyright statements” in the library, and click the Use Action button.
4. Set the replacement to “\1-” (backslash, one, dash) followed by the current year.
5. Set the target and backup file options as you like them.
6. Click the Preview button to run a test.
7. If all looks well, click the Replace button to actually update the copyright statements.

This was all so easy, because the regular expression we used had already been created. Writing the regular expression to take into account the problems mentioned above is the hard part.

How The Regular Expression Works

The regular expression we used is «(copyright +(©|\(c\)|©) +\d{4})(*[-,] *\d{4})*». We take care of problem 2 by not just searching for the year, but for the complete copyright statement. We solve problem 3 by having the regex search for different styles, and by putting the actual copyright statement in a backreference. Problem numbers 4 and 5 are solved by only putting the first year in the backreference that we use in the replacement.

In the replacement we used “\1” which is replaced by the text matched by the part between the first set of parenthesis in the regular expression. In our case: «copyright +(©|\(c\)|©) +\d{4}». This regular expression matches the literal text “copyright” followed by one or more spaces, followed by either the real copyright symbol ©, the textual representation (c), or the HTML character ©. The symbol must be followed by one or more spaces, and 4 digits.

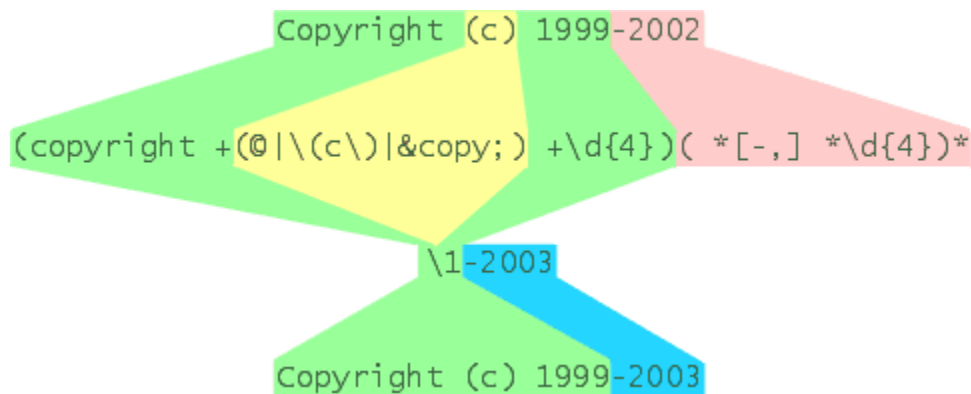
The first part of the regular expression will successfully match a copyright statement in the form of "Copyright (c) 1999".

However, some statements may have the form "Copyright (c) 1999, 2000, 2001" or "Copyright (c) 1999-2002". In either case, the first part of the regular expression will match "Copyright (c) 1999". So we need to add a second part to the regular expression to match the additional years. We will put this part outside of the first parenthesis so it will be excluded from the replacement text.

We match the additional years with: «(*[-,] *\d)*». This will match zero or more spaces, followed by a dash or a comma, followed by zero or more spaces, followed by 4 digits. All of this can appear zero or more times.

If we use "\1-2005" as the replacement, we will replace the entire copyright statement with all the years by the same copyright statement with only the first year, followed by -2005. So both statements mentioned two paragraphs earlier will be replaced by "Copyright (c) 1999-2005". We maintained the style and the first year, and updated the year even if the first copyright statement wasn't updated the past few years.

Here is a visual representation of how the original text is matched by the regular expression and turned into the final text by the replacement text with the backreference \1.



17. Padding Replacements

PowerGREP's match placeholders make it easy to pad the replacement text in a search-and-replace action, or the text to be collected in a "collect data" action. If the text to be collected is simply a regular expression match or a single capturing group, you can use the padding specifiers directly in the main action. If the replacement text is more complex, you can use PowerGREP's extra processing feature to pad it.

1. Select the files you want to process in the File Selector.
2. Start with a fresh action.
3. Set the action type to "search-and-replace" or "collect data". Leave the search type as "regular expression".
4. Enter the regular expression that matches the items you want to collect or replace. Use capturing groups to extract specific parts of each record.

For the replacement text or text to be collected, you have two options: the whole regex match or a single capturing group, or something more complex. For the whole match or a single group, you can use a match placeholder with padding specifier directly:

5. As the replacement text or text to be collected, enter "%MATCH:6L%" or "%GROUP1:6L%". Use %MATCH% for the whole match, or %GROUP1% for the capturing group with index 1, %GROUP2% for group #2, etc. :6L is the padding specifier. This one pads up to 6 spaces at the left. Use R instead of L to pad at the right, C for center, Z for zero-padding at the left, and A for a-padding at the left. Enter any number instead of 6 to set the length the final replacement should have.
6. Set the target and backup options as you want.
7. Click the Preview button to check the results.

If the replacement text is a combination of multiple capturing groups and/or literal text, you'll need to use extra processing:

5. As the replacement text or text to be collected, without regard to padding.
6. Turn on the "extra processing" option.
7. Leave the extra processing search type as "regular expression", and turn on the option "dot matches newlines".
8. In the "extra processing search" box, enter the regular expression «.++». This regular expression matches the entire replacement text from step 5.
9. As the "extra processing replacement", enter %MATCH:12L% to pad the replacement up to 12 spaces at the left. Use R instead of L to pad at the right, C for center, Z for zero-padding at the left, and A for a-padding at the left. Enter any number instead of 12 to set the length the final replacement should have.
10. Set the target and backup options as you want.
11. Click the Preview button to check the results.

18. Capitalize The First Letter of Each Word

PowerGREP’s match placeholders make it easy to change the case of the replacement text in a search-and-replace action, or the text to be collected in a “collect data” action. This example shows how you can capitalize the first character in each word. For a more generic example, see “padding replacements”.

1. Select the files you want to search through in the File Selector.
2. Set the action type to “search-and-replace”. Leave the search type as “regular expression”.
3. Enter the regular expression «\w++». This regular expression matches a single word.
4. Enter “%MATCH:F%” as the replacement text. This match placeholder inserts the entire regular expression match, with the first character converted to upper case and the rest to lower case.
5. Set the target and backup file options as you like them.
6. Click the Preview button to run a test.
7. If all looks well, click the Replace button to actually capitalize the words.

Capitalizing Words in Strings

In practice, you’ll often want to capitalize only certain words in the file rather than all words. The example below shows how to capitalize words inside double-quoted strings only. PowerGREP’s file sectioning feature makes this easy. The first five steps are the same as in the previous action.

6. Select “search for sections” in the “file sectioning” drop-down list.
7. Enter the regular expression «"[^"\r\n]++» or one of the other regexes for matching strings in the “section search” box. This regular expression matches a double-quoted string that does not span across lines. Double quotes cannot appear inside the string.
8. Click the Preview button to run a test.
9. If all looks well, click the Replace button to actually capitalize the words.

That’s all there’s to it. The file sectioning feature simply restricts the main action to the parts of the file matched by the file sectioning regular expression. There’s no need to modify the main action.

19. Add Proper HTML <TITLE> Tags

Many web authors are sloppy at adding proper <TITLE> tags to their HTML files. They are easy to forget because they are not clearly visible when viewing a website. However, <TITLE> tags are important because they're used as the default name for bookmarks/favorites. Most search engines will use the titles to list your pages in the search results.

Assuming you have been more careful with adding the title to the HTML body, you can easily fix this problem with PowerGREP. Usually, <H1> tags are used to add titles to the body. We will use <H1> tags in the example below, but you can easily adapt it to whatever tags you have been using. What we'll do is tell PowerGREP to find the <H1> tag in each file and capture its contents. Then we use the captured text to replace the <TITLE> tag.

The “filter files” feature on the Action panel is what we'll use to capture the <H1> tag into a named capturing group. Then we can set the main action to search for the <TITLE> tag and to replace it with the contents of the named capturing group. This relies on PowerGREP's special ability to carry over text matched by named capturing groups from one part of the action to the next.

1. Select the HTML files you want to update in the File Selector.
2. Start with a fresh action.
3. Set the action type to “search-and-replace”.
4. Set “filter files” to “require all search terms to match”. Leave the search type as “regular expression”.
5. Enter the regular expression «<h1>(?'h1' .*)</h1>» in the Search box in the “filter files” part of the action. This regex matches the opening and closing <h1> tags and any text between them. The text between them is captured into the named capturing group “h1”.
6. Enter the regular expression «<title>.*?</title>» in the Search box in the main part of the action. This regex matches the opening and closing <title> tags and any text between them. This regex does not capture anything.
7. Enter the replacement text “<title>\${h1}</title>” to insert a new pair of title tags with the text matched by the named capturing group “h1” between them.
8. Set the target and backup file options as you like them.
9. Click the Preview button to run a test.
10. If all looks well, click the Replace button to actually update the TITLE tags.

Should a file not have an <H1> tag, then it is filtered out and no changes are made to it. If a file has more than one <H1> tag, then only the first tag is used. Once all the regular expressions in “filter files” have found a match, PowerGREP considers the file to meet the filtering requirement. It won't look for any further matches for the filtering regex.

If a file does not have an <TITLE> tag, the search-and-replace won't replace anything. If a file has more than one <TITLE> tag, then all of them are replaced with the contents of the first <H1> tag in the file.

This action is available in the PowerGREP.pgl library as “Update HTML title tags”.

How to Insert Missing <TITLE> Tags

If some of your HTML files do not have TITLE tags at all, but they do all have <HEAD> tags, you can use the following regular expression «<head>(?(.?)<title>.*?</title>)?» for the search-and-replace.

This regex matches the `<head>` tag optionally followed by the group `«(. *?)<title>.*?</title>»`. This group starts with `«(. *?)»` to skip over any number of characters and capture those into capturing group number one. The star is made lazy so this group matches as few characters as possible, expanding only as needed to allow `«<title>.*?</title>»` to match the title tag. If there is a title tag, then the first capturing group matches the text between the head and title tags. If there's no title tag, then `«(. *?)»` expands all the way to the end of the file before giving up (assuming we turned on "dot matches newlines"). Since the question mark and the end of the regex makes the group after the head tag optional, the regex matches only `„<head>”` in that case.

The replacement text becomes `"<head>\1<title>${h1}</title>"`. In addition to inserting the new title tag and the named capturing group, this replacement text also re-inserts the `<head>` tag that we matched and the text between the head and title tags that we may have matched. If there was no title tag in the file, then the first capturing group did not participate in the match, and `"\1"` inserts nothing.

You can find this action in the PowerGREP.pgl standard library as "Update or insert HTML title tags".

20. Rename Files Based on HTML Title Tags

The “rename files” action type enables you to rename files by searching and replacing through their file names or paths. With the “filter files” feature you can first run a search through the contents of each file and then use (part of) the search match in the search-and-replace through the file’s name. This way you can extract text from the file’s contents and insert it into the file’s name.

As an example, we’ll rename a bunch of HTML files. The new name of each file will be whatever is specified in the <TITLE> tag inside the file. If a file does not have a <TITLE> tag, we use the contents of the <H1> tag instead. If a file has neither tag, it is not renamed.

1. Select the HTML files you want to rename in the File Selector.
2. Start with a fresh action.
3. Set the action type to “rename files”.
4. Leave “what to rename” set to “file name only”. Our search-and-replace should only change the file’s name.
5. Set “filter files” to “require all search terms to match”. Leave the search type as “regular expression”.
6. Enter the regular expression «<(TITLE|H1) [^<]*>(?'title' [^<]+)</\1>» in the Search box in the “filter files” part of the action. This regex matches the opening and closing <TITLE> or <H1> tags (whichever pair comes first) and any text between them. The text between them is captured into the named capturing group “title”.
7. Enter the regular expression «^.*\.» in the Search box in the main part of the action. This regex matches everything up to and including the last dot in the file’s name.
8. Enter the replacement text “\${title}.” to replace the file’s name with the contents of the tag matched by the “filter files” regex. The replacement also puts back the dot that delimits the file’s extension. The extension is not matched by the regex and thus remains unchanged.
9. Tick the extra processing checkbox. An additional set of controls for entering search terms appears.
10. Use «[\\/:*?"<>|]» as the regular expression for extra processing. This regex matches any character that is not allowed in file names by the Microsoft Windows operating system.
11. Leave the extra processing replacement blank so invalid characters are deleted.
12. Set the backup file options as you like them.
13. Click the Preview button to run a test.
14. If all looks well, click the Rename button to actually rename the files.

Should a file not have a <TITLE> or <H1> tag, then it is filtered out and not renamed. If a file has both a <TITLE> and <H1> tag, or multiple occurrences of the same tag, then only the first tag is used. Once all the regular expressions in “filter files” have found a match, PowerGREP considers the file to meet the filtering requirement. It won’t look for any further matches for the filtering regex.

This action is available in the PowerGREP.pgl library as “Rename files based on HTML title tags”.

21. Replace HTML Tags

When editing a web site, you may want to update some HTML tags to give the site a more consistent look. Let's say some pages were created by other people, and they used slightly different text and background colors. With PowerGREP, you can easily do a search and replace to replace any <body> tag with the one you want.

1. Select the files you want to search through in the File Selector.
2. Start with a fresh action.
3. Set the action type to "search-and-replace". Leave the search type as "regular expression".
4. In the search box, enter the regular expression «<BODY[^>]*>» and make sure to leave "case sensitive search" off. This regular expression will match <BODY, followed by zero or more characters that aren't a closing sharp bracket, followed by a single closing sharp bracket.
5. Type the tag you want to replace all body tags with in the Replacement box. E.g.: "<BODY BGCOLOR=white TEXT=black>"
6. Set the target and backup file options as you like them.
7. Click the Preview button to run a test.
8. If all looks well, click the Replace button to actually replace the tags.

Maintaining your web site is much easier with the help of PowerGREP. Most (visual) HTML editors cannot do a search and replace across all files your web site consists of. Most text editors can only search and replace literal strings, which makes it tedious to replace several styles of tags with the same tag.

You can find this action in the PowerGREP.pgl standard library as "Replace HTML tags".

22. Replace HTML Attributes

This was one of the most complicated examples in the documentation that shipped with PowerGREP 1.0. Like most grep tools, PowerGREP 1.0 was not able to search through only certain sections of files. Now that PowerGREP has this ability, replacing HTML attributes is very straightforward. Makes you wonder why PowerGREP is the only Windows grep tool to support file sectioning.

When editing a web site, you may want to update some HTML tags to give the site a more consistent look. Suppose you have some tables on your web site with different background colors, and you want to give all of them the same color. However, you only want to update the “bgcolor” attribute of the tables. All the other attributes should remain unchanged.

1. Select the files you want to search through in the File Selector.
2. Start with a fresh action.
3. Set the action type to “search-and-replace”. Leave the search type as “regular expression”.
4. Select “search for sections” from the File Sectioning drop-down list. Leave the search type as “regular expression”.
5. In the Section Search box, enter the regular expression «<table[^>]*» and make sure to leave “case sensitive search” off.
6. In the search box of the main part of the action, enter the regular expression «bgcolor=([_a-z0-9]+|' [^\\']* |" [^\\"]*")» and make sure to leave “case sensitive search” off. This regular expression matches any bgcolor attribute with an unquoted value, or a single-quoted value, or a double-quoted value.
7. Enter “bgcolor=blue” in the Replacement box. Each bgcolor attribute will be replaced with whatever you enter in the Replacement box.
8. Set the target and backup file options as you like them.
9. Click the Preview button to run a test.
10. If all looks well, click the Replace button to actually replace “bgcolor” attributes in “table” tags.

You can find this action in the PowerGREP.pgl standard library as “Replace HTML attributes”.

If you’re curious, with a basic grep tool that can only search-and-replace using one regular expression, this is the search pattern to use:

```
(<table([\s\r\n]+[a-z]+(=[_a-z0-9]+|' [^\\']* |" [^\\"]*" )?)*)([\s\r\n]+bgcolor=( [_a-z0-9]+|' [^\\']* |" [^\\"]*" )?(([\s\r\n]+[a-z]+(=[_a-z0-9]+|' [^\\']* |" [^\\"]*" )?)*[\s\r\n]*>)
```

The replacement text would be `\1 bgcolor=blue \7`

You can see the same regular expression we used to match the bgcolor attribute in the middle of this behemoth regex. All the other stuff is for matching the table tag around the attribute. It works, but PowerGREP’s sectioning abilities do make life a lot easier.

23. Put Anchors Around URLs That Are Not Already Inside a Tag or Anchor

Suppose you have an HTML file that has URLs in its body text that are not clickable. You want to make them clickable by placing the URLs inside anchor tags. But like any other HTML file, your file also has URLs as part of anchors (links), images, and other tags. Those URLs should be left alone. You also want to ignore URLs that have already been placed inside anchor tags.

1. Select the files you want to search through in the File Selector.
2. Start with a fresh action.
3. Set the action type to “search-and-replace”. Leave the search type as “regular expression”.
4. Select “split along delimiters” from the File Sectioning drop-down list.
5. Set the “section search type” to “list of regular expressions”.
6. Add «`<a\b[^<]*>.*?`» as the first file sectioning regular expression. It matches any `<a>` tag and its contents.
7. Add «`[^<]+>`» as the second regex. This regex matches any opening or closing HTML tag. This regex assumes all `<` characters in your HTML file that aren’t part of tags are properly escaped as `<`;
8. Make sure “non-overlapping search” is turned on. The file sectioning should make one pass of the file using both regular expressions.
9. In the search box of the main part of the action, enter the regular expression «`https?://\S+`» which is a quick way of matching any web URL.
10. Enter «`\0`» in the Replacement box. This replaces each URL with itself wrapped inside an anchor using itself as the destination.
11. Set the target and backup file options as you like them.
12. Click the Preview button to run a test.
13. If all looks well, click the Replace button to actually replace the URLs.

When PowerGREP executes this action, it first uses the file sectioning regex to match all the anchor tags with their contents, and all other HTML tags without contents. Because we put the anchor tag regex first in the list, it takes precedence over the HTML tag regex. At a position where both regexes can match, only the first one will. With “non-overlapping search” turned on, searching for the list of regular expressions «one» and «two» (in that order) is exactly the same as searching for the single regex «one|two». A list of multiple short regexes is easier to manage than a long regex with many alternatives. But there’s no functional difference.

Because “file sectioning” is set to “split along delimiters”, PowerGREP treats the matches of the file sectioning regexes as delimiters that chop the file into pieces. The action’s search-and-replace separately processes each bit of text between two delimiters (and before the first and after the last delimiter). In this case, the search-and-replace works on each bit of text between two HTML tags, between two anchor tags (with contents), or between an anchor tag and another HTML tag. Essentially, the search-and-replace skips over all anchor tags (with contents) and all HTML tags.

You can find this action in the PowerGREP.pgl standard library as “Put anchors around URLs that are not already inside a tag”.

24. Replace in File Names and Contents

You can search and replace through file names with the “rename files” action type. You can search and replace through the contents of files with the “search-and-replace” action type. PowerGREP does not have an action type that does both at the same time. Fortunately, we can use the Sequence panel to execute two actions as one operation.

Let’s say you have a set of files that need to be updated annually. The files that need updating have the year in their file name. You have to create a copy of those files with the new year in the file name. You also have to replace the year in the contents of those files.

1. Select the files you want to update in the File Selector.
2. Start with a fresh action.
3. Set the action type to “rename files”.
4. Set “what to rename” to “full path” if the folders containing the files also have year numbers in them, and you want to create new folders for the new year.
5. In the Search box, enter the old year (e.g. 2010).
6. In the Replace box, enter the new year (e.g. 2011).
7. Set “target file creation” to “copy files”.
8. Set the backup file options as you like them.
9. Click the New Step button on the Sequence panel to add the contents of the Action panel as the first step in the sequence.
10. On the Action panel, change the action type to “search-and-replace”.
11. Set “target file creation” to “modify original files”. In this case, the original files will be the files that were copied by the first step in the sequence.
12. Click the New Step button on the Sequence panel to add the contents of the Action panel as the second step in the sequence.
13. With the second step still selected on the Sequence panel, select “target files from other step” in the “file selection” drop down list.
14. Select step 1 in the “step” drop-down list. The second step is now configured to process the target files created by the first step.
15. Click the Execute button on the Sequence panel to execute both steps. The first step copies the files, changing 2010 in their paths into 2011. When that’s done, the second step searches through the contents of the copied files, replacing 2010 with 2011.

This sequence is available in the PowerGREP.pgl library as “Replace in file names and contents”.

25. Replacing Named XML Entities

PowerGREP's ability to search and replace using a delimited list of search terms makes it very easy to search-and-replace all reserved XML character with their named XML entities. Simply set the search type to "delimited literal text", set the extra item delimiter to a line break, the extra pair delimiter to an equals sign, and paste in the following search text:

```
&=&
<=&lt;
>=&gt;
'=&apos;
"=&quot;
```

When extracting text from an XML file, you can easily turn things around to replace the named XML entities with the characters they represent:

```
&amp;=&
&lt;=<
&gt;=>
&apos;='
&quot;="
```

Collect XML Data with Entities Replaced

PowerGREP's extra processing feature makes it very straightforward to collect text from an XML file with all entities replaced with their corresponding characters.

1. Select the files you want to search through in the File Selector.
2. Start with a fresh action.
3. Set the action type to "collect data". Leave the search type as "regular expression".
4. In the search box, enter the regular expression that matches the XML data you want to extract. E.g. «<tag [^>]+> ([^<>]+) </tag>» matches any text (but no XML) between <tag> and </tag>.
5. Type \1 in the Collect box. This will collect just the text between the tags matched by our regular expression.
6. Tick the extra processing checkbox. An additional set of controls for entering search terms appears.
7. Set the extra processing search type to "delimited literal text".
8. Leave the "extra item delimiter" field set to "Line break". Type a single equals sign in the "extra pair delimiter" field.
9. Copy the second list of search-and-replace pairs in the first section of this help topic. Paste it into the "extra processing search" box in PowerGREP.
10. Set the target and backup file options as you like them.
11. Click the Preview button to run a test.
12. If all looks well, click the Collect button to actually collect the text.

PowerGREP will now collect all the text between <tag> and </tag> tags in your XML files. If any of the text contains named entities, they will be replaced before the text is collected. The replacements are only made to the text being collected. They're not made to the original XML files.

You can find this action in the PowerGREP.pgl standard library as "XML: Collect search matches with named entities replaced".

The example “replace reserved characters in XML files” in the PowerGREP library shows how you might use the “extra processing” feature for doing the opposite: replacing reserved characters with entities.

26. Fix Invalid Characters in XML

Sometimes, XML files generated by poorly written software or by careless programmers will contain lone characters like < and &. These will cause the XML file to be rejected by XML parsers. They must be replaced with the entities < and &. Using PowerGREP, we can easily fix this with a search-and-replace using two regular expressions.

1. Select the files you want to search through in the File Selector.
2. Start with a fresh action.
3. Set the action type to “search-and-replace”. Set the search type to “list of regular expressions”.
4. In the Search box, enter the regular expression «<(?![_: a-z] [-._: a-z0-9]*\b[<>]*)» and make sure to leave “case sensitive search” off. This regex matches any < symbol that is not followed by what looks like a valid XML tag. I’m using «[_: a-z] [-._: a-z0-9]*\b» to check for an XML tag name, and «[<>]*» to skip over any attributes. This regex isn’t 100% exact, but it’s easy to deal with. The example in the PowerGREP Library does include an exact regex.
5. In the Replacement box, type “<”.
6. Click the button with the green plus symbol to the left of the Search box to prepare for another search-and-replace pair.
7. In the Search box, enter the regular expression «&(?! (? : [a-z]+ |# [0-9]+ |#x [0-9a-f]+) ;)». This regex matches any ampersand that is not followed by an entity name or character code.
8. In the Replacement box, type “&”.
9. Set the target and backup file options as you like them.
10. Click the Preview button to run a test.
11. If all looks well, click the Replace button to actually replace the tags.

This action will replace all invalid < and & characters with their respective entities. This action is a solution for XML files generated by a computer program that inserted arbitrary text into an XML structure without replacing the < and & characters in that text first.

If the computer program inserts the invalid XML only between certain XML elements, you can leverage PowerGREP’s “file sectioning” feature to use simpler regular expressions. The example below assumes a computer-generated XML file that is valid, except that the program inserted some SQL code between <sql>...</sql> tags without replacing the “greater than”, “less than”, and “and” symbols in the SQL with XML entities.

1. Select the files you want to search through in the File Selector.
2. Start with a fresh action.
3. Set the action type to “search-and-replace”.
4. Set “file sectioning” to “search and collect sections”. Leave the section search type as “regular expression”.
5. In the “section search” box, enter the regular expression «<sql>(.*?)</sql>» to match the <sql> element and its contents.
6. Turn on “dot matches newlines” to allow the section to span across lines.
7. Enter “\1” into the “section collect” box. This restricts the section to the contents of the <sql> element without the element’s enclosing tags. This is important because we only want to replace the reserved characters in the element’s contents.
8. Set the search type of the main part of the action to “delimited literal text”.
9. Leave the “search term delimiter” field set to “Line break”. Type a single equals sign in the “search pair delimiter” field.
10. Paste these three lines into the search box:

11. `<=<`;
12. `>=>`;
 `&=&`;

13. Set the target and backup file options as you like them.
14. Click the Preview button to run a test.
15. If all looks well, click the Replace button to actually replace the tags.

You can find this action in the PowerGREP.pgl standard library as “Replace reserved characters in XML files”.

27. Search Through or Skip Source Code Comments and Strings

When searching through or modifying source code files, you'll often want to restrict the search to comments and/or strings, or search through comments and/or strings exclusively. E.g. if you discover you've been misspelling "referrer" as "referer" throughout your project, you'd probably want to fix the mistake in comments and strings, but leave the actual source code untouched. Modifying the source code might break ties to other modules, a hassle not worth correcting a spelling mistake. (As a bit of trivia: the Apache web server stores the referring URL in a variable HTTP_REFERER for exactly this reason.)

PowerGREP makes this easy with the "file sectioning" part of the action definition. The examples below only describe the file sectioning settings. Enter the actual search terms in the as usual.

Search Through Comments and Strings Only

1. Select files and set the main part of the action as usual.
2. Select "search for sections" from the "file sectioning" list.
3. Set the section search type to "list of regular expressions". Make sure "non-overlapping search" is on.
4. Add one regular expression to the list for each kind of string and comment the programming language you're working with supports. E.g. for C or Java, use `«//.*»` for single-line comments, `«(?s)/*.*?\/»` for multi-line comments, and `«"[^"\\\r\n]*(?:\\.[^"\\\r\n]*)*»` for strings. The `«(?s)»` in the second regex turns on "dot matches newline" for that regular expression only. Make sure the checkbox is *not* checked.

Don't Search Through Either Comments or Strings

Searching through source code only, skipping comments and strings, is just as easy. Instead of selecting "search for sections" in step 2 above, select "split along delimiters" instead.

"Split along delimiters" means that PowerGREP will treat comments and strings as delimiters. PowerGREP will make the main action search through everything between comments and strings, skipping the comments and strings themselves.

Search Through Comments Only, or Strings Only

You might be tempted to clear the checkboxes in front of the regular expressions in the file sectioning that match the parts of the file you don't want to search through. But that won't have the effect you intended.

Unticking a checkbox disables that regular expression completely. This is be useful when you're testing the effect of different regular expressions while designing a PowerGREP action. That is not what you want in this situation. E.g. if you disable the regexes for matching comments, the string regex will match strings in commented-out code.

To skip certain sections, select "search and collect sections" from the "file sectioning" list. A new Section Collect box will appear. In this box, enter "\0" for each sectioning step that the main action should search through. Leave it blank for sections that should be skipped.

\0 is a backreference to the entire regular expression match. When using your own regular expressions to section files, you can also use backreferences to capturing groups in the regular expression. Then PowerGREP will restrict the main part of the action to the part of the file matched by that capturing group.

28. Convert Windows to UNIX Paths

PowerGREP's "extra processing" feature makes it very easy to search through text files and replace file references in those files from Windows paths into UNIX paths. The example replaces all references to files under `c:\My Documents\` into `/home/me/`, converting backslashes into forward slashes and spaces into underscores.

1. Select the files you want to search through in the File Selector.
2. Start with a fresh action.
3. Set the action type to "search-and-replace". Leave the search type as "regular expression".
4. In the Search box, enter the regular expression `«c:\\My Documents ([^\\t\\r\\n<>|/:"]* [^\\s<>|/:"]»` and make sure to leave "case sensitive search" off. The regex matches a Windows path under `c:\My Documents`. The second character class makes sure that a space after the path is not matched as part of the path.
5. In the Replace box, enter `"/home/me\1"`
6. Tick the extra processing checkbox. An additional set of controls for entering search terms appears.
7. Set the extra processing search type to "delimited literal text".
8. Enter a single semicolon in the "extra item delimiter" field, and a single equals sign in the "extra pair delimiter" field.
9. In the "extra processing search" box, enter `"\=/; =_"` to substitute backslashes with forward slashes, and spaces with underscores.
10. Click the Preview button to run a test.
11. If all looks well, click the Replace button to actually replace the paths.

Technically, this action consists of two search-and-replace operations. The one you define first is the main action. It searches through the files you marked in the File Selector. The "extra processing" search-and-replace is applied each time the main action finds a match. Extra processing does not search through any files, but makes replacements in the replacement text of the main action, just before the main action substitutes the search match in the file.

An example will make this clear. If you apply the above action to a single file containing the text "The path `c:\My Documents\Test Files\Path Test.txt` will be converted", PowerGREP does the following:

1. The regular expression of the main action matches `„c:\My Documents\Test Files\Path Test.txt”`
2. The backreference in the replacement text of the main action is expanded. The replacement becomes `"/home/me\Test Files\Path Test.txt”`
3. The extra processing part of the action is invoked on the replacement. It makes 4 substitutions, replacing two spaces with underscores, and two backslashes with forward slashes. The new replacement text for the main action becomes `"/home/me/Test_Files/Path_Test.txt”`
4. The main action deletes the search match from the file, and substitutes it with the new replacement text.
5. The whole process is repeated from step 1 for all remaining search matches in the file. There are none in this example.

The end result is "The path `/home/me/Test_Files/Path_Test.txt` will be converted"

You can find this action in the PowerGREP.pgl standard library as "Convert Windows paths into UNIX paths".

29. Extract Data into a CSV File or Spreadsheet

With PowerGREP, you can easily extract any sort of information from large numbers of documents, archives or spreadsheets, and save the collected information into a comma-delimited text files or CSV files. Here are the basic steps:

1. Select the files you want to extract information from in the File Selector.
2. Start with a fresh action.
3. Set the action type to “collect data”. Leave the search type as “regular expression”.
4. Enter the regular expression that matches a data record. Use capturing groups to extract specific parts of each record.
5. As the text to be collected, enter a comma-delimited list of backreferences to those capturing groups.
6. Set the target type to “save results into a single file” if you want to create one CSV file that holds all the records. Specify the name of the file as the target location. Or, set the target type to “save one file for each searched file” to create one CSV file for each original file. In that case, you may want to set the target destination type to “path placeholders”. Enter `c:\Output\%FILENAMENOEXT%.csv` or `c:\Output\%FOLDER\FILENAMENOEXT%.csv` as the target location. The former placeholder will create one CSV file in the folder `c:\Output` for each source file with the same name as the source, but a `.csv` extension. The latter will also recreate the folder structure under `c:\Output`.
7. Leave “between collected text” set to “Line break”. PowerGREP will insert a line break between each collected match. PowerGREP will *not* insert it before the first match or after the last match.
8. Click the Collect button to create the CSV files.

Extracting a List of Delivery Addresses

Suppose you have a large number of orders stored in text documents, and you want to make a list of the delivery addresses. In each file, the delivery address has the following layout:

```
Deliver to:
Joe N. Doe
Street address (one or two lines)
City, ST, 12345-6789
```

In the CSV file, you want to have the following fields: `name,address 1,address 2,city,state,zip`

You can easily achieve this following the steps above. First, we need to create a regular expression that matches a delivery address, which is quite straightforward. We match "Deliver to:" first. Then we capture one line of text with `«(.*)\r\n»` which is the name. Then one or two lines with `«(.*)\r\n(?: (.*)\r\n)?»` which are the address. Finally, we match one line that ends with a state code `«[A-Z]{2}»` and ZIP code `«[0-9]{5}(?:-[0-9]{4})»`. Using `«[,]+»` we allow commas and/or spaces as delimiters in the last line. The complete regular expression becomes: `«Deliver to:\r\n(.*)\r\n(.*)\r\n(?: (.*)\r\n)?(.*)[,]+([A-Z]{2})[,]+([0-9]{5}(?:-[0-9]{4}))»`.

The `(?:group)` parts in the regular expression are non-capturing groups. Those simply group items to repeat them together. The `(group)` parts are capturing groups. They're essential in allowing us to insert part of the regular expression match into the text to be collected. In this case, we simply reference each group once. As the text to be collected, enter: `“\1,\2,\3,\4,\5,\6”`. If a capturing group did not participate in the match,

it is substituted with nothing. E.g. in a delivery address with only one line for the street address, \4 will remain blank.

Set the target type to save results into a single file to get one CSV file with all delivery addresses. You can then open the CSV file in a spreadsheet program or other application.

You can find this action in the PowerGREP.pgl standard library as "CSV: Extract data into a CSV file or spreadsheet".

30. Padding and Unpadding CSV Files

When viewing comma-delimited text files or CSV files in a text editor, the columns won't align unless the fields were padded with spaces to make their widths equal. Other applications may treat any whitespace as significant, requiring an unpadded CSV file. With PowerGREP, you can easily pad and unpad CSV files.

Padding CSV Files

To pad CSV files, we'll search using a regular expression that matches one CSV field. As the replacement, we'll re-insert the regex match padded with spaces to a specific length. Padding is easy in PowerGREP with match placeholders.

1. Select the files you want to pad in the File Selector.
2. Start with a fresh action.
3. Set the action type to "search-and-replace". Leave the search type as "regular expression".
4. Enter the regular expression «`[\t]**+("[^"\r\n]**+"[\t]**+|^,\r\n]**+)`». This regular expression matches a single field in a CSV file. The field can be enclosed in double quotes, and extra whitespace is allowed before and after the quotes.
5. Enter "`%GROUP1:40L%`" as the replacement text. This match placeholder inserts the text matched by the first (and only) capturing group in the regular expression, with as many spaces added at the left side to make it at least 40 character long. Fields that are longer to begin with will not be truncated. We're using the capturing group in the regular expression to discard any padding that may already be present in the file, so everything gets padded to 40 characters.
6. Set the target and backup file options as you like them.
7. Click the Preview button to run a test.
8. If all looks well, click the Replace button to actually pad the CSV files.

You can find this action in the PowerGREP.pgl standard library as "CSV: Pad fields".

Unpadding CSV Files

To remove the padding from the CSV file, you can use the exact same action. Only the replacement text will be different. The regular expression uses a capturing group to store the actual text of the CSV field, separate from the whitespace at the start of the field that the overall regular expression also matches. So to remove the padding, all we need to do is to replace the overall regex match with the text matched by the capturing group. You can do this with the placeholder "`%GROUP1%`" or simply the backreference "`$1`".

You can find this action in the PowerGREP.pgl standard library as "CSV: Unpad fields".

31. Collect a Numbered List

Using a “collect data” action with match placeholders makes it easy to create all kinds of numbered lists. You could create a simple numbered list of all search matches as follows:

1. Select the files you want to list matches from in the File Selector.
2. Start with a fresh action.
3. Set the action type to “collect data”. Leave the search type as “regular expression”.
4. Enter the regular expression that matches the items you want to collect. Use capturing groups to extract specific parts of each record.
5. As the text to be collected, enter “%MATCHN%. \0”. %MATCHN% is a placeholder for the number of the match. \0 is a backreference to the entire regex match.
6. Set the target type to “save results into a single file” to output all search matches as one long list in a single file. If you choose to save one file for each searched file, you’ll probably want to use %MATCHFILEN% instead of %MATCHN% in the text to be collected, so the numbering starts from 1 in each file.
7. Leave “between collected text” set to “Line break”. PowerGREP will insert a line break between each collected match. PowerGREP will *not* insert it before the first match or after the last match.
8. Click the Collect button to create the numbered list.

You can find this action in the PowerGREP.pgl standard library as “Collect a numbered list”.

32. Collect a List of Header and Item Pairs

This example illustrates how you can use file sectioning to extract items from sections. It also shows how named capturing groups carry over regex matches from the file sectioning to the main part of the action. This makes it easy to collect both part of the section (e.g. its header), and part of the item, for each item found in each section.

Windows applications often store their settings in .ini files. Such files consist of one or more headers, with one or more name and value pairs.

```
[Header1]
Name1=Value1
Name2=Value2
[Header2]
Name3=Value3
Name4=Value4
Name5=Value5
; etc...
```

With PowerGREP, you can easily extract a list of header and item pairs from such a list. E.g. let's produce the following list from the above:

```
Header1/Name1
Header1/Name2
Header2/Name3
Header2/Name4
Header2/Name5
```

To do this, we need two regular expressions. One to get the headers, and another to get the items for each header. This impossible with most grep tools, since they only allow you to use one regular expression. PowerGREP's file sectioning feature makes this task very straightforward.

You can find this action in the PowerGREP.pgl library as "Collect header/item pairs from .ini files".

1. Select the files you want to search through in the File Selector.
2. Start with a fresh action.
3. Set the action type to "collect data".
4. Select "search for sections" from the "file sectioning" list. Leave the section search type as "regular expression".
5. In the Section Search box, enter the regular expression «`^\s*\[(? 'header' [^\r\n]+)](?:\r\n\s*+[^[].*+)+`» and make sure to leave "dot matches newlines" off. This regex matches a header with «`^\s*\[(? 'header' [^\r\n]+)`» and everything that follows it up to the next header with «`(?:\r\n\s*+[^[].*+)`». It contains one named capturing group 'header'.
6. In the Search box in the main part of the action, enter the regular expression «`^([^=;\r\n]+)=.*$`» and make sure to leave "dot matches newlines" off. This regex matches a single name=value pair, and captures the name into the first backreference.
7. In the Collect box, enter «`${header}/\1`» to collect the name of the header (named capturing group carried over from the file sectioning) and the name of the value (first backreference), delimited by a forward slash.
8. Click the Preview button to see the results.

When PowerGREP executes this action, the following happens for each file:

1. The sectioning regex matches a section in the .ini file, e.g. „[Header1]\r\nName1=Value1\r\nName2=Value2”. The section’s header „Header1” is stored in the named group “header”.
2. The main action now searches through this section, and matches a name=value pair, e.g. „Name1=Value1”
3. The main action substitutes backreferences in the text to be collected for this search match, e.g. “Header1/Name1”. The result is added to the results.
4. The main action repeats steps 2 and 3 until all name=value pairs in the current section have been found.
5. PowerGREP repeats steps 1 through 4 for all sections in the .ini file.

You can easily adapt the techniques shown in this example for your own purposes.

1. Create a regular expression that matches all sections in the file you’re interested in.
2. Add named capturing groups to the regex for each part of the section (headers, footers, etc.) you want to collect for all items.
3. Create a second regular expression that matches each item in those sections. This regular expression will only “see” one section at a time. You don’t need to worry about this regex matching any part of the file outside the sections matched by the first regex.
4. Add named or numbered capturing groups to the second regex for each part of the item you want to collect.
5. Compose the text to be collected using backreferences to the groups you added in steps 2 and 4.

33. Collect Paragraphs (Split along Blank Lines)

This example illustrates how you can use file sectioning to process files paragraph by paragraph, where a paragraph is a block of lines delimited from other paragraphs by one or more blank lines.

You can find this action in the PowerGREP.pgl library as "Collect paragraphs (split along blank lines)".

1. Select the files you want to search through in the File Selector.
2. Start with a fresh action.
3. Select "split along delimiters" from the "file sectioning" list. Leave the section search type as "regular expression".
4. In the Section Search box, enter the regular expression `«(?:\r\n){2,}+»`. This regular expression matches two or more consecutive line breaks, or one or more consecutive blank lines. If lines containing only whitespace should be considered blank, use the regular expression `«\r\n(?:[\t]+)\r\n++»`.
5. Turn on the "collect whole sections" checkbox.
6. Set the action type to "search". Since we turned on the "collect whole sections" checkbox, there's no need to enter a text to be collected, so we can use the "search" action type instead of "collect data".
7. Enter the search term you want to find in the paragraphs.
8. Click the Preview button to see the results.

If you want to treat lines that consist of nothing but spaces and tabs as blank lines, use the regular expression `«\r\n(?:[\t]+)\r\n++»` to find your sections. This regex also matches two or more line breaks. The difference is that it allows whitespace between each pair of line breaks. This makes it match one or more consecutive lines that do not contain anything except whitespace. You can find this action as "Collect paragraphs (split along whitespace-only lines)" in the library.

34. Apply an Extra Search-And-Replace to Target Files

This example shows how you can use the Sequence panel in PowerGREP to run an extra search-and-replace on each target file produced by a PowerGREP action. This is different from the “extra processing” part of a PowerGREP action. This is different from the “extra processing” feature on the Action panel, which performs an extra search-and-replace on each replacement text or each text to be collected, handling each bit of text separately. The Sequence panel allows you to run a whole new action on the target files, reprocessing each file as a whole.

Suppose you want to condense consecutive spaces into single spaces in your target files when running a “collect data” action. If you used the “extra processing” feature to search for « {2,} » and replace with a single space, you’d condense consecutive spaces within each search match that is collected into the file. But if one search match ends with a single space and the next match starts with a single space, the target file will still end up with two consecutive spaces. The reason is that the “extra processing” processes each search match separately. It sees a single space at the end of the first match, and a single space at the start of the second match, neither of which are replaced because the regex doesn’t match them. To condense all consecutive spaces, even those that were part of different matches, we need to search for « {2,} » through the target file as a whole.

1. Prepare the “collect data” action on the File Selector and Action panels. Don’t worry about consecutive spaces just yet.
2. Start with a fresh sequence.
3. Click the New Step button on the Sequence panel to add the contents of the Action panel as the first step in the sequence.
4. Start with a fresh action.
5. Set the action type to “search-and-replace”.
6. In the Search box, enter the regular expression « {2,} ». This regular expression matches two or more consecutive line spaces.
7. In the Replacement box, enter a single space.
8. Click the New Step button on the Sequence panel to add the contents of the Action panels as the second step in the sequence.
9. With the second step still selected on the Sequence panel, select “target files from other step” in the “file selection” drop down list.
10. Select step 1 in the “step” drop-down list. The second step is now configured to process the target files created by the first step.
11. Click the Quick button on the Sequence panel to execute both steps. The first step collects the search matches. When that’s done, the second step condenses consecutive spaces.

35. Inspect Web Logs

While there is a lot of specialized software available for gathering useful information from web server logs, sometimes you want to get some information that standard web log analyzers do not offer.

PowerGREP is most useful for analyzing logs for which no specialized software is available. The basic concepts illustrated in this example are applicable to analyzing any kind of server or system log.

In this example, we will use Apache's extended log format. Most other web servers also use this format, or offer it as a choice. In this log format, each event gets one line in the log file:

```
bds1.66.14.88.130.gte.net - - [31/Jan/2005:00:06:55 -0500] "GET / HTTP/1.1" 200
8669 "http://www.google.com/search?q=regex+tutorial" "Mozilla/4.0 (compatible;
MSIE 6.0; Windows NT 5.1; SV1)" (In the actual log file, all this is on a single line.)
```

Each line consists of eight elements. If we assume that we will only apply our regular expression to valid log files, and therefore our regex need not exclude invalid log file lines, we can easily write the regular expression for each item:

1. Domain name or IP address of the client making the request: «\S+»
2. Basic authentication (two dashes in the above example, indicating no authentication): «\S+\s+\S+»
3. Date, time and time zone stamp: «\ [[^]]+\ \s+»
4. HTTP request, consisting of request method (GET), file (/) and protocol (HTTP/1.1): «"(?:GET|POST|HEAD) [^ "]+ HTTP/[0-9.]+»
5. Status code returned by the server (200): «[0-9]+»
6. Number of bytes served (8669): «[-0-9]+»
7. Referring URL, between double quotes: «"[^"]*"»
8. User agent, between double quotes: «"[^"]*"»

We can easily put all of this together. Items are separated by whitespace, which we match with «\s+». The result is:

```
«^\S+\s+\S+\s+\S+\s+\ [[^]]+\ \s+"(?:GET|POST|HEAD) [^ "]+ HTTP/[0-9.]+"\s+[0-9]+\s+[-0-9]+\s+"[^\s]"*\s+"[^\s]"*"$»
```

While this regular expression properly matches a server log line, it is not useful for collecting information. To make it useful, we have to add capturing groups, so we can collect only the information we want. To make things easy, we'll use named capturing groups. If we capture everything, and split the file in the HTTP request into file name and parameters, we get:

```
«^(?<client>\S+)\s+(?<auth>\S+\s+\S+)\s+\[(?<datetime>[^\]]+\)\ \s+"(?:GET|POST|HEAD)
) (?<file>[^ ?"]+)\??(?:<parameters>[^ ?"]+)? HTTP/[0-9.]+"\s+(?<status>[0-9]+\s+(?<size>[-0-9]+\s+"(?<referrer>[^\s]"*)"\s+"(?<useragent>[^\s]"*)"$»
```

Collecting Referring URLs

1. Select the log files you want to search through in the File Selector.
2. Start with a fresh action.

3. Set the action type to “collect data”.
4. Turn on “group identical matches” and “group results for all files”.
5. Set “sort collected matches” to “by decreasing totals” so we can see which URLs are most important.
6. Set “minimum number of occurrences” to 10 or higher, to avoid collecting too many URLs that are of no importance.
7. Search for the regular expression «"GET [^ ?"]+?\.\html?[^ "]* HTTP/[0-9.]+\s+[0-9]+\s+[-0-9]\s+" ([^"]*)"» which captures the referring URL in backreference one.
8. Enter “\1” in the Collect box, to collect referring URLs.
9. Click the Preview button to run the action.

The regex for matching complete log file entries was clipped at the start and the end to produce this example. By removing the parts we aren’t interested in, we speed up the action. Capturing groups we don’t care for were also removed. We’re capturing the HTTP request with «GET [^ ?"]+?\.\html?[^ "]* HTTP/[0-9.]+» to restrict matches to page hits only. This makes sure the statistics aren’t skewed, since most browsers send the same referrer information when loading images as when loading the page containing those images.

This action is available in the PowerGREP.pgl library as “Inspect Apache web logs - Referring URLs”.

If we want to collect referring sites (domain names) rather than complete URLs, we have to refine the regular expression, to separate the domain name from the rest of the URL. Instead of using «["^"]*», we will use «(?:-|http://([-a-z0-9]+) ["^"]*)». We are using two pairs of parenthesis now: the outer pair to group the pipe symbol, and the inner pair to create a backreference with the domain name part of the URL. The complete regular expression thus becomes:

```
«"GET [^ ?"]+?\.\html?[^ "]* HTTP/[0-9.]+\s+[0-9]+\s+[-0-9]+\s+"(?:-|http://([-a-z0-9]+) ["^"]*)"»
```

If the web browser did not pass referrer info, then the referrer item in the logs will show up as “-”, including the quotes. This is why we are using the pipe symbol to match this option, in addition to the domain name. If the dash was matched, the part of the regular expression in the capturing group will not have matched anything. In that case, the backreference will be empty. Since we only put \1 in the collect box, an empty string will be collected in that case.

This action is available in the PowerGREP.pgl library as “Inspect Apache web logs - Referring domains”.

36. Extract Google Search Terms from Web Logs

In the preceding example I showed you how to extract information and statistics from web logs. I will now build upon that example to accomplish a specific task: get a list of search terms that people used to find your web site in Google.

The regular expression for matching web log entries needs three adaptations. The first one is optional. I like to restrict the search to hits to web pages, so I've changed the part of the regular expression that matches the file in the HTTP request to `«/([-_a-z0-9]+\.\html)?»`

The second change is what makes this example work. Instead of using `«"[^"]*"»` to match any referring URL, we'll use `«(?:http://www\.google\.(?:com?\.)?[a-z]{2,3}/search\?.*?q=\+\+([\^\&"\r\n]++)["\r\n]*)»` to match only Google search pages, and extract the search terms. The `«http://www\.google\.(?:com?\.)?[a-z]{2,3}/search»` part matches URLs such as `http://www.google.com/search` on any country-specific top-level domain. The other part `«q=\+\+([\^\&"\r\n]++)["\r\n]*»` matches the `q` parameter in the search page URL. This parameter lists the URL-encoded search terms. The regex captures these into a backreference.

The third change speeds up the action. Since we only care about the HTTP request and the referrer, we can remove the parts of the regex before the HTTP request and after the referrer. The part of the regex matching the HTTP request cannot match anywhere else in the log entries, so our regular expression is still properly anchored.

Since the search terms are part of the referring URL, they are URL-encoded. Spaces have been substituted with pluses, and various other special characters are substituted with hexadecimal values. E.g. the plus itself was substituted with `%2B`, and the quote character with `%22`. When PowerGREP's "extra processing" feature, the search terms can easily be made readable again.

1. Select the log files you want to search through in the File Selector.
2. Open the PowerGREP.pgl library file included with PowerGREP. You can find it in the folder where PowerGREP is installed, `c:\Program Files\JGsoft\PowerGREP4` by default.
3. Select "Inspect Apache web logs - Google search terms" in the library, and click the Use Action button. This sets up the regular expression and extra processing as I explained above.
4. Click the Preview button to run the action.

When the action finishes running, the Results panel will show a list of search terms, sorted from most to least occurrences.

If you select the action "Inspect Apache web logs - Google search terms with landing pages" in the library, you will get a list of search terms paired with the page the visitor clicked on in Google's search results. Search terms without a page brought the visitor to the home page. The only difference in the action that shows landing pages is the text to be collected, which uses two backreferences instead of one.

The regular expression has two capturing groups. The first one matches the file name in the HTTP request, which is the landing page. The second group matches the Google search terms. The text to be collected uses `«\12»` (backslash ell two) to collect the search terms converted to lowercase, and `«\1»` to collect the landing page.

37. Split Web Logs by Date

Software that generates log files often dumps everything into a single log file. As the log file grows in size it becomes difficult to work with. Using PowerGREP you can easily split the log into multiple files, such as one file per day.

For this example we'll split an Apache web log which stores one log entry per line, with each entry storing the date formatted like 25/Apr/2010. The Merge Web Logs by Date example does the opposite.

1. Select the log files you want to split in the File Selector.
2. Start with a fresh action.
3. Set the action type to "split files".
4. In the "file sectioning" list, select "line by line, including line breaks". Each line in the file is one log entry.
5. Turn on the option "collect/replace whole sections". This makes sure lines will be extracted as a whole into the target files.
6. In the Search box, enter the regular expression «(? 'day' \d\d?) / (? 'month' Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec) / (? 'year' \d{4})». Only lines that match this regex are written to a target file.
7. In the Target File box, enter something along the lines of "c:\logs\web logs \${day} \${month} \${year}.txt.bz2" to build a path using replacement text syntax that includes the date from the regex match. Lines with the same target file (after substituting backreferences) are written to the same file. Lines with different target files are written to different files. We've added a .bz2 extension to the target file name to make PowerGREP automatically compress the file.
8. Set "between collected text" set to "nothing". Since we're collecting whole lines including line breaks, there's no need to add more delimiters.
9. Set the backup file options as you like them.
10. Click the Quick Split button to split the file. Use this button instead of Split Files. Otherwise PowerGREP will waste a lot of time and memory to display your entire log files on the Results panel.

Splitting files does not delete the original files. It may overwrite original files if the Target File for one or more search matches is a file that is searched through. When splitting files PowerGREP does not write the final target files until the action has completed. Overwriting source files won't alter the search matches.

This action is available in the PowerGREP.pgl library as "Split Apache web logs".

Recombining Log Files

The above example can also be used to recombine log files. Suppose your application writes log files to a certain size. It might write up to 100,000 entries in a single log file, and then start with a new file. Doing so keeps log file sizes manageable, but you'll end up with entries from multiple days in the same file, and entries from the same day in multiple files.

To recombine the logs so you'll have one file for the log entries of one day, simply mark all the files with your logs in the File Selector. Then execute the action described above. If log entries from different files result in the same target file, they'll be merged into that target file, even though you're executing a "split files" action. The key difference between "split files" and "merge files" actions is that "split files" calculates the target file for each search match, while "merge files" calculates the target file for each file searched through.

The “order of collected matches” drop-down list determines the order in which matches (log entries in this case) from different files are written when a “split files” action calculates the same target file path from matches from multiple files. This is important if you want your log entries in the combined file to have the same order as in the original files. If your original log files put the log entries in order if you sort the files alphabetically by name, then choose "sort files Alphabetically A..Z". If the time stamp on the log files puts the files in the correct order (e.g. the time stamp on each log file is the time the last entry was written) then you can choose “oldest file to newest file”.

38. Merge Web Logs by Date

Software that generates log files is often configured to start with a new log file every now and then, such as one log file per day. This is great for keeping file sizes small, but results in a large number of log files. If the log files are small it may be more convenient if you combine them into a smaller set of files.

For this example we'll merge an Apache web log which stores one log entry per line, with each entry storing the date formatted like 25/Apr/2010. The example assumes each file has the logs for one day. It combines these logs into one file per month. The Split Web Logs by Date example does the opposite.

1. Select the log files you want to merge in the File Selector.
2. Start with a fresh action.
3. Set the action type to “merge files”.
4. Leave the “file sectioning” set to “do not section files”. The “merge files” action always combines entire files. We don't need to use file sectioning to process log entries separately. The first date we find in the file determines the target file.
5. In the Search box, enter the regular expression `«(? 'day' \d\d?) / (? 'month' Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec) / (? 'year' \d{4})»`. Only files in which this regex can find a match are combined. PowerGREP only finds the first regex match in each file. As soon as one match is found, the file is merged.
6. In the “target file creation” drop-down list near the bottom of the Action panel, select “merge based on search matches”. This makes the Target File box visible.
7. In the Target File box, enter something along the lines of `“c:\logs\web logs ${month} ${year}.txt.bz2”` to build a path using replacement text syntax that includes the date from the regex match. Since only the first regex match is used, the first date found in the file determines the target file it is merged into. We've added a .bz2 extension to the target file name to make PowerGREP automatically compress the target file.
8. Set “between collected text” set to “nothing”. Apache log files already have a line break at the end.
9. The “order of collected matches” drop-down list determines the order in which our log files are combined. This is important if you want your log entries in the combined file to have the proper order. If your original log files put the log entries in order if you sort the files alphabetically by name, then choose “sort files Alphabetically A..Z”. If the time stamp on the log files puts the files in the correct order (e.g. the time stamp on each log file is the time the last entry was written) then you can choose “oldest file to newest file”.
10. Set the backup file options as you like them.
11. Click the Merge Files button to merge the files. A “merge files” action never lists anything but file names on the Results panel, so there's no difference between Merge Files and Quick Merge.

Merging files does not delete the original files. It may overwrite original files if the Target File for one or more search matches is a file that is searched through. When merging files PowerGREP does not write the final target files until the action has completed. Overwriting source files won't alter the search matches.

A “merge files” action always merges files as a whole. If you want to merge multiple files but put different parts of each file into different target files, essentially splitting and merging files at the same time, use a “split files” action. The Split Web Logs by Date example can do this.

This action is available in the PowerGREP.pgl library as “Merge Apache web logs”.

39. Split Logs into Files with a Certain Number of Entries

Dealing with large log files is often cumbersome. With PowerGREP you can easily split logs into separate files with a specific number of entries per log. E.g. splitlog0.txt would have the entries 0 to 999, splitlog1.txt has entries 1,000 to 1,999, and so on. You can put all the entries from the original log or logs into the target files, or you can extract only those entries that you're interested in.

1. Select the log files you want to split in the File Selector.
2. Start with a fresh action.
3. Set the action type to "split files".
4. In the "file sectioning" list, select "line by line, including line breaks". This works for logs that use one line for each entry.
5. Turn on the option "collect/replace whole sections". This makes sure lines will be extracted as a whole into the target files.
6. If you want all log entries to be in the split files, enter the regular expression «.» into the search box, and make sure "dot matches newlines" is off. This puts any line that is not blank into the target files. If you want only certain lines, enter a search term or regex that (partially) matches the log entries you want to extract. E.g. search for «^Error» to extract only those entries starting with the word "Error".
7. In the Target File box, enter something along the lines of "c:\logs\splitlog%MATCHNZ:/1000%.txt" to specify the target path. The match placeholder %MATCHNZ:/1000% takes the number of the match counting from zero, and divides it by 1000. This results in 0 for the first 1,000 matches, 1 for the second 1,000 matches, and so on. If you want the first log file to be number one, use %MATCHNZ:/1000+1%. Match placeholders use integer arithmetic that is calculated strictly from left to right. If you expect to have more than 10 but less than, say, 100 log files, you can pad the number in the target file name to have two digits by using %MATCHNZ:/1000:2Z% or %MATCHNZ:/1000+1:2Z% as the placeholder. 2Z means to pad with zeros to make the placeholder have at least two digits.
8. Set "between collected text" set to "nothing". Since we're collecting whole lines including line breaks, there's no need to add more delimiters.
9. Set the backup file options as you like them.
10. Click the Quick Split button to split the file. Use this button instead of Split Files. Otherwise PowerGREP will waste a lot of time and memory to display your entire log files on the Results panel.

Splitting files does not delete the original files. It may overwrite original files if the Target File for one or more search matches is a file that is searched through. When splitting files PowerGREP does not write the final target files until the action has completed. Overwriting source files won't alter the search matches.

This action is available in the PowerGREP.pgl library as "Split Logs into Files with a Certain Number of Entries".

Recombining Log Files

The above example can also be used to recombine log files. Say you previously split your logs into files with 1,000 entries and deleted the original logs. Now want the logs to be split into files with 2,500 entries each.

Follow the exact same steps as above. In the first step, select all the previously split log files. In step 7, use %MATCHNZ:/2500% as the placeholder.

In PowerGREP, a “split files” action can put search matches from one file into different target files. It can also put search matches from different files into the same target file. In our example, all matches from old logs 1 and 2 (with 1,000 entries each) are saved into new log number 1. The first 500 matches from old log number 3 are saved into new log number 1, and the remaining 500 are saved into old log number 2.

The “order of collected matches” drop-down list determines the order in which matches (log entries in this case) from different files are written when a “split files” action calculates the same target file path from matches from multiple files. This is important if you want your log entries in the combined file to have the same order as in the original files. If your original log files put the log entries in order if you sort the files alphabetically by name, then choose "sort files Alphabetically A..Z". If the time stamp on the log files puts the files in the correct order (e.g. the time stamp on each log file is the time the last entry was written) then you can choose “oldest file to newest file”.

40. Compile Indices of Files

By using path placeholders in a collect data action, you can easily index files with PowerGREP. Let's say you have a large number of HTML files saved into a particular folder. Now you want to compile a single index of those files.

1. Select the files you want to index in the File Selector.
2. Set the action type to "collect data".
3. Turn on "group results for all files" and "group identical matches". Since each file has only one TITLE tag, and we include the name of the file in the text to be collected, each text to be collected will be different. This means "group identical matches" won't really group anything, but it does allow matches to be sorted alphabetically.
4. In the Search box, enter the regular expression «<TITLE>(.*?)</TITLE>» and make sure to leave "case sensitive search" off. This regex will match an HTML title tag, and store its contents into the first backreference.
5. In the Collect box, enter "<P>\1</P>". The path placeholder %FILENAME% will be replaced with the name of the file in which the HTML title tag was found, and \1 will be replaced with the contents of the title tag.
6. Select to sort collected matches alphabetically, and set the minimum number of occurrences to one.
7. Select "save results into a single file" in the target file creation list.
8. Click the ellipsis (...) button next to "target file location", and select the name of the file you want to save your HTML index into.
9. Leave "between collected text" set to "line break" so each index entry we collect appears on its own line.
10. Turn on the "collect headers and footers" checkbox.
11. In the list that appears, click on "target file header". In the edit box next to that list, paste:


```
<html><head><title>HTML Index</title></head>
<body><h1>HTML Index</h1>
```
13. Select "target file footer" in the list and type in </body></html>. These two steps make sure we collect a valid HTML file.
14. Click the Collect button to run the action.

This action is available in the PowerGREP.pgl library as "Indexing HTML files".

How much information you can include in the index is up to your imagination. The above example is very minimal, to make it easy to understand. If you also want to include the first paragraph in each HTML file, you could search for:

```
«<TITLE>(.*?)</TITLE>.*?<P[^>]+>(.*?)</P>»
```

and collect:

```
<P><A HREF=\"%FILENAME%\">\1</A></P>
<UL>\2</UL>
```

This action is available in the PowerGREP.pgl library as "Indexing HTML files with first paragraph".

41. Make Sections and Their Contents Consistent

This example illustrates how you can use named capturing groups to carry over regex matches from the file sectioning to the main part of the action.

Suppose you have a number of HTML files, with headings such as `<h1>heading 4</h1>` that you want to make consistent. The 4 should be changed into a 1.

PowerGREP makes this easy. Use file sectioning to match the header tag and its contents. Then make the main action search-and-replace through the header, replacing numbers in the header's contents with the header's nesting level carried over from the file sectioning.

You can find this action in the PowerGREP.pgl library as “Make numbers in HTML heading tags consistent”.

1. Select the files you want to search through in the File Selector.
2. Start with a fresh action.
3. Set the action type to “search-and-replace”.
4. Select “search and collect sections” from the “file sectioning” list. Leave the section search type as “regular expression”.
5. In the Section Search box, enter the regular expression `«h(? 'headerlevel'[1-6])>(?'tag'.*)</h\k'headerlevel'>»` and make sure to leave “case sensitive search” off. This regular expression contains two named capturing groups, “headerlevel” and “tag”.
6. In the Section Collect box, enter the named backreference `“${tag}”` to restrict the main action to the contents of the tag.
7. In the Search box in the main part of the action, enter the regular expression `«\d+»` to match any number.
8. In the Replace box, enter the named backreference `“${headerlevel}”`
9. Set the target and backup file options as you like them.
10. Click the Preview button to run a test.
11. If all looks well, click the Replace button to update the headers.

When PowerGREP executes this action, the following happens for each file:

1. The sectioning regex matches a heading tag in the file, e.g. `„<h1>heading 4</h1>”`. The heading tag's number `„1”` is stored in the named group “headerlevel”, and the tag's contents `„heading 4”` are stored in the named group “tag”.
2. Because the section collect is set to a reference to the named capturing group “tag”, the main action will search only through the contents of the heading tag.
3. The main action matches the first number `„4”` in the heading tag's contents.
4. The main action replaces the matched number with the contents of the backreference “headerlevel”: `“1”`
5. The main action repeats steps 3 and 4 until all numbers have been replaced. In the example, the section after substitution becomes `“<h1>heading 1</h1>”`
6. PowerGREP repeats steps 1 through 5 for all heading tags in the file.

Updating the Heading Tags Themselves

Doing the opposite, updating a heading tag to make it consistent with numbers in the tag's contents, is almost as easy. What we'll do is replace `<h1>heading 4</h1>` with `<h4>heading 4</h4>`

You can find this action in the PowerGREP.pgl library as "Make HTML heading tags consistent with their contents".

1. Select the files you want to search through in the File Selector.
2. Start with a fresh action.
3. Set the action type to "search-and-replace".
4. Select "search for sections" from the "file sectioning" list. Leave the section search type as "regular expression".
5. In the Section Search box, enter the regular expression `<<h(? 'headerlevel' [1-6])>(?'tag' . *?)</h\k'headerlevel'>>` and make sure to leave "case sensitive search" off. This regular expression contains two named capturing groups, "headerlevel" and "tag".
6. Turn on the option "collect/replace whole sections".
7. In the Search box in the main part of the action, enter the regular expression `<<\b[1-6] \b>` to match a number between 1 and 6. The word boundaries also make sure we don't match the number in the heading tag itself.
8. In the Replace box, enter `<<h\0>${tag}</h\0>`
9. Set the target and backup file options as you like them.
10. Click the Preview button to run a test.
11. If all looks well, click the Replace button to update the headers.

When PowerGREP executes this action, the following happens for each file:

1. The sectioning regex matches a heading tag in the file, e.g. `„<h1>heading 4</h1>”`. The heading tag's number `„1”` is stored in the named group "headerlevel", and the tag's contents `„heading 4”` are stored in the named group "tag".
2. The main action searches through the entire section, i.e. tag with contents.
3. The main action matches the first number `„4”` in the heading tag. Because of the word boundaries in our regular expression, the `„1”` in `„h1”` is not matched.
4. The backreference `\0` in the replacement text is substituted with the regex match `„4”` and the named backreference "tag" is substituted with `„heading 4”` captured by the file sectioning. The result is `<<h4>heading 4</h4>`
5. Since we turned on "collect/replace whole sections", the whole section is substituted with the replacement, and the main action is done with this section.
6. PowerGREP repeats steps 1 through 5 for all heading tags in the file.

42. Generate a PHP Navigation Bar

Using path placeholders in collect data actions, you can easily compile indices of files.

Let's say you are developing a web site in PHP. You keep all the main PHP files in a separate folder. Now, you want to create a navigation bar in PHP. While you could do this by hand, automating this in PowerGREP will save you a lot of time, certainly if pages are frequently added and removed. Though I am using PHP in this example, the same principles apply to any other web scripting language.

The final PHP file should look like this, with the complete if statement repeated for every main page on the site:

```
function navbar($mainpage) {
    if ('currentpage' == $mainpage) {
        print '<B>currentpagetitle</B><BR>';
    } else {
        print '<A HREF="currentpage">currentpagetitle</A><BR>';
    }
}
```

Obviously a very simple navigation bar, but good enough to illustrate the idea.

1. Select the files you want to add to the navigation bar in the File Selector.
2. Start with a fresh action.
3. Set the action type to “collect data”. Leave the search type as “regular expression”.
4. In the Search box, enter the regular expression «<title>(.*?)</title>» and make sure to leave “case sensitive search” off. This regular expression matches an HTML title tag, capturing the title into the first backreference.
5. In the Collect box, enter the following six lines of text.
6. # \1
7. if ('%FILENAME%' == \$mainpage) {
8. print '\1
';
9. } else {
10. print '\1
';
- }
11. Select “save results into a single file” in the target file creation list.
12. Click the ellipsis (...) button next to “target file location”, and select the name of the file you want to save your PHP navigation bar into.
13. Leave “between collected text” set to “line break”.
14. Turn on the “collect headers and footers” checkbox.
15. In the list that appears, click on “target file header”. In the edit box next to that list, paste <?function navbar(\$mainpage) { and press Enter to add a line break after the {.
16. Select “target file footer” in the list and type in } ?>. These two steps make sure we collect a valid PHP file.
17. Click the Collect button to run the action to create a complete navigation bar.

Two techniques make this action work. The regular expression that searches for the title tag captures its contents into a backreference \1 which we use to insert the title into the collected data three times. The path placeholder %FILENAME% inserts the name of the file being searched through into the collected data. This example assumes that all HTML files are in the same folder. If not, you'll need to use another path

placeholder. The finishing touch is to use the headers and footers option in PowerGREP to wrap our PHP code into a complete PHP function.

You can find this action in the PowerGREP.pgl standard library as “Generate a PHP navigation bar”.

All that is left to do now is to include a reference to the navigation bar in each of the web pages, something you can also easily do with PowerGREP.

43. Include a PHP Navigation Bar

The previous example showed you how to generate a php navigation bar. This example shows you how to include a reference to the navigation bar in each of the PHP files.

1. Unless you still have the files for creating the navigation bar marked in the File Selector, mark the files you want to add the navigation bar to.
2. Start with a fresh action.
3. Set the action type to “search-and-replace”. Leave the search type as “regular expression”.
4. In the Search box, enter the regular expression «</h1>» and make sure to leave “case sensitive search” off.
5. Enter “\0<? requires('navbar.php'); navbar('%FILENAME%'); ?>” as the replacement text. You will probably also want to press Enter after the \0 to insert a line break into the replacement text, so the navbar stuff ends up on its own line, rather than after the </H1>.
6. Set the target and backup file options as you like them.
7. Click the Preview button to run a test.
8. If all looks well, click the Replace button to add the navigation bar reference to each file.

When PowerGREP find a match in c:\web\source\index.php, then the replacement text will become “<H1><? requires('navbar.php'); navbar('index.html'); ?>”. Whether your site consists of just a dozen or thousands of pages, inserting the reference with PowerGREP is easy and saves you a lot of time. Not to mention the potential errors if you have to type in the file names manually into each file.

You can find this action in the PowerGREP.pgl standard library as “Include a PHP navigation bar”.

Part 3

PowerGREP Reference

1. PowerGREP Assistant

The PowerGREP Assistant displays helpful hints as well as error messages while you work with PowerGREP. Select the Assistant item in the View menu to show or hide the PowerGREP Assistant. In the default layout, the assistant is permanently visible along the bottom of PowerGREP's window. Immediately above the assistant's caption bar, there is a splitter that you can drag with the mouse to make the assistant taller or shorter.

Helpful Hints

When you use the mouse, the assistant explains the purpose of the menu item, button or other control that you point at with the mouse. When you use the Tab key on the keyboard to move the keyboard focus between different controls, the assistant describes the control that just received keyboard focus. If you move the mouse pointer over the assistant, the assistant also explains the control that has keyboard focus, whether you pressed Tab or clicked on it.

Some of the assistant's hints mention other items that have an effect on or are affected by the control the assistant is describing. These are underlined in blue, like hyperlinks on a web site. When you click on such a link, the assistant moves keyboard focus to the item the link mentions. Since you can only click on a link when moving the mouse pointer over the assistant, you are only able to click on a link when the assistant is describing the control that has keyboard focus. After you've clicked, the assistant automatically describes the newly activated control.

Follow Mouse

If you find it distracting that the assistant's hint changes constantly as you move the mouse around, right-click on the assistant and select the Follow Mouse context menu item. By default, there's a checkbox next to that item to indicate that the assistant's hint follows the mouse pointer as described in the previous section. Selecting the Follow Mouse item removes the checkbox. The Assistant then only displays the hint for the control that has keyboard focus.

Error Messages

Most applications display error messages on top of the application, blocking your view of the application and whatever the error may be complaining about. Clicking OK brings the application back to life again, but then you have to remember what the problem was before you can fix it.

PowerGREP uses a different approach. When there is a problem, PowerGREP uses the Assistant panel to deliver the message. If you closed the assistant, it automatically pops up in the place it was last visible.

You can recognize error messages by their bold red headings. Hints have black headings. The assistant continues to show the error message until you resolve the problem, or dismiss the error by clicking the Dismiss link below the error message. Meanwhile, the assistant does not display hints or descriptions. If the

assistant was invisible before the error occurred, dismissing an error automatically hides the assistant. Otherwise, the assistant resumes showing hints.

Error messages disappear automatically when you fix the problem. E.g. if you click the Preview button without entering a search text, the error message automatically disappears if you enter a search text and click the Preview button again. So you don't need to dismiss errors, unless you want to see the hints again.

2. File Selector Reference

In the default layout, the File Selector is visible along the left side of the PowerGREP window. The File Selector displays a tree of folders and files, and enables you to select which files PowerGREP will work on.

Import File Listings

If you already have a text file with the list of files or folders that you want PowerGREP to search through, click the Import button to show the Import File Listings screen. This screen provides a variety of options to tell PowerGREP how to extract paths from the text file. You can tell PowerGREP to import the file listing just once, as a template for creating your own file selection in PowerGREP using the folders and files tree. You can also tell PowerGREP to import the file listing each time you execute the action so the text file becomes the actual file selection instead of the markings in the folders and files tree.

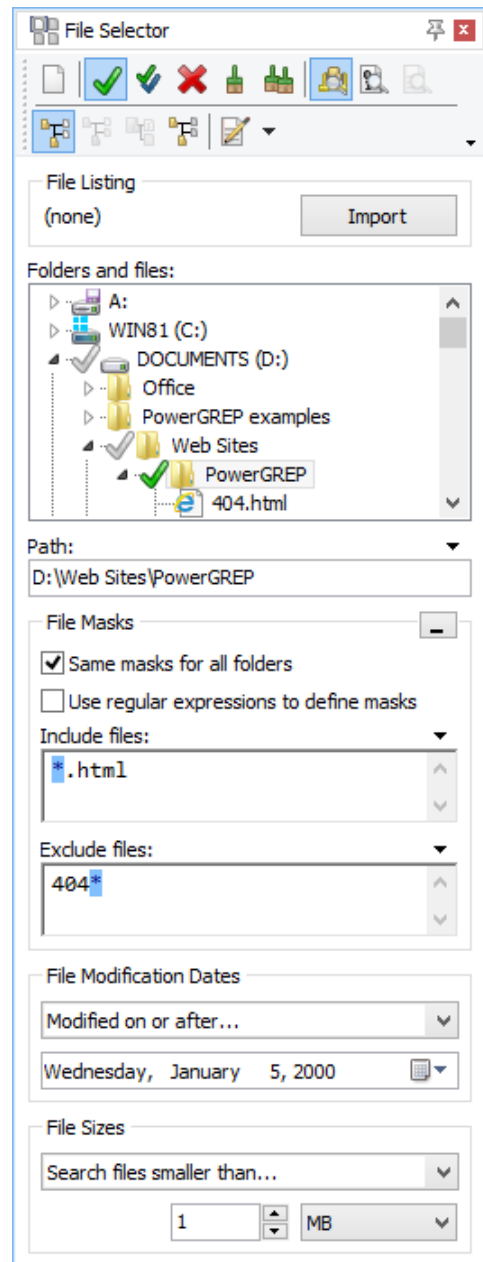
Folders and Files

The “folders and files” tree view shows all drives, folders and files on your computer. The only files not visible in the tree are those you excluded from all actions in the file selection preferences. By default hidden files, system files, and files with names that look like backup files created by PowerGREP are excluded.

The network node at the bottom of the tree provides you access to all network shares on your local area network. PowerGREP does not make any difference between files on your local PC, and files on other computers on your network. Unless you turn on the option to automatically scan the network in the preferences, network servers and shares only appear after you’ve typed in a UNC path.

To include a specific file in the next action, click on the file in the tree and select Include File or Folder from the File Selector menu, or click the corresponding button on the toolbar, or press the Ctrl+I keyboard shortcut. Alternatively, you can right-click on the file you want to include, and select the Include File or Folder item from the context menu. A green tick mark will appear next to the file, indicating it is marked for inclusion in the next action.

To include all the files in a particular folder, invoke Include File or Folder on the folder. A green tick mark will appear next to the folder, and gray tick marks will appear next to all files in the folder. The gray marks indicate the files are indirectly included, because you marked the folder. Files in subfolders of the included folders will not be included.



To include all files in a particular folder and all of its subfolders, select the folder and invoke the Include Folder and Subfolders command. A double tick mark will appear next to the folder you marked. Double gray tick marks will appear next to its subfolders. All the files the marked folder and its subfolders get gray tick marks.

If a file is included (gray tick) because you marked the folder it is in, you can exclude it from the action with the Exclude File or Folder choice. A red X will replace the gray tick next to the file.

If a folder is included (double gray tick) because you marked its parent folder, you can exclude it with Exclude File or Folder. This will also indirectly exclude all files and subfolders of the excluded folder. That is, they won't be included unless you explicitly mark them for inclusion.

When you change your mind about including or excluding a file or folder, select it and remove the mark with the Clear File or Folder command. To remove the marks from a folder and all the files and subfolders it contains, use Clear Folder and its Files and Subfolders. To start from scratch, select Clear in the File Selector menu. Clearing a file or folder is not the same as excluding it. If you exclude a file or folder, it won't be searched through no matter what. If you clear a file or folder, it may be searched through if you included its parent folder. In that case, a gray tick will appear after you clear the green tick or red X.

Single gray tick marks also appear next to drives and folders that directly or indirectly contain files that will be searched through. This makes it easy for you to find the files that are being included when most of the nodes in the tree are collapsed.

Searching Through a Single Folder

If you don't mark any files or folders with ticks or crosses then you can run a quick search through a single folder and its subfolders simply by selecting that folder in the folders and files tree and executing the action on the Action panel. The selected folder is not marked automatically. The next action you execute again depends on the folder selected in the folders and files tree. File masks and date and size filters are still taken into account.

Entering Paths with The Keyboard

Instead of navigating the folder tree, you can directly type in a path in the Path field just below the folder tree. The tree will automatically follow you as you type. You can also paste in a path from the clipboard.

To access files on the network, type in a UNC path. E.g. to access the network share "share" on the server "server", type \\server\share. That share will then appear under the Network node in the folders and files tree until you close PowerGREP.

To include the path you entered in the search, press Ctrl+I (Include File or Folder) or Shift+Ctrl+I on the keyboard. To include multiple paths, type in the first path and press (Shift+)Ctrl+I. The text in the Path field will become selected, so you can immediately type in the second path, replacing the first. Press (Shift+)Ctrl+I again to include the second path. To start over, press Ctrl+N to clear the file selection.

File Masks

With file masks you can include or exclude files by their names or extensions (i.e. file types). You can use traditional file masks, or regular expressions. Simply clear or mark the "use regular expressions to define masks" to make your choice.

In a traditional file mask, the asterisk (*) represents any number (including none) of any character, similar to «.*» in a regular expression. The question mark (?) represents one single character, similar to «.» in a regular expression. E.g. the file mask *.txt tells PowerGREP to include any file with a .txt extension.

Traditional file masks also support a simple character class notation, which matches one character from a list or a range of characters. E.g. to search through all web logs from September 2003, use a file mask such as `www.200309[0123][0-9].log` or `www.200309??.log`. To add a literal opening square bracket to a file mask, you need to place it into a character class. The closing square bracket has no special meaning outside of a character class. If you want to add a literal closing square bracket to a character class, place it immediately after the opening square bracket. So the file mask `*[[][0-9]].txt` matches file names like `whatever[1].txt`. In this file mask, `[[` is a literal opening bracket, `[0-9]` is a single digit, and `]` is a literal closing bracket.

You can delimit multiple file masks with any mixture of semicolons, commas, and line breaks. To search through all C source and header files, use `*.c;*.h`. To add a literal semicolon or comma to a file mask, either place the semicolon or comma between square brackets, or place the whole file mask between double quotes. The file mask `*[,]*.txt;"*;*.doc"` matches all .txt files that have a comma in their name, and all .doc files that have a semicolon in their name.

If you choose to use regular expressions to define masks, you have the full regular expression syntax at your disposal. Semicolons, commas, and line breaks are treated as delimiters between multiple regular expressions. To add a literal semicolon or comma to a mask defined by a regular expression, escape it with a backslash, or place it inside a character class.

One important difference between traditional masks and regular expressions is that the traditional mask must always match the whole file name, while a regular expression only needs to match part of a file name. E.g. the mask *.txt matches `joe.txt`, but not `joe.txt.doc` since the latter does not end in .txt. You could use the mask `*.txt*` to match both. However, the regular expression «.*\ .txt» will match both file names. In fact, «.\ .txt» has exactly the same effect. Use the regular expression «.\ .txt\$» with the end-of-string anchor to match only files with a .txt extension.

When you do not specify any file masks for a folder, all files in that folder are included. If you specify an inclusion mask, only files that match the inclusion mask will be included. If you specify an exclusion mask, all files matching the exclusion mask will be excluded from the next action. The exclusion mask takes precedence. If you specify both, files matching both will not be searched through. Note that *.* tells PowerGREP to search through all files with a dot in the file name, or files that have an extension. If you want to search through all files, leave the file mask blank instead of specifying *.*.

If a file is excluded from the search because of the file masks you specified, the gray tick next to it will disappear from the file tree, indicating the file was not included. Masks only apply to files that were included because you marked the folder containing them. If you directly mark a file, file masks do not apply to that file.

In the screen shot above, the folder "My Stuff" was marked for inclusion with Include File or Folder, as evidenced by the green tick next to it. The file "404.html" is not included, because it matches the exclusion

mask for the folder “My Stuff”. The file "atomic.html" is not included either, even though it matches the inclusion mask, because it was excluded with the Exclude File or Folder command.

By default, the same file masks are used for all folders that you marked for inclusion. If you want to use different masks for different folders, deselect the "same masks for all folders" option. After that, editing a mask will only edit it for the highlighted folder. If you turn on “same masks for all folders” again, the masks for all folders are immediately set to those displayed in the File Selector.

Example: Search through file names

File Masks with Folder Names

When you’ve marked a folder with Include Folder and Subfolders, you can use more complex file masks to filter out some of the folder’s subfolders, without marking each of them in the tree. To do this, simply use one or more backslashes in the file mask to indicate you want the mask to take into account the folder names. E.g. to exclude all files in folders named “junk”, use `*junk*` as the file mask.

When you don’t use any backslashes in the file mask, the file mask is compared with each file’s name only. When you do use backslashes, it is compared with the file’s path relative to the marked folder. E.g. if you marked "C:\My Documents\My Stuff", then the relative path of "C:\My Documents\My Stuff\Web Site\about.html" is "Web Site\about.html".

File Masks and Archives

PowerGREP treats archives such as .zip files as (compressed) folders. File masks such as `*.zip` are applied to files rather than to folders. That means you cannot use `*.zip` to control whether PowerGREP searches through .zip files. Instead, use the Search through Archives option in the File Selector menu. You can configure which file extensions PowerGREP should recognize as being archives in the Archive Formats section in the Preferences.

This rule does not apply when you execute a list files action without a search text, a file name search action, or a rename files action and you have the Search through Archives option turned off. In those situations, PowerGREP will treat archives as ordinary files. File masks such as `*.zip` do work in that situation, and the “list files” action will list the zip files themselves in the results.

Since PowerGREP treats archives as folder, the same file masks rules apply. To exclude all files in archives named "junk.zip", use `*junk.zip*` as the file mask.

File Modification Dates

After marking files and folders and specifying file masks, you can further reduce the files that will be searched through by filtering them by their modification dates. Unlike file masks, which do not affect files which are directly included (green tick), the file modification dates filter affects all files, whether they are directly (green tick) or indirectly (gray tick) included.

If a file with a green tick is excluded because of its modification date, the green tick will disappear. However, the File Selector will remember that you marked the file. If you cancel the file modification filter, the green tick will automatically reappear.

PowerGREP can treat modification dates in several ways:

- **Modified during the past...:** Only search through files that have been modified in a certain number of past hours, days, weeks, months or years.
- **Not modified during the past...:** Only search through files that were not modified in a certain number of past hours, days, weeks, months or years.
- **Modified on or after...:** Only search through files last modified on or after a specific date. Files last modified on the date you specify are searched through.
- **Not modified on or after...:** Only search through files that were last modified before a specific date. Files last modified on the date you specify are searched through.
- **Last modified between...:** Only search through files that were last modified on or between two specific dates. Files modified on those dates are searched through.
- **Not last modified between...:** Only search through files that were last modified before a specific date or after another specific date. Files modified on those dates are searched through.

When specifying a time period, PowerGREP starts counting from the start of the current period. E.g. if you tell PowerGREP to search through files modified during the last two hours at half past three, PowerGREP will search through files modified at or after one o'clock, two hours before the start of the current hour.

Weeks start on Monday. If you tell PowerGREP to search through files modified during the last week on Wednesday the 14th, PowerGREP will search through files modified on or after Monday the 5th. The only exception to this rule is when you limit the search to a number of weeks on a Sunday. Then, PowerGREP starts counting from the next Monday. The same search on Sunday the 18th will have PowerGREP search files modified on or after Monday the 12th.

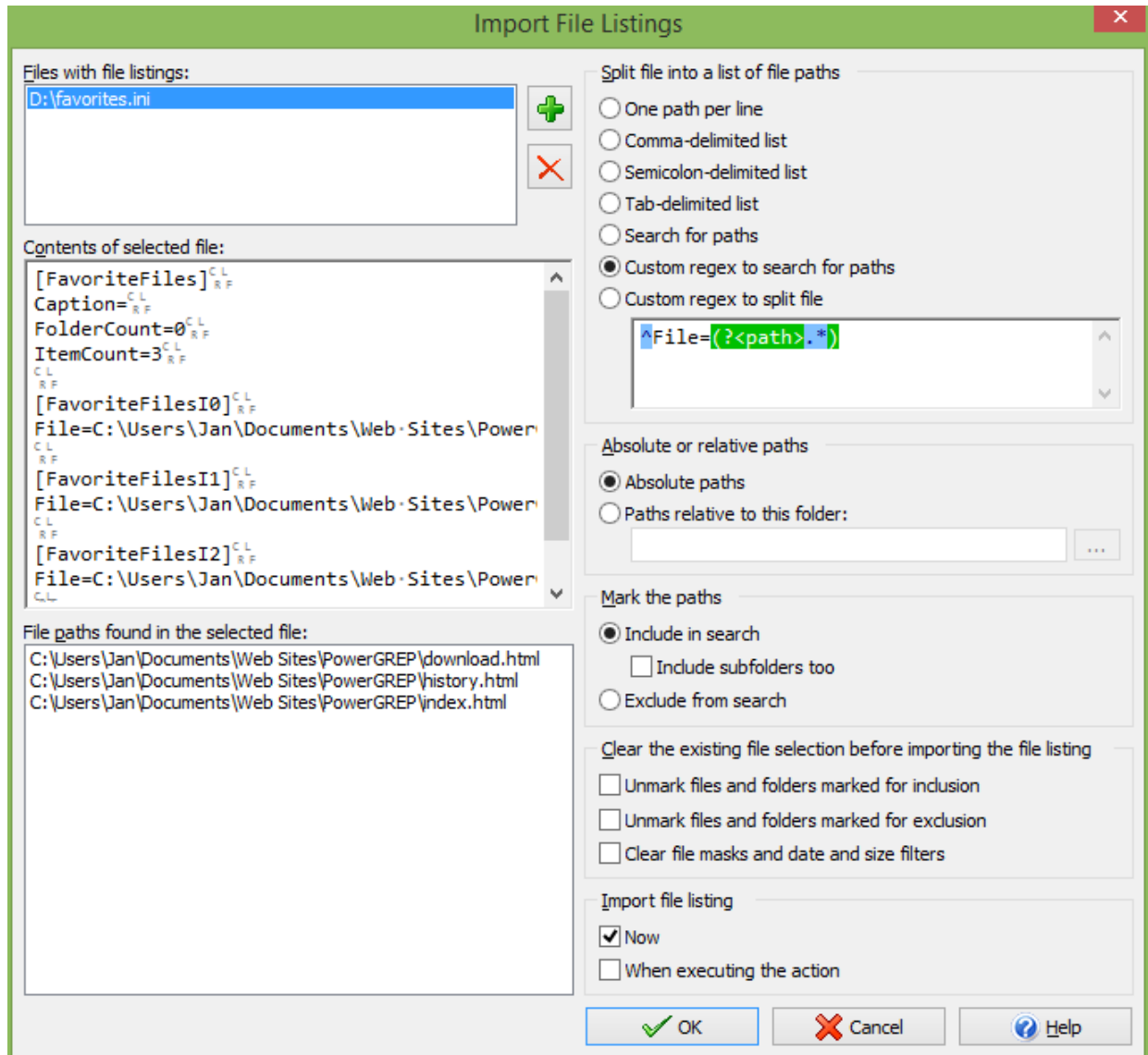
File Sizes

You can also further reduce the files that will be searched through by filtering them by their sizes. The file size filter affects all files, whether they are directly (green tick) or indirectly (gray tick) included.

Use the file size setting to specify that you only want to search through files smaller than or larger than a certain size, or files with a size between two sizes. You can specify sizes in bytes, kilobytes, megabytes, or gigabytes. 1 kilobyte equals 1024 bytes.

3. Import File Listings

If you already have a text file with the list of files or folders that you want PowerGREP to search through, click the Import button to show the Import File Listings screen. This screen provides a variety of options to tell PowerGREP how to extract paths from the text file.



First, click the button with the green plus symbol next to the “files with file listings” list to select one or more text files that list the files or folders you want to search through. You can use the green plus button repeatedly to add files from different folders to the list. The red X button deletes the selected file from the list.

Click on one of the files that you added to the “files with file listings” list. PowerGREP then shows the raw contents of that file in the “contents of the selected file” box. PowerGREP also shows the file and folder paths that it has detected in that file in the “file paths found in the selected file” list. The settings on the right hand side of the Import File Listings screen determine how PowerGREP detects those paths. PowerGREP automatically filters out anything that does not look like a valid path.

Choose one of the options in the “split file into a list of file paths” box to tell PowerGREP how your list of paths is delimited. If your list doesn’t use a consistent delimiter, select “search for paths” to tell PowerGREP to extract all absolute paths from the file, regardless of any other text that may occur in the file. If you want PowerGREP to extract only certain paths from the file, select one of the two “custom regex” options and type in a regular expression in the box below them. The “custom regex to search for paths” option needs a regular expression that matches the paths you want to mark in the file selection. PowerGREP uses the whole regex match as the path unless it contains a named capturing group called “path”. E.g. «`^File=(? 'path' .*)`» extracts the paths from “File” values in an .ini file. The “custom regex to split file” option needs a regular expression that matches the delimiters between those paths. E.g. «`[\r\n;]+`» allows line breaks and semicolons as delimiters.

If you select “absolute paths”, PowerGREP only uses fully qualified paths such as `c:\folder\file.txt` and `\\server\share\folder\file.txt`. Any relative paths in the file listing you’re importing are ignored. If you want PowerGREP to process relative paths as well then you need to select the “paths relative to this folder option”. Type in the base folder below that option, or click the (...) button to select it from a folder tree. Note that if your file contains text in addition to paths, you need to use one of the “custom regex” options to tell PowerGREP how to find only the actual paths. Otherwise, PowerGREP will treat each word in the text as a file name. You cannot use the “search for paths” option because that option finds absolute paths only, regardless of the “absolute or relative paths” setting.

Once you’ve set the options that make PowerGREP find the paths that you want to import, you need to tell PowerGREP what you want to do with those paths. The “mark the paths” option provides three choices. Select “include in search” without “include subfolders too” to mark each file or folder with a single green tick, just like the Include File or Folder item in the File Selector menu. Select both “include in search” and “include subfolders too” to mark each folder with a double green tick, just like Include Folder and Subfolders, while still marking files with a single green tick. The third option is to select “exclude from search”, which gives the file or folder a red X like the Exclude File or Folder menu item does.

If you previously marked files or folders in the tree in the file selector, whether you did that manually or by importing a file listing, those markings will remain in place unless you tick both “unmark files and folders” options. If you select only “unmark files and folders marked for inclusion”, then only green ticks are removed. If you select only “unmark files and folders marked for exclusion”, then only red X marks are removed. Leaving existing marks in place can be useful if you want to search through additional files or folders not present in the file listing.

Finally, you can choose when PowerGREP should actually import the file listing. If you select “now”, PowerGREP imports the file listing when you click the OK button. The files and folders tree on the File Selector panel will show you the result. The imported inclusion or exclusion marks become part of the file selection just like they do when you manually include or exclude files. There’s no way to distinguish between files and folders that you marked manually and those that were imported. Choose this option if you want to import the file listing just one time.

If you’re preparing a PowerGREP action that you’ll reuse in the future and the action needs to adapt whenever the text file with the file listings changes, then you need to select “when executing the action”. That tells PowerGREP to import the file listing whenever you execute the action, using the latest contents of the text file(s) you’re importing file listings from.

If you turn on both “unmark files and folders” options then you can turn on both the “now” and “when executing the action” options if you want to preview the imported file listings in the files and folders tree as well as make sure that PowerGREP always uses the latest file listings.

4. File Selector Menu

The File Selector menu lists commands for use with the File Selector. See the File Selector reference chapter for more information on the File Selector itself.

Clear

Removes all markings from all files and folders, and clears all file masks.

Open

Loads the file selection from a PowerGREP file selection file that you previously saved. PowerGREP action files and PowerGREP results files also contain file selection information. If you select an action or results file, only the file selection information will be read from the file.

You can quickly reopen a recently opened or saved file selection by clicking the downward pointing arrow next to the Open button on the File Selector toolbar. Or, you can click the right-pointing arrow next to the Open item in the File Selector menu. A new menu listing the last 16 opened or saved files will appear. Select “Maintain List” to access the last 100 files.

Save

Save the current file selection into a PowerGREP file selection file. You will be prompted for the file name each time.

All settings you made in the File Selector will be saved. That includes file markings, file masks, and the options to search through archives or binary files.

Favorites

If you often open the same files, you should add them to your favorites for quick access. Before you can do so, you need to save the file selection to a file. PowerGREP’s window caption will then indicate the name of the file selection file. Click the downward pointing arrow next to the Favorites button on the File Selector toolbar, or the right-pointing arrow next to the Favorites item in the File Selector menu. Then select “Add Current File Selection” to add the current file selection file to the favorites. Pick a file from the menu to open it.

If you click the Favorites button or menu item directly, a window will pop up where you can organize your file selection favorites. If you have many favorites, you can organize them in folders for easier reference later.

By default, the Favorites button is not visible on the toolbar. To make it visible, click on the downward pointing arrow at the far right end of the File Selector toolbar. A menu will pop up where you can toggle the visibility of all toolbar buttons.

Include File or Folder

Marks the file or folder you selected in the files and folders tree for inclusion in the next action. A green tick mark will appear next to the file or folder, indicating it is marked for inclusion in the next action. When you include a folder, gray tick marks will appear next to all files in the folder. The gray marks indicate the files are indirectly included, because you marked the folder. Files in subfolders of the included folders will not be included.

Include Folder and Subfolders

Marks the folder you selected in the files and folders tree for inclusion in the next action. A double green/blue tick mark will appear next to the file or folder, indicating it is marked for inclusion in the next action. Gray tick marks will appear next to all files in the folder and its subfolders. Double gray tick marks will appear next to the subfolders.

Exclude File or Folder

Excludes the file or folder you selected in the files and folders tree for inclusion from the next action, indicated by a red X. If you exclude a folder, all files and subfolders of the excluded folder will be excluded too, unless you explicitly mark them for inclusion.

Clear File or Folder

Removes the inclusion or exclusion mark from the file or folder you selected in the files and folders tree. Clearing a file or folder is not the same as excluding it. If you exclude a file or folder, it won't be search through no matter what. If you clear a file or folder, it may be searched through if you included its parent folder. In that case, a gray tick will appear after you clear the green tick or red X.

Clear Folder and its Files and Subfolders

Clears the selected folder like the “Clear File or Folder” command, and also clears all files and subfolders in that folder.



Mark Files with Search Results

This command is only available after you have previewed or executed an action. It removes all inclusion and exclusion marks, and then individually marks for inclusion all files in which search matches were found during the previous action. These files are indicated in the files and folders tree by "(matched)", after the name of the file. Since the files are all individually marked for inclusion, file masks are ignored.



Exclude Files with Search Results

This command is only available after you have previewed or executed an action. It marks all files in which search matches were found during the previous action to be excluded from the next action. Any folders that were marked for inclusion remain marked after using this command. Essentially, this command allows you to search again through the same set of files, minus those files in which you just found some search matches.



Search through Archives

This option is on by default. Toggle it to enable or disable searching through archives. When on, archives are treated as (compressed) folders, and PowerGREP will search through the files inside the archive. When off, archives are ignored completely. Since PowerGREP treats archives as (compressed) folders, you cannot use the file masks boxes on the File Selector to include or exclude them. You have to use the Search through Archives menu item. If an archive contains other archives, the files in those other archives will be searched through as well. PowerGREP can search through archives inside other archives without restrictions.

These rules does not apply when you execute a list files action without a search text, a file name search action, or a rename files action and you have the Search through Archives option turned off. In those situations, PowerGREP will treat archives as ordinary files.

You can configure which files PowerGREP treats as archives in the Archive Formats Preferences.



Search through Binary Files

This option is off by default. Toggle it to enable or disable searching through binary files. This option is ignored when you set the search type to "binary data". In that case, binary files are always searched.

You can influence which files are treated as binary files in the text encoding preferences. By default, PowerGREP will check the first 64K of each file for NULL bytes. If a NULL byte is found, PowerGREP treats the file as a binary file. Text files should not contain NULL bytes, while binary files frequently contain NULL bytes.

Since PowerGREP does not know whether a file is binary before reading the file, the File Selector does not indicate whether a file is binary or text. Even when not searching binary files, those files will have tick marks in the file selector. The results will indicate skipped binary files.



Search Only through Files with Results

Turn on this option to limit the next search to files that are listed in the search results. The files must also be marked for inclusion in the File Selector. If there are no previous search results, this option is ignored.

This option is useful to further narrow down search results. E.g. if you first search for “Joe”, and then turn on “search only through files with results” without making any other changes to the file selection, PowerGREP will restrict the search to those files containing “Joe”. If you then search for “Jack”, you will get a list of files containing both “Joe” and “Jack”.

If you know in advance that you only want files with both “Joe” and “Jack”, turn on the “list only files matching all terms” option on the Action panel instead.

Another way to use this option is to speed up executing an action for real after previewing it first. If you know none of the files were modified since you did the preview, turn on this option so PowerGREP doesn't needlessly search files without matches again.



Show All

Show all files and folders in the files and folders tree. Use this mode when deciding which files and folders to include in the next action.



Show Included Files

Show only files and folders that are directly or indirectly included, as well as their parent folders and drives, in the files and folders tree. Use this mode to reduce clutter when inspecting the results after you have previewed or executed an action.



Show Files with Results

Show only files in which search matches were found during the previous action, as well as their parent folders and drives, in the files and folders tree. These files are indicated by "(matched)" after the name of the file. Use this mode to reduce clutter when inspecting the results after you have previewed or executed an action. If no matches were found, the files and folders tree will be blank.



Show Folders

Show only folders in the files and folders tree. All folders are shown. No files are shown. Use this mode when you want to mark folders to be included in the next search and long lists of files are making the files and folders tree unwieldy.



Refresh

PowerGREP's File Selector automatically tracks changes to files and folders. Normally, there's no need to manually refresh the File Selector. Drive letters appear and disappear immediately when you insert and remove drives. When you collapse and re-expand a folder node, that folder is automatically refreshed if Windows notified PowerGREP that files or folders inside that folder were changed. When you execute an action, all files and folders that you marked to be part of the action are automatically refreshed as needed.

The only situation in which PowerGREP's File Selector won't be refreshed automatically is in the rare event that Windows does not notify PowerGREP of (all) changes to a particular drive. Should that happen, you can select the Refresh item in the File Selector. This tells PowerGREP to discard all information it keeps about files and folders. If you execute an action after refreshing the File Selector this way, PowerGREP will glob all folders in the action again, forcing file listings to be up-to-date.



Edit File

Opens the selected file in PowerGREP's built-in file editor. The editor can edit both text and binary files.

If you prefer to use an external editor or application to view or edit the file, first configure the editor or application in the external editors preferences. You can then click on the downward pointing arrow next to the Edit button on the toolbar, or the right-pointing arrow next to the Edit item in the File Selector menu, to open the selected file with that application. The applications that are associated with that file type in Windows Explorer are also listed in the Edit submenu.

If you configured an external editor as the default editor, then the Edit File command will invoke that editor instead of using PowerGREP's built-in editor. This saves you having to go through the Edit File submenu.



Open File in EditPad

Opens the selected file in EditPad. EditPad is a most convenient text editor. Just like PowerGREP, EditPad has been designed by Jan Goyvaerts and is sold by Just Great Software Co. Ltd. EditPad is available at <http://www.editpadpro.com/>.



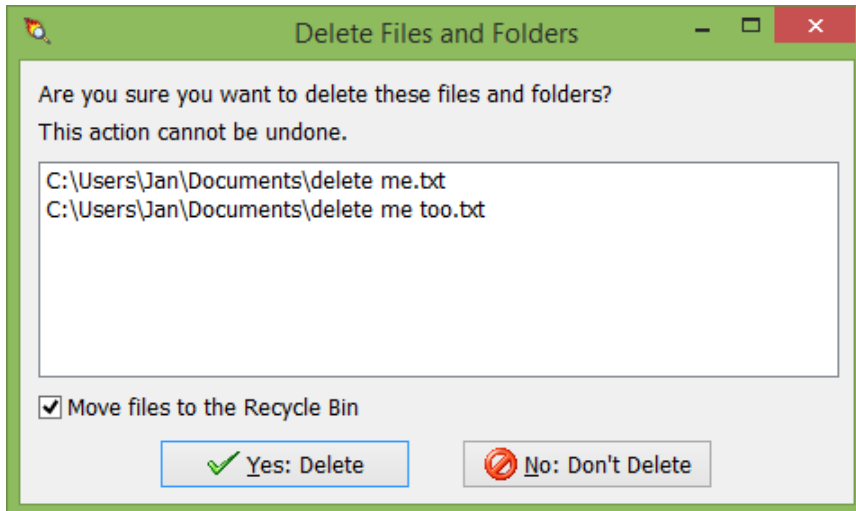
Delete Files

The Delete Files submenu of the File Selector menu allows you to delete four sets of files:

1. Delete Selected Files and Folders: Delete the files and folders that you have selected in the File Selector, whether they were part of the previous action or not.
2. Delete Matched Files: Delete all the files in which search matches were found during the previously executed action.
3. Delete Unmatched Files: Delete all the files that were searched through but did result in any matches during the previously executed action.

4. **Delete Target Files:** Delete all target files that were created during the previously executed action. Note that this is not the same as undoing the action. PowerGREP's undo history restores backup files. Deleting target files in the File Selector does not.

All four options ask for confirmation before actually deleting any files. The confirmation lists the files that will be deleted and gives you the option between moving the files to the Windows Recycle Bin or permanently deleting the files. Neither choice allows you to undo deleting the files in PowerGREP. If you choose to move the files to the Recycle Bin, you can recover the files manually from the Recycle Bin icon on your Windows desktop, at least until you make the Recycle Bin empty.



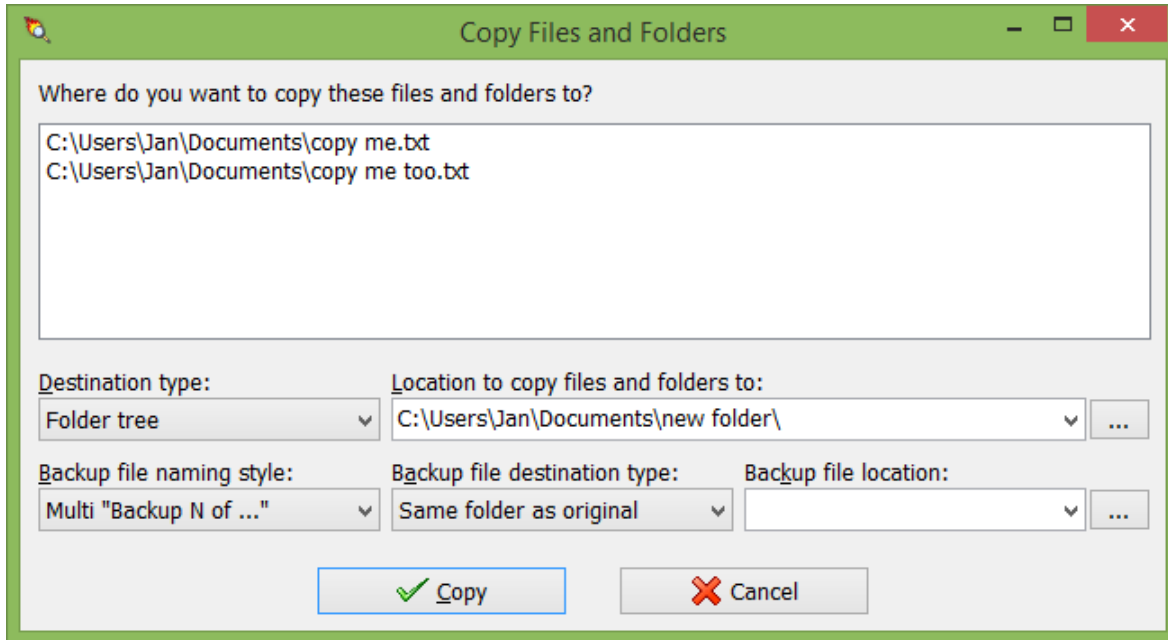
Copy Files

The Copy Files submenu of the File Selector menu allows you to copy four sets of files:

1. **Copy Selected Files and Folders:** Copy the files and folders that you have selected in the File Selector, whether they were part of the previous action or not.
2. **Copy Matched Files:** Copy all the files in which search matches were found during the previously executed action.
3. **Copy Unmatched Files:** Copy all the files that were searched through but did result in any matches during the previously executed action.
4. **Copy Target Files:** Copy all target files that were created during the previously executed action.

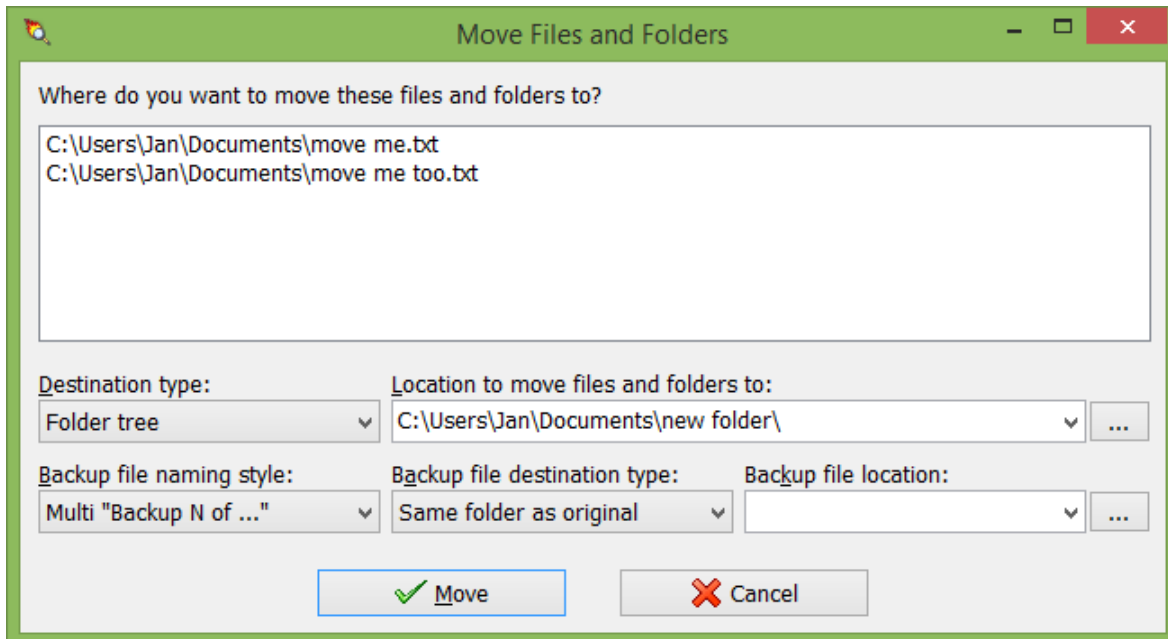
All four options show a screen listing all files that you are about to copy. You can specify the destination for the copied files and whether backups should be created for any files that are overwritten. These are the same target destination and backup options as on the Action panel.

After the files are copied a new item appears in the Undo History. There you can undo copying the files if you choose to create backups, or if no files were overwritten during the copy operation. The Undo History also allows you to clean up backup files when you're sure you don't want to undo the operation.



Move Files

The Move Files submenu of the File Selector menu offers the same options as the Copy Files menu. It shows the same screen with options. The only difference is that the files are moved rather than copied to their new locations. Move operations are also added to the Undo History.



5. Action Reference

The Action panel is the place where you define the task that PowerGREP will execute. The Action panel uses a dynamic user interface. Options that do not apply to the action you are defining will be invisible. This reduces clutter and confusion, and leaves more space to enter long lists of search terms. Since changing some of the options will make other options relevant or irrelevant, changing an option is likely to cause the Action panel to change its appearance.

All the options on the Action panel are arranged into nine parts. The parts are laid out from top to bottom in the order that PowerGREP uses them when you execute the action. Some parts are not available for certain action type. So when defining an action, start with selecting the action type at the top. Then work your way through the Action panel from top to bottom.

1. Action type: Tell PowerGREP what kind of action you want to execute: “simple search”, “search”, “collect data”, “list files”, “rename files”, “search-and-replace”, “search-and-delete”, “merge files”, or “split files”. Options that are specific to certain action types appear in this part when you select the action type. Some action types are more flexible than their names imply. E.g. the “list files” and “rename files” action types can also copy files if you choose that target type.
2. Filter files: Prior to performing the actual action on a file, search through that file’s contents to determine if this file should be processed or skipped. Since named capturing groups carry over from the filtering part to all following parts, you can also use the “filter files” feature to capture a part of the file’s text to be used in the following parts of the action. The “filter files” part is available for all action types except “simple search”.
3. File sectioning: You can make the main action search through only part of each file, or split up each file any way you want, rather than searching the whole file at once. A common choice is to process files line by line. Available for all action types except “rename files”.
4. Main action: The main part of the action is the set of search terms that perform the action you selected in the “action type” part. All action types have a main part. All action types except “list files” and “merge files” require at least one search term in the main part of the action. When you set the action type to “simple search”, the main part of the action is the only part where you can enter search terms.
5. Extra processing: Only used for “search-and-replace”, “rename files”, and “collect data” actions. You can apply an extra search-and-replace to the replacement text or the text to be collected in the main action.
6. Context: PowerGREP can display extra context around each search match on the Results panel if you use the “context” part of the action to collect that context.
7. Between collected text: Specify whether search matches collected into target files should be delimited with certain text and whether the target files should have header or footer text. Only used for the “search”, “collect data”, and “merge files” action types and only when “target file creation” is set to anything except “do not save results to file”.
8. Target and backup files: Tell PowerGREP where it should save collected search matches, whether you want to modify the original files or create a new set of target files, or whether you want to copy, move, or delete the listed files. The available options depend greatly on the action type and can even change its apparent function. E.g. selecting the “delete matching files” target type in combination with the “list files” action type essentially changes the action into a “delete files” operation. Only the “simple search” does not have any target options at all.
9. Comments: Enter a description of the action’s purpose before adding it to a PowerGREP Library or saving it into an action file.

When you're done defining the action, use the Preview, Execute or Quick Execute items in the Action menu to execute it. The Preview item is the safest one, since it will never modify any files, or do anything else you might regret.

The buttons on the Action toolbar that correspond with the Execute and Quick Execute menu items change their labels to indicate exactly what will happen. The labels change whenever you change the action type or target type.

Action

Preview Collect Quick Collect

Action type:
Collect data

List only files matching all terms Group identical matches

Filter files:
Do not filter files

File sectioning:
Do not section files

Search type:
Regular expression Non-overlapping search
 Case sensitive search Adapt case of replacement text Dot matches newlines

Search:
main·search·term

Collect:

Extra processing. Perform a search and replace on the replacement text or collect text.

Context type:
No context

Between collected text:
Line break Collect headers and footers

Target file creation: Save results into a single file
Target file destination type: Single folder
Target file location:

Target file text encoding: Same as original file
Target file line break style: Same as original file
Order of collected matches: No particular order

Backup file naming style: Multi "Backup N of ..."
Backup file destination type: Same folder as original
Backup file location:

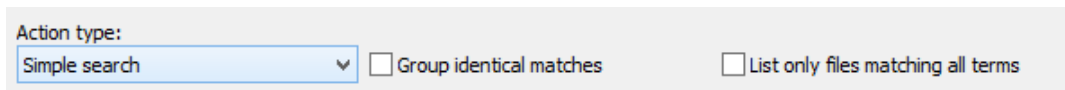
Comments:
Action panel showing all nine parts.

6. Action Types

The first thing to do on the Action panel is to use the "action type" drop-down list to select the kind of action you want to execute. The action type largely defines which settings are available on the Action panel. The combination of action type and target type determines what PowerGREP does with the search matches found by the main part of the action or with the files in which those matches are found. PowerGREP offers nine action types:

1. Simple search
2. Search
3. Collect data
4. List files
5. File name search
6. Rename files
7. Search-and-replace
8. Search-and-delete
9. Merge files
10. Split files

Simple Search



The "simple search" action type hides most of the controls that normally give the Action panel its complicated appearance. Use this action type if you just want to search for one or more search terms through a bunch of files and see the results in PowerGREP. The main differences between "simple search" and "search" are that "simple search" does not support filtering files or creating target files and that file sectioning and context only offer a choice between "off" and "line by line".

Group Identical Matches

Turn on to collect identical search matches only once. Turn off to list all search matches, including identical ones. When grouping identical matches, the Editor will not highlight matches in the source file, and the Results will display matches without context.

List Only Files Matching All Terms

Turn on to list or process only files that match all of the search terms. Turn off to list or process all files matching one or more of the search terms. This option is only available when the search type of the main part of the action is a list or a delimited set of search terms.

Search

Action type:
 Search

List only files matching all terms
 Group identical matches
 Group results for all files

Sort collected matches: Minimum number of occurrences:
 Alphabetically, A..Z 1

The “search” action type extends “simple search” with additional capabilities. You can use the “filter files” action part to exclude files in which one or more search terms can be found. Essentially, you can search for files matching “A and not B” by having the main part of the action search for A and the file filtering search for B. The “file sectioning” and “context” parts of the Action panel also show their full sets of options. At the bottom of the Action panel, you can set target and backup options to save the search matches into one or more files.

At the top of the Action panel, the “search” action type initially shows the same two “list only files matching all terms” and “group identical matches” checkboxes as the “simple search” action type. Additional options appear when you turn on “group identical matches” that are not available with “simple search”. These options are useful when saving search matches into files.

Group Results for All Files

Turn on to produce one set of results for all files searched through. Turn off to produce a separate set of results for each file.

Sort Collected Matches

Select the order in which the collected matches should be saved into the target file.

- **Alphabetically, A..Z:** Alphabetically, from A to Z.
- **Alphabetically, Z..A:** Alphabetically, from Z to A.
- **By increasing totals:** Count how often each match occurs, and sort matches from least occurrences to most occurrences.
- **By decreasing totals:** Count how often each match occurs, and sort matches from most occurrences to least occurrences.

Minimum Number of Occurrences

Set to 1 to save all matches into their target files, regardless of how many times each match occurs. Set to a number higher than 1 to save only those matches that occur that many times. Matches that occur fewer times are eliminated from the results. They are not saved into target files, they aren’t listed on the Results panel, and they aren’t highlighted in the Editor.

Example: Extract or delete lines matching one or more search terms

Collect Data

The screenshot shows the configuration for the 'Collect data' action type. It includes a dropdown menu for 'Action type' set to 'Collect data'. Below this are three checkboxes: 'List only files matching all terms' (unchecked), 'Group identical matches' (checked), and 'Group results for all files' (checked). There are also two more dropdown menus: 'Sort collected matches:' set to 'Alphabetically, A..Z' and 'Minimum number of occurrences:' set to '1'.

The “collect data” action type runs a search just like the “search” action type. It provides all the same options, with an extra edit control in the main part of the action labeled “collect”. There you can enter the text to be collected each time a search match is found. When using regular expressions, you can use the same syntax you use for the replacement text in a search-and-replace. When searching for a list of search terms, you can enter a different text to be collected for each search match. When using a delimited list of search terms, you can enter the search terms and their corresponding texts to be collected as delimited pairs. If you don’t type in any text to be collected for a particular search term, the actual search match is collected, just like the “search” action type does.

If you want to manipulate the text to be collected beyond what can be easily done with capturing groups and backreferences, turn on the “extra processing” to run an extra search-and-replace on the text to be collected for each search match. In the text to be collected and in the extra processing you can use backreferences to named capturing groups from regular expressions in the “filter files”, “file sectioning” and main parts of the action.

Examples: Find email addresses, Collect page numbers, Collect XML Data with entities replaced, Inspect web logs and Extract Google search terms from web logs

List Files

The screenshot shows the configuration for the 'List files' action type. It includes a dropdown menu for 'Action type' set to 'List files' and another dropdown menu for 'What to list' set to 'File name only'. Below these are two checkboxes: 'Invert search results' (unchecked) and 'List only files matching all terms' (unchecked).

The “list files” action type does not require a search term in the main part of the action. If you don’t provide a search term, PowerGREP simply lists all files that you included in the File Selector. If you do provide a search term, PowerGREP searches through the contents of the files to find it. Files that contain the search term are listed, while files that do not contain the search term are not listed. You can do the opposite by turning on “invert search results”. If you want to do both, listing files that match “A and not B”, turn off “invert search results”. Set the main action to search for A and use the “filter files” option to exclude files containing B.

Search matches are never listed in the results. Only file names are. At the top of the Action panel you can choose to list only file names, paths relative to the folder marked in the File Selector, or complete paths. This makes the “list files” action type significantly faster than “simple search” or “search” because “list files” continues with the next file as soon as the first search match is found in a file, while the other two action types always try to find all search matches in each files so they can be listed in the results.

The target options for “list files” actions allow you to save the list of paths into a file as well as copy, move, or delete the files that were found.

Example: Find files not containing a search term

File Name Search

The screenshot shows the configuration panel for the 'File Name Search' action. It features two dropdown menus: 'Action type' set to 'File name search' and 'What to search through' set to 'Full path'. Below these are two checkboxes: 'Invert search results' and 'List only files matching all terms', both of which are currently unchecked.

Set the action type to “file name search” if you want the main part of the action to search through the names of files rather than their contents. The “what to search through” setting determines whether only the file’s name, the file’s path relative to the folder you’ve marked in the File Selector, or the file’s full path is searched through. If you set the target options to save the list of files that was found, then the “what to search through” setting also determines the part of the file’s path that is saved into the target file.

If you turn on “invert results”, then the results show the file paths in which none of the search terms can be found. If you turn on “list only files matching all terms”, you get the file paths in which all the search terms can be found. If you turn on both these options, you get a list of all paths in none of the search term or some of the search terms but not all of the search terms can be found.

There are two ways to search through both file names and file contents. The simplest way is to use the “file name search” action type. Use the main part of the action to search through file names, and use the “filter files” option to search through the contents of the files. Filtering files is always done based on the contents of the files, even for “file name search” actions.

The other way to search through both file names and file contents is to use the “include files” and “exclude files” boxes in the File Selector to include or exclude files based on their file names. Then you can use the Action panel to search through the contents of the included files using the “list files” action type or one of the other action types that search through the contents of files.

Rename Files

The screenshot shows the configuration panel for the 'Rename Files' action. It features two dropdown menus: 'Action type' set to 'Rename files' and 'What to rename' set to 'File name only'. To the right of these is a checkbox labeled 'List only files matching all terms', which is currently unchecked.

The “rename files” action type makes the search term in the main part of the action works on file names rather than on file contents. The new name of each file is determined by running a search-and-replace on the file’s name.

The “rename files” action type can do more than just rename files. At the top of the Action panel you can choose if this search-and-replace should be performed on the file’s name only, or on the file’s path relative to the folder marked in the File Selector, or on the file’s complete path. If you search-and-replace through the file’s name only, the file is renamed and stays in the same folder. If you search-and-replace through the path relative to the folder, the file may be moved into a different subfolder of that folder depending on the result

of the search-and-replace. If you search-and-replace through the full path, the file can be moved anywhere. It is your responsibility to make sure the result is a valid path. If you don't, PowerGREP skips files that don't get a valid path.

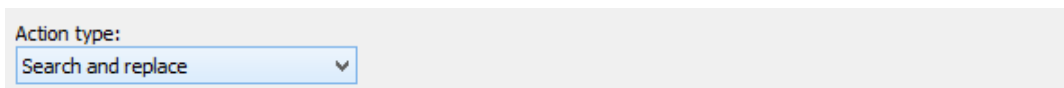
Beyond renaming and moving files, you can also copy them by setting the “target file creation” option to “copy files”. You can even add files to .zip archives or extract them from any archive format that PowerGREP supports. If you rename `c:\file.txt` into `d:\archive.zip::file_from_c.txt` then `file.txt` is added to `d:\archive.zip` under the new name `file_from_c.txt`. Doing the reverse extracts the file from the archive. PowerGREP uses a double colon to delimit archives from the path of each file inside the archive. The archive's extension must be one that is configured in the Archive Formats section in the Preferences. Similarly, renaming `file.txt` into `file.txt.bz2` compresses this file, while renaming `file.gz` to `file.txt` decompresses that file. You can even decompress and recompress by renaming `file.gz` into `file.txt.bz2`. This works for any extension that is configured as a single file compressed format in the Archive Formats preferences.

The “rename files” action type does allow you to search through the contents of each file using the “filter files” settings. You can use this to rename only certain files based on their contents. Even more powerful is to add named capturing groups to the regular expression for “filter files”. You can then use backreferences to that named capturing group in the regular expression and/or replacement text of the main part of the action. This enables you to use a specific part of the file's contents in its name.

The “extra processing” part of the Action panel is also available. You can use it to run an extra search-and-replace on the replacement text that will be used to rename the file.

Examples: Replace in file names and contents and Rename files based on HTML title tags

Search and Replace



When you select Clear in the Action menu and set the action type to “search and replace” you can easily run a search-and-replace as you would in any text editor that supports regular expressions. Simply type in your search text into the Search box and the replacement text into the Replacement box in the main part of the action and click the Replace button in the toolbar.

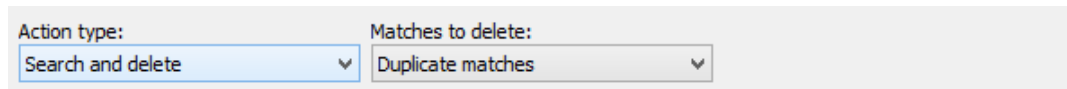
But PowerGREP provides a lot more options on the Action panel. The “filter files” settings allow you to restrict the search-and-replace to files in which a separate set of search terms can or cannot be matched. With the “file sectioning” options you can split the files into lines or other sections and search-and-replace each line or section separately. You can also have lines or sections replaced as a whole.

When using regular expressions, you can use backreferences in the replacement text to capturing groups not only from the main part of the action, but also to named capturing groups from the “file sectioning” and “filter files” parts. If capturing groups and backreferences aren't enough to build up the replacement text, you can turn on “extra processing” to run an extra search-and-replace on each replacement text.

The “context” settings aren’t used to execute the search-and-replace. They are only used to display the results in PowerGREP. Collecting extra context can be very useful if you plan to manually make or revert replacements after previewing or executing the search-and-replace.

Examples: Delete repeated words, Add a header and footer to files, Add line numbers, Insert proper HTML title tags, Replace HTML tags, Replace HTML attributes, Put anchors around URLs that are not already inside a tag or anchor, Replace in file names and contents and Apply an extra search-and-replace to target files

Search and Delete



The image shows a configuration panel for the 'Search and Delete' action type. It contains two dropdown menus. The first, labeled 'Action type:', is set to 'Search and delete'. The second, labeled 'Matches to delete:', is set to 'Duplicate matches'.

The “search and delete” action type is essentially the “search and replace” action type without any replacement text. Search matches are replaced with nothing or deleted. There is one extra option that allows you to delete only certain matches.

This action type deletes search matches rather than files. If you want to delete entire files in which search matches are found, use the “list files” action type together with the “delete files” target type.

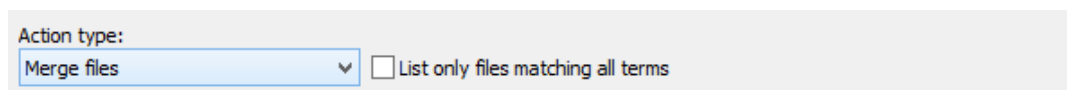
Matches to Delete

Choose which search matches you want to delete:

- **All matches:** Delete all search matches.
- **Duplicate matches:** Delete search matches only if the same text was already matched in the file.
- **Duplicates, separately per search term:** Delete search matches only if the same text was already matched in the file by the same search term.
- **Second and following matches:** Delete the second and following search matches in the file, regardless of whether the same or different text was matched.
- **Second and following, separately per search term:** Delete the second and following search matches of each search term in the file, regardless of whether the same or different text was matched.

Example: Extract or delete lines matching one or more search terms

Merge Files



The image shows a configuration panel for the 'Merge Files' action type. It contains a dropdown menu labeled 'Action type:' set to 'Merge files' and an unchecked checkbox labeled 'List only files matching all terms'.

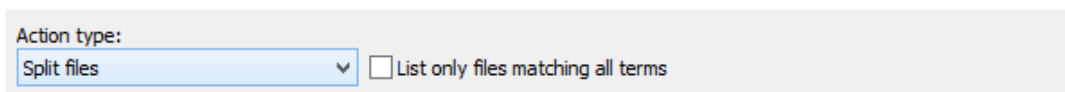
The “merge files” action type can be used with or without search terms in the main part of the action to gather a list of files as described for the “list files” action type. Whenever a file is found to match, the file’s entire contents are saved into a new file. PowerGREP overwrites the file if it existed before the action started. If you specify the same target file for two or more matching files (or even for all files) in a single “merge files”

action then those files are merged together into the target file. You can use the “between collected text” settings on the Action panel to specify if any text should appear between the files when they are merged together and whether the target files should have some header or footer text.

If you set the “target creation type to ”merge based on search matches“ then a box labeled ”target file” appears in the main part of the action. This box allows you to enter a replacement text as you would for a search-and-replace, including backreferences and “extra processing”. The difference is that PowerGREP expects your replacement text to be a valid path. That is the path that PowerGREP merges the file into.

Example: Merge web logs by date

Split Files



The “split files” action type is identical to the “collect data” action type, except for one thing: instead of a text to be collected for each search match you need to provide a full path to a target file for each search match. You can use all the same syntax for backreferences and “extra processing” to build up this target path. The search match is saved into this file. PowerGREP overwrites the file if it existed before the action started. If you specify the same target file for multiple matches in a single “split files” action then they are all saved into that target file. This works even if the matches were found in different source files, so you can essentially split and merge at the same time. You can use the “between collected text” settings on the Action panel to specify if any text should appear between the collected matches and whether the target files should have some header or footer text.

Example: Split web logs by date

7. Search Terms and Options

The Action panel always provides space for at least one set of search terms: the search terms used by the main part of the action. The action panel may provide space for four other sets of search terms used by four other parts of the action: “filter files”, “file sectioning”, “extra processing”, and “context”. How many of those are available depends on whether your chosen action type uses those parts, and whether you’ve selected an option for those parts that requires the part to use a search term. E.g. if you set “file sectioning” to “line by line” then the “file sectioning” part of the action doesn’t need any search terms. It does when you set it to “search for sections”. If you turn on “extra processing” that part requires one or more search-and-replace pairs. If you turn off “extra processing”, that part doesn’t show anything but its checkbox.

Search Types: Text, Regex or Binary

PowerGREP can search for four kinds of items:

- Literal text: A literal word, phrase or text fragment that must appear exactly this way in the search text (except for case).
- Regular expression: A pattern describing the format of the text you want to find. This is the most powerful and flexible way to search.
- Free-spacing regular expression: A regular expression that ignores spaces and comments in the pattern, allowing you to format it freely.
- Binary data: a literal block of bytes which you enter in hexadecimal mode.

Most of the time you will be working with regular expressions. They allow you to specify the form of the text or data you want to search for, rather than entering the exact text or data you want to find. By using regular expressions, you can unleash PowerGREP’s full potential. Automating search or text processing tasks using PowerGREP and regular expressions will save you a lot of time and tedious work.

If you are new to regular expressions, the regular expression tutorial in this manual will teach you everything you need to know. Given an hour or two of practice, you will soon be up to speed.

You probably also want to have a look at [RegexBuddy](#) and [RegexMagic](#). Both products are available separately. [RegexBuddy](#) makes it much easier to work with the regular expression syntax to create and edit regular expressions for use with PowerGREP and a variety of other tools and programming languages. While editing a regular expression in PowerGREP, simply click the [RegexBuddy](#) button in the Action toolbar to edit it with [RegexBuddy](#). [RegexMagic](#) allows you to generate regular expressions without dealing with the regular expression syntax at all. [RegexMagic](#) supports all popular regular expression flavors, including the one used by PowerGREP. Simply click the [RegexMagic](#) button in the Action toolbar to invoke [RegexMagic](#) to generate a regular expression.

All four search types allow you to use match placeholders and path placeholders. Match placeholders allow you to insert search matches and search match numbers. Path placeholders are substituted with various parts of the name and path of the file being searched through. Use them to search for and/or to create file references. You can disable placeholders in the action & results preferences if they conflict with text you’re searching for.

Examples: Add line numbers, Collect page numbers and Update copyright years

Search Types: Single, List or Delimited

The search items can be entered in three ways. For free-spacing regular expressions, only the “single item” and “list” entry methods are available. The other three kinds of search items support all three entry methods.

The screenshot shows the search configuration window with the following settings:

- Search type:** Literal text (selected in a dropdown)
- Non-overlapping search
- Case sensitive search
- Adapt case of replacement text
- Whole words only
- Search:** search·text
- Replacement:** replacement·text

Single item: Enter just one literal piece of text, one regular expression, or one chunk of binary data. PowerGREP will give you one edit box for the search text, and one edit box for the replacement text or the text to be collected (if any).

The screenshot shows the search configuration window with the following settings:

- Search type:** List of literal text (selected in a dropdown)
- Non-overlapping search
- Case sensitive search
- Adapt case of replacement text
- Whole words only
- Search:** first·search·term
- Replacement:** first·replacement·text
- List of items:**
 - 1: first search term
 - 2: second search term
 - 3: third search term
 - 4: canceled search term
 - 5: fourth search term

List: Enter multiple items, one by one. PowerGREP will give you one edit box for the search text, and one for the replacement or collection text, plus a list to add, remove and rearrange the items. Each item in the list will have a check box. Clear the check box to disable the item without deleting it from the list. This can help you experiment with different alternatives.

Example: Boolean operators “and” and “or”

The screenshot shows the search configuration window with the following settings:

- Extra processing. Perform a search and replace on the replacement text or collect text.
- Extra processing search type:** Delimited literal text (selected in a dropdown)
- Non-overlapping search
- Case sensitive search
- Adapt case of replacement text
- Whole words only
- Extra prefix label delimiter:** :
- Extra term delimiter:** ;
- Extra pair delimiter:** =
- Extra processing search:** first:before·after·second:old·new·last:third·search·term·third·replacement·text

Delimited: PowerGREP will give you one edit box to enter multiple search terms, or multiple search-and-replace or search-and-collect pairs. This way of entering the search terms is most convenient if you already have them in some sort of delimited format. You can copy and paste them into the edit box. If the search terms are stored in a delimited text file, you can right-click on the box and select Insert File in the context menu to load the search terms from the file.

The search prefix label delimiter is optional. If you specify one, you can use it to prefix search search term with a descriptive label. The search item delimiter delimits search terms, or search-and-replace or search-and-collect pairs. The search pair delimiter separates each search term from its substitution.

All three delimiters must be unique. You can use any sequence of characters that does not occur in any of the regular expressions or replacement texts you'll be working with.

Example: Collect XML Data with entities replaced

Replacement Text or Text to Be Collected

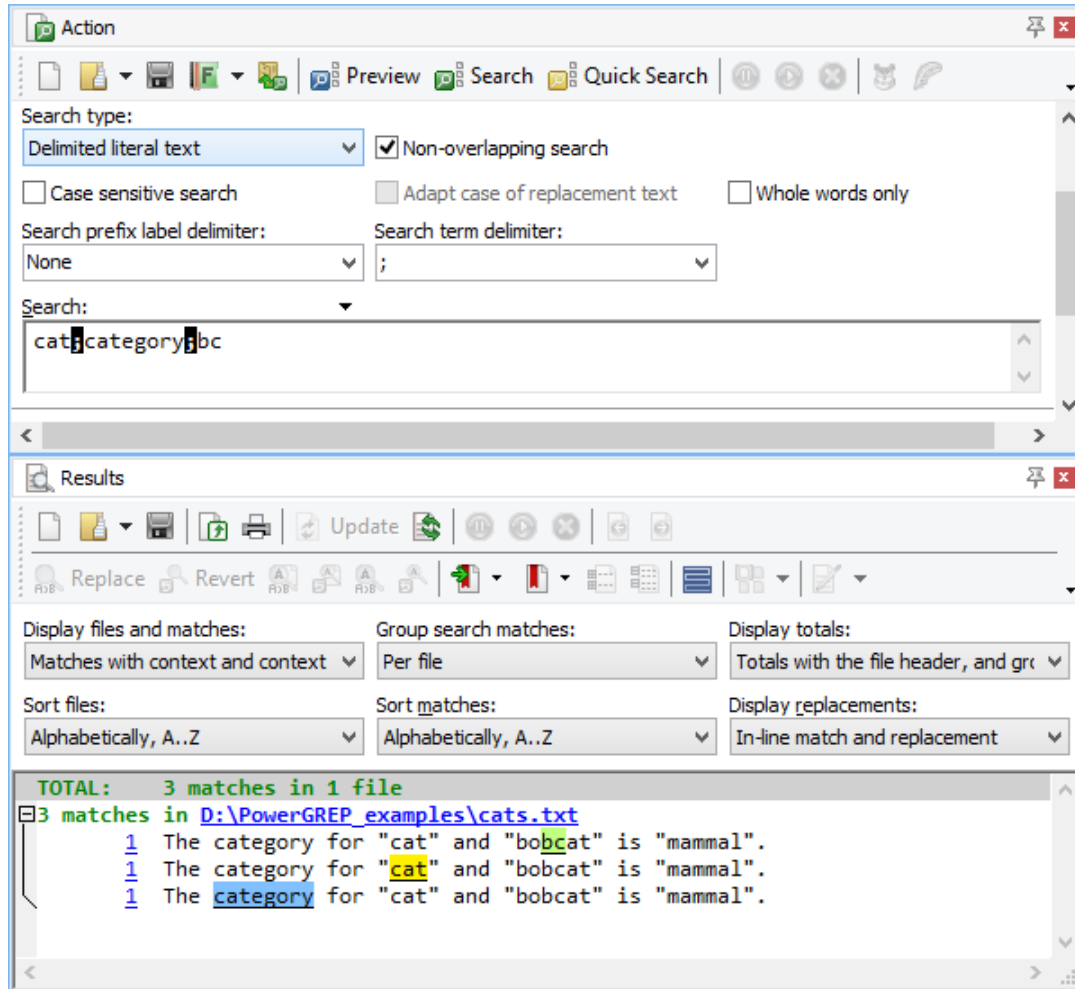
Several action parts such as “extra processing” or the main part of the action expect a replacement text or a text to be collected for each search term. When the search type is a regular expression, you can use backreferences to capturing groups to reinsert part of the regex match into the replacement or the text to be collected. Regardless of the search type you can also use match placeholders to reinsert the search match and path placeholders to insert parts of the path of the file being searched through.

Non-Overlapping Search

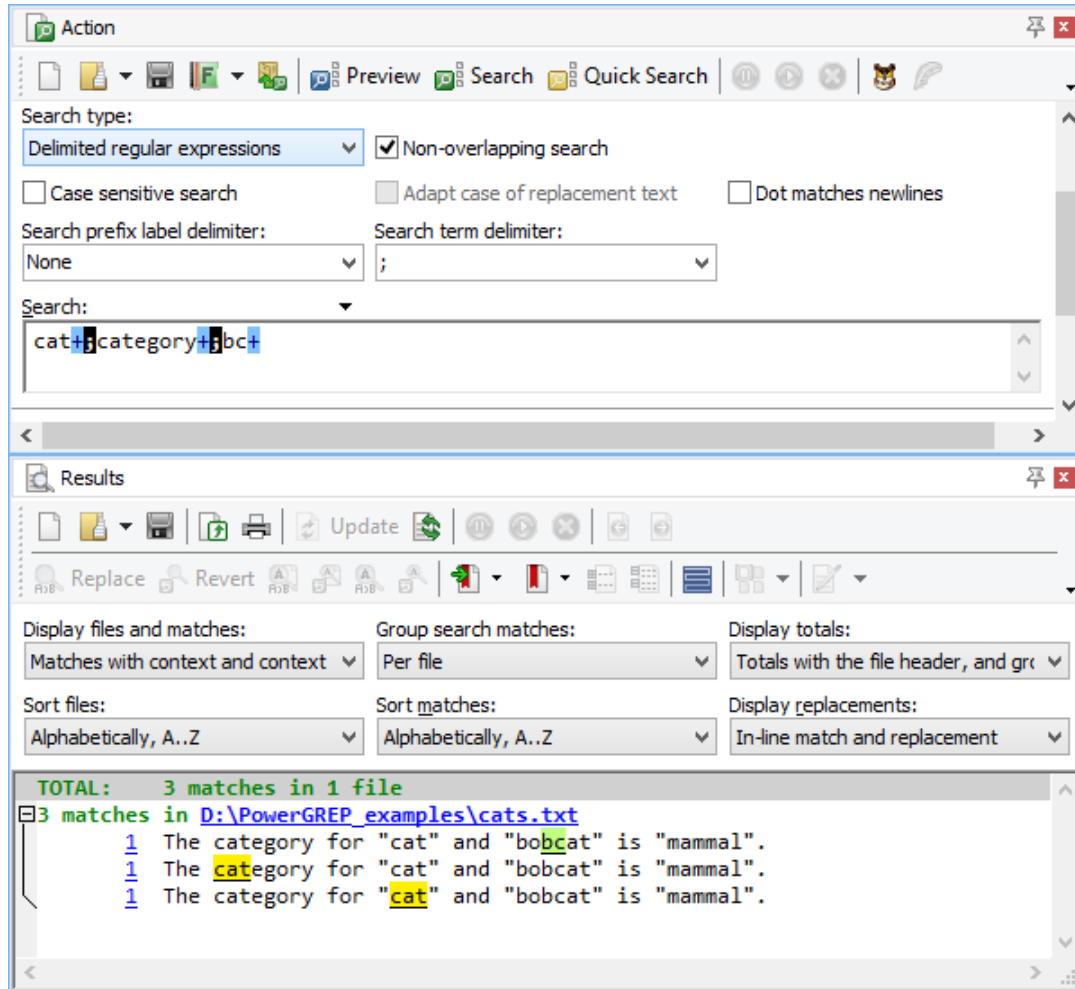
The "non-overlapping search" option is only available for search term lists, and delimited search terms. It is on by default. Turning it on or off can have a major effect on the search results.

With non-overlapping search enabled, PowerGREP will search through the text only once, looking for all search terms at the same time. Two search matches can never overlap. With non-overlapping search disabled, PowerGREP will search through the text as many times as you provided search terms. The same part of the text can be matched by more than one search term, causing those matches to overlap. Obviously, searching through the text multiple times takes longer than searching it only once.

Non-overlapping search works differently for literal text or binary data than it does for regular expressions. Suppose you are searching through one file containing the text “The category for "cat" and "bobcat" is "mammal".” You entered the literal text search terms «cat», «category» and «bc». For literal text, the order of the search terms doesn't matter. Executing this search finds each of the 3 terms once. When doing a literal text search, PowerGREP scans the file one character after the other. At each character position it evaluates the entire list of search terms. If more than one term matches, PowerGREP chooses the longest one. That is why the word “category” in our sample is matched by the search term «category» instead of «cat» even though both could be matched. When a match is found, PowerGREP continues the search after the match. Each character can be part of only one search match. That is why «cat» does match the second syllable in “bobcat” in our example. When „bc” is matched, PowerGREP continues the search starting with the “a” in “bobcat”. At that point none of the search terms can be matched in the remainder of the file.

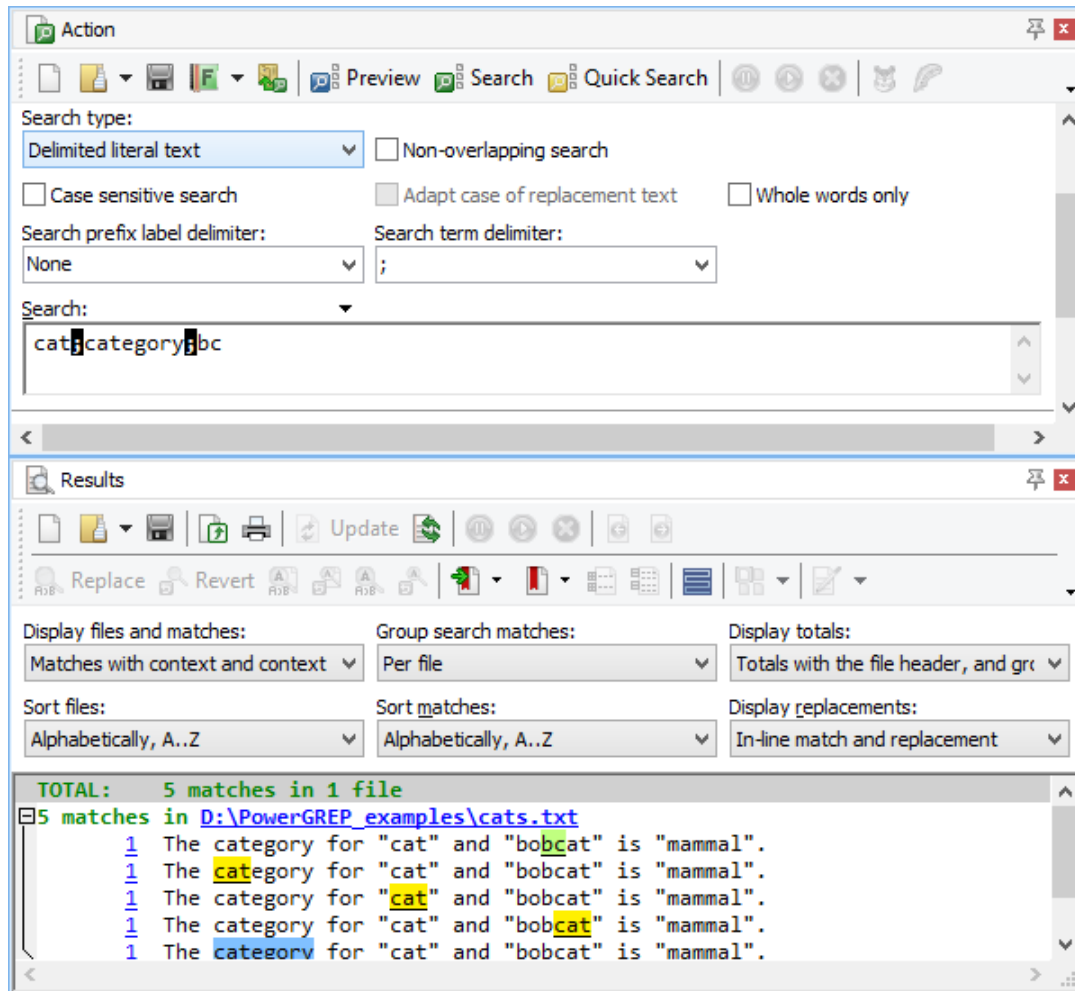


When searching for a list of regular expressions, the order of the search terms does matter. Note that merely setting the “search type” to “regular expression” is not enough to trigger this difference. If you select “regular expression” but then type in pure literal text, PowerGREP runs a literal text search. So we change our example to search for the regular expressions «cat+», «category+», and «bc+». Executing this search finds „cat” twice and „bc” once. The regex «category+» finds no matches at all. PowerGREP scans the file one character after the other as for a literal text search. But this time it tries the regular expressions one after the other. As soon as one regex matches, the match is accepted. The other regexes are not tried at the same character position, even if one of them might find a longer match. As soon as PowerGREP finds „cat” at the start of the word “category”, it accepts the match. The search then continues with “egory”, and «category+» can no longer match.



If you turn off the non-overlapping option, there is no difference between using literal text or regular expressions. PowerGREP scans the whole file once for each search term. Because each search term is handled separately, the matches of different terms can overlap. Using the same example text with either the 3 literal words or the 3 regular expressions yields the same results: „cat” is matched three times, „category” is matched once, and „bc” is also matched once. The first match of „cat” overlaps entirely with the sole match of „category” and the third match of „cat” overlaps partially with the match of „bc”.

To make all five matches clearly visible in the results, the “sort matches” option on the Results panel was set to “alphabetically”. If you set it to “original order”, PowerGREP shows the line of text only once with all five matches highlighted, making them difficult to distinguish.



In a search-and-replace action or extra processing search-and-replace, turning off non-overlapping search has an additional effect. The second search-and-replace in the list is not performed on the original text, but on the text as modified by the first search-and-replace. The third search-and-replace works on the results of the second, and so on. If the original text is “The classification for “cat” is “mammal”.”, and your first search-and-replace pair is «classification=category», and your second pair is «cat=dog», the end result will be “The dogegory for “dog” is “mammal”.”. The first iteration replaced „classification” with “category”, and the second replaced the first three letters of “category” with “dog”.

This last example executed as a non-overlapping search would yield “The category for “dog” is mammal.” After replacing „classification” with “category”, PowerGREP only searches through the remainder of the text “ for “cat” is “mammal”.”.

Though in this example, “dogegory” is not the result we wanted, in other situations the ability to have each search-and-replace pair work on the results of all previous pairs can be very useful, and result in some very powerful text processing.

Search Options

Turn on "case sensitive search" if the difference between uppercase and lowercase letters your search terms matters. When on, «cat» matches only „cat”. When off, „Cat”, „CAT” and even „cAt” are also valid matches. Case sensitive searches are faster than case insensitive ones.

Turn on "adapt case of replacement text" to automatically give the replacement text or the text to be collected the same letter casing as the search match. Suppose you are searching for «TWO cats» and replacing with “one DOG”. You have “case sensitive” turned off and “adapt case of replacement text turned” on. The PowerGREP replaces „two cats” with “one dog”, „Two cats” with “One dog”, „Two Cats” with “One Dog”, and „TWO CATS” with “ONE DOG”. PowerGREP adapts all lowercase, all uppercase, title case, and first uppercase only. A match with any other combination of uppercase and lowercase letters is replaced with the replacement text as you entered it. So „TWO cats”, „two CATS”, and „Two CaTs” are all replaced with “one DOG” as you entered it.

Turn on "whole words only" to match only complete words. With this option on, searching for «cat» does *not* match the first three letters in “category”.

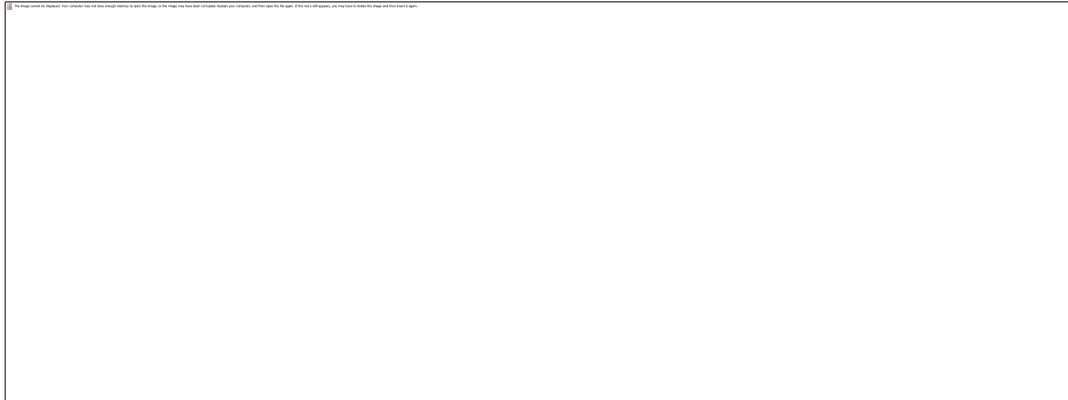
What “whole words only” really does is check if the match is not immediately preceded and not immediately followed by a character that could be part of a word. «cat» fails “category” because of the “e” immediately after the potential match. Note that your search term must be a word or phrase. If your search term does not start with a character that can occur in a word, PowerGREP will not find any matches at all when you turn on “whole words only”.

The option "dot matches newlines" controls the behavior of the dot in a regular expression. By default, the dot will match any character except the line break characters CR (carriage return), LF (line feed), VT (vertical tab) and FF (form feed). When you turn on “dot matches newlines”, the dot will match any character including line break characters.

Regex Options and Lists

When using a list of search terms, the above options apply to all search terms. When using regular expressions, you can use mode modifiers to toggle the some of the options for individual regular expressions (or even parts of regular expressions). Put «(?i)» in front of a regular expression to make it case insensitive, or «(?-i)» to make it case sensitive. Use «(?s)» to turn on “dot matches newline”, and «(?-s)» to turn it off.

8. Action Part: Filter Files

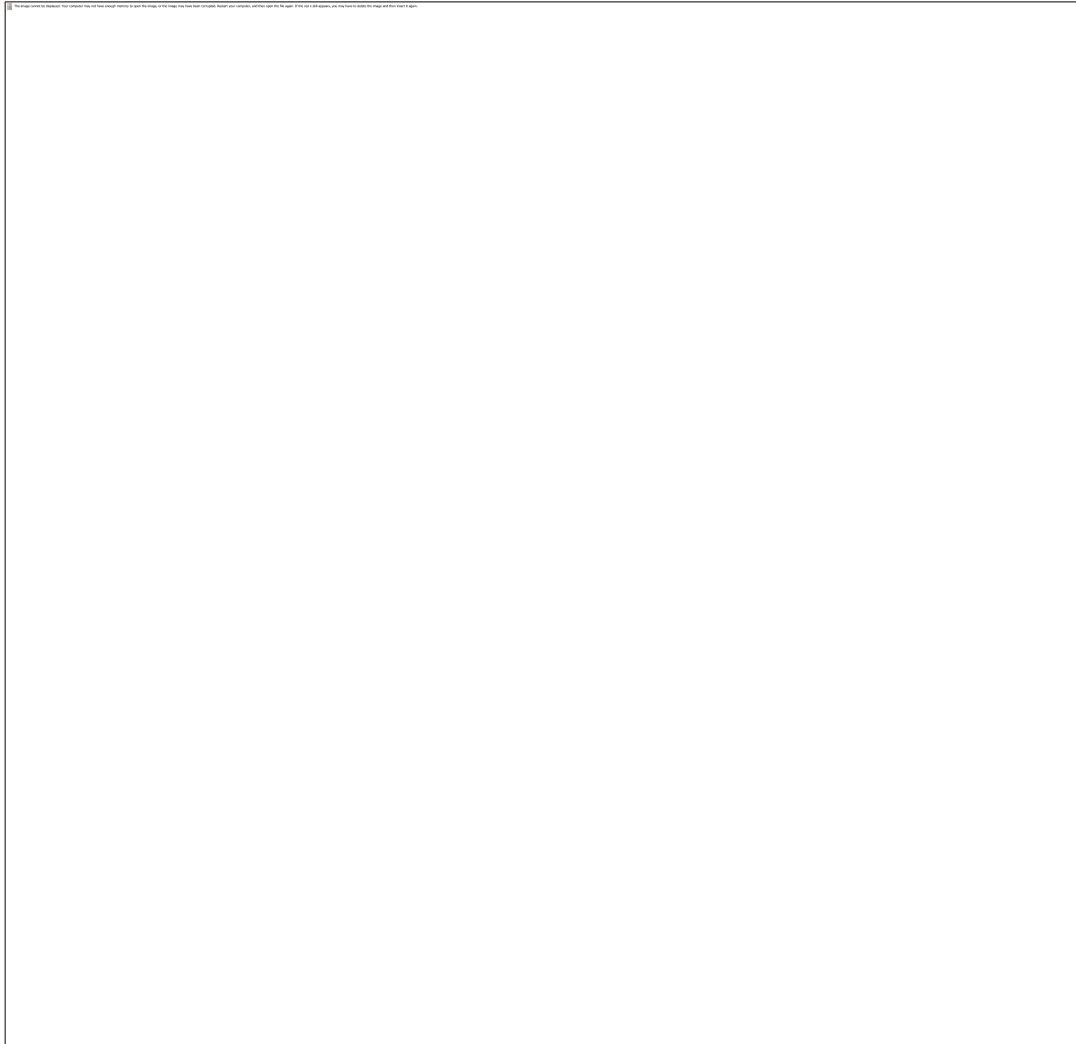


The "filter files" action part is available for all action types except "simple search". It enables you to use an extra set of search terms to filter out files. The "filter files" drop-down list gives you four choices:

- **Do not filter files:** Process all files marked in the File Selector. All the other controls of the "filter files" action part are hidden when this option is selected.
- **Disallow any terms to match:** Process only files in which none of the file filtering search terms can be matched.
- **Require one term to match:** Process only files in which at least one of the file filtering search terms can be matched.
- **Require all terms to match:** Process only files in which all of the file filtering search terms can be matched. The difference between the last two options only comes into play when filtering using a (delimited) list of multiple search terms.

There are no options to specify other than the search terms you want to filter with. Matches found by the "filter files" action part never show up on the Results panel. If a file is filtered out, it is indicated in the results as not having any search matches, regardless of whether it was filtered with "disallow any terms to match" or "require one/all terms to match". If a file is not filtered out, the results show its search matches as usual, if any are found by the main part of the action.

A And Not B



To get a list of files that contain “A” but not “B”, set the “action type” to “list files”. Specify A as the search term in the main part of the action. Set “filter files” to “disallow any terms to match” and specify B as the search term to filter with.

If you do this with the “search” action type, you’ll get a list of all the occurrences of A in all the files that do not contain B.

Example: Boolean operators “and” and “or”

Capture Text for Reuse

When using regular expressions, named capturing groups carry over from the “filter files” part of the action to all the other parts. This means you can use the “filter files” part to run an extra search to capture a part of the file. The filter doesn’t necessarily have to exclude any files.

You can even capture multiple parts of the file by using a list of regular expressions with a differently named capturing group in each regular expression. Set “filter files” to “require all terms to match” and turn off “non-overlapping search”. These two settings ensure PowerGREP searches through the entire file once using each regular expression in your list, filling the named capturing group(s) of each regular expression, provided they can all find a match in the file.

Examples: Insert proper HTML title tags and Rename files based on HTML title tags

9. Action Part: File Sectioning

With the "file sectioning" part of the action definition, you can specify which parts of each file PowerGREP should process. You can make PowerGREP search through only part of each file, or split up each file any way you want, rather than searching the whole file at once. A common choice is to process files line by line. The "file sectioning" action part is available for all action types except "rename files".

Disabling File Sectioning



The default is "do not section files". PowerGREP searches the whole file without any boundaries. Search matches can span multiple lines, or even the entire file. Not sectioning files is the fastest option.

Simple File Sectioning (Line by Line)



The "simple search" action type offers only two file sectioning options. Instead of the "file sectioning" dropdown list you get a "line by line" checkbox. Ticking the checkbox selects "line by line" file sectioning, while clearing the checkbox is the same as selecting "do not section files".

When you select "line by line", PowerGREP scans the file for line breaks. Each line is then searched through separately. The line breaks are not included in the sections. This means the search terms in the main part of the action can never match line breaks or span across lines. Traditional UNIX `grep` always applies your regular expression to one line at a time. The "line by line" option makes PowerGREP do the same.

If you tell PowerGREP to search through binary files and turn on "line by line" file sectioning then PowerGREP also scans binary files for line breaks and processes them line by line. Whether this is a good idea depends on the contents of your binary files and how many line breaks they have.

The line breaks between the lines are not part of the section when you select "line by line". They are included when you select "line by line (including line breaks)". The difference is important when replacing or collecting whole sections (see below). If the line break is included in the section, it will be deleted when the section is replaced, or included in the text to be collected.

When you tick the "line by line" checkbox, one or two additional checkboxes appear. The option "invert search results" makes the main part of the action match sections (lines) in which the search terms can *not* be found. The entire section (line) is then treated as the search match. That means that when collecting matches or replacing matches, whole sections (lines) are collected or replaced with a common replacement text.

When the action type is set to "list files", the "invert search results" option appears even when file sectioning is disabled. The reason is that for "list files" action, this option applies to the whole file rather than just the section. When you turn on "invert search results" for a "list files" action, you get a list of all files in which the search terms cannot be found.

The option "list only sections matching all terms" appears if the main part of the action has more than one search term. Turn on this option to tell PowerGREP to retain only matches from sections (lines) in which all the search terms can be found. Search matches found in sections (lines) that contain only some of the search terms are discarded. If you turn on this option for a search-and-replace action, whole sections (lines) must be replaced with a common replacement text.

Example: Find two or more words on the same line

Regular File Sectioning (Line by Line)



All other action types that support file sectioning provide additional options. The option "match whole sections only" limits search matches in the main part of the action to those that match an entire section. All other matches are skipped. E.g. when sectioning a file line by line, only search matches spanning a complete line are retained.

"Collect/replace whole sections" causes the main part of the action to act as if the entire section matched a search term, even if the search term matches only part of the section. The whole section will be returned as the search result. In a search-and-replace action, the whole section will be replaced with the replacement text.

Inverting search results has a different meaning in "list files" actions. The option inverts the whole list of files, not individual sections or lines, since "list files" does not list individual matches or sections in the search results.

Examples: Extract or delete lines matching one or more search terms, Boolean operators "and" and "or" and Split web logs by date

Other Ways to Split Files into Sections



To split a file into chunks other than single lines, use the "split along delimiters" sectioning type. The two most common situations for splitting a file into sections are files with custom record delimiters (i.e. not line breaks), and files where you *don't* want to search through part of the file.

Custom delimiters are easy. Simply enter the record delimiter the files use as the search term in the sectioning part. PowerGREP will first search for the delimiters, and then search through each section of the file (i.e. record) between delimiters, as well as the sections before the first delimiter and after the last delimiter. The delimiters themselves are never "seen" by the main part of the action.

Particularly powerful is the ability to specify which sections of the file you do *not* want to search through. E.g. if you want to process some source code, but don't want to search through comments or strings in the source code, use the "split along delimiters" sectioning type, and enter a list of regular expressions matching comments and strings in the source code. The screen shot above shows comments (steps 1 to 3) and strings (step 4) used by the Delphi programming language. The result is that PowerGREP will treat comments and strings as "delimiters", and only search through the sections of the file between comments and/or strings.

Examples: Search through or skip comments and strings, Add line numbers, Collect page numbers, Collect paragraphs (split along blank lines) and Put anchors around URLs that are not already inside a tag or anchor

To do things the other way around, i.e. specify the sections that you *do* want to search through, select the "search for sections" sectioning type. When executing the action, PowerGREP will first search for the sectioning search terms. The main part of the action is then restricted to the sections in the file matched by the sectioning search terms. Anything between the sections is ignored by the main action. E.g. the four regular expressions in the screen shot could be used to search through *only* comments and strings in Delphi source code.

The last sectioning type, "search and collect sections" is useful when you cannot easily create a regular expression that matches only the section of the file you want to search through. Though you can usually solve that problem with clever use of lookahead, collecting sections is often much easier and more straightforward.

When collecting sections, each sectioning step requires a "section collect". The "section collect" must be a backreference to a numbered or named capturing group in the sectioning regular expression. The text matched by the capturing group is the section that will be searched through. You can specify only one capturing group per sectioning step. E.g. if you set "section search" to the regex «<H[1-6]>(.*?)</H[1-6]>» and the "section collect" to the backreference «\1», the main action will process everything between heading tags in an HTML file, ignoring the heading tags themselves and everything outside heading tags.

If you leave the "section collect" empty for a particular sectioning step, that step's matches will never be searched through. This can be useful in a non-overlapping search where you want to exclude some sections.

Examples: Search through printable content in Word .docx files, Search through printable content in XPS files, Search through printable content in OpenDocument Format files, Search through or skip comments and strings, Make sections and their contents consistent and Replace HTML attributes

Testing File Sectioning

You can easily test the file sectioning settings by running a dummy search. Set the action type to “search”. Enter the regular expression «.++» as the search term and turn on “dot matches newline”. This regex will match each section entirely and display it in the results.

How Sectioning Affects The Main Search

When you don't section files, the main part of the action searches through the entire file. When you do section files, the main part of the action searches through the sections *only*. The main part of the action cannot “see” outside of the sections. This doesn't matter when searching for literal text or binary data, but it does matter when searching using regular expressions.

As far as the regular expression engine is concerned, when it searches through a section, that section is all that exists. The start-of-file anchor «\A» and the end-of-file anchor «\Z» will match at the start and the end of every section. Lookaround will not be able to “see” beyond the section.

Sectioning and Overlapping Search

As described in the chapter discussing search terms, when the search terms consist of multiple items, an option "non-overlapping search" becomes available. What follows assumes you have already understood the implications of overlapping and non-overlapping searches described there.

This option is enabled by default. PowerGREP divides the file into sections only once and sections never overlap. In most situations, a non-overlapping search is what you need. E.g. when sectioning along comments and strings in a programming language, you want to ignore comment characters inside strings, and quote characters inside comments. A non-overlapping search automatically takes care of that.

When you turn off “non-overlapping search”, PowerGREP will section the file as many times as you provided sectioning search terms. The main action is run entirely on all the sections found by the first sectioning step, before PowerGREP continues with the second sectioning step.

This means that in a search-and-replace action, which modifies the file being searched through, the second sectioning step will process the file after all the sections found by the first sectioning step have been searched-and-replaced through completely. This means the second sectioning step may find sections differently than it would when processing the original file. Depending on what you're doing, and whether you took this into account, this cascade effect may produce desirable or undesirable results. This applies even when the target types is set to make a copy of the file, and even when previewing the action. PowerGREP will modify the working copy of the file, regardless what happens with it in the end.

Named Capturing Groups Carry Over

When using regular expressions, named capturing groups carry over from the file sectioning to both the main part of the action and the extra processing part. If the sectioning regex uses a capturing group, you can use a backreference to that capturing group in the regular expression and/or the replacement text of the main action and/or the extra processing.

10. Main Part of The Action

The Action panel is the place where you define the task that PowerGREP will execute. All the options on the Action panel are arranged into nine parts, though not all those parts are always visible. This screen shot shows all nine:

The screenshot displays the 'Action' panel in PowerGREP, which is used to define search and collection tasks. The panel is organized into nine distinct sections:

- Action type:** Set to 'Collect data'. Includes checkboxes for 'List only files matching all terms' and 'Group identical matches'.
- Filter files:** Set to 'Do not filter files'.
- File sectioning:** Set to 'Do not section files'.
- Search type:** Set to 'Regular expression'. Includes a checked 'Non-overlapping search' checkbox and unchecked checkboxes for 'Case sensitive search', 'Adapt case of replacement text', and 'Dot matches newlines'.
- Search:** A text field containing the regular expression 'main·search·term'.
- Collect:** An empty text field for defining collection patterns.
- Extra processing:** An unchecked checkbox for 'Perform a search and replace on the replacement text or collect text'.
- Context type:** Set to 'No context'.
- Between collected text:** Set to 'Line break'. Includes an unchecked checkbox for 'Collect headers and footers'.
- Target file creation:** Set to 'Save results into a single file'.
- Target file destination type:** Set to 'Single folder'.
- Target file location:** An empty text field with a browse button ('...').
- Target file text encoding:** Set to 'Same as original file'.
- Target file line break style:** Set to 'Same as original file'.
- Order of collected matches:** Set to 'No particular order'.
- Backup file naming style:** Set to 'Multi "Backup N of ..."'.
- Backup file destination type:** Set to 'Same folder as original'.
- Backup file location:** An empty text field with a browse button ('...').
- Comments:** A text area containing the comment 'Action panel showing all nine parts.'

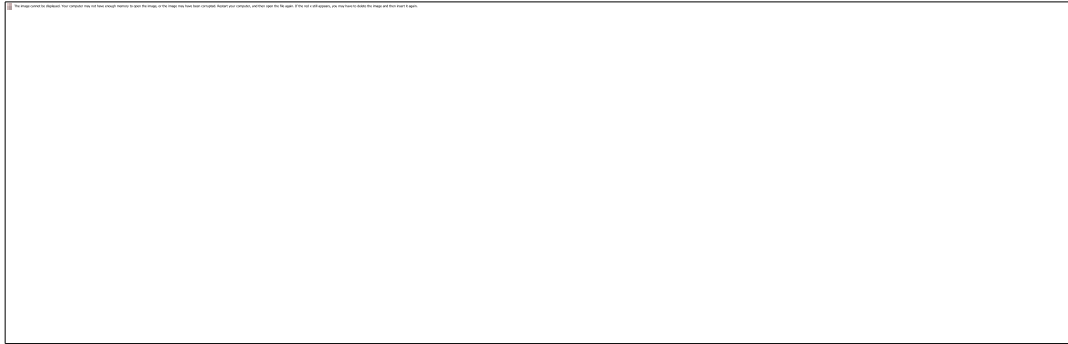
The main part of the action is always visible, and always shows a Search box to enter search terms. In the screen shot, the main part of the action is the only part that shows the Search box. All of the other parts are turned off.

What PowerGREP does with the main search term of your action depends on the “action type” that you’ve selected at the top of the Action panel. All action types except “list files” and “merge files” need at least one search term in the main part of the action.

The action type also determines if the main part of the action shows a second box to enter a replacement text or a text to be collected. The “search-and-replace” action type needs a replacement text to replace search matches with. The collect data action type needs a text to be collected.

Exactly what PowerGREP does with the search term(s) in the main part of the action is described in detail in the topic about action types. Normally, the main search terms are used to search through the contents of the files. The “file name search” and “rename files” action types use them to search through the names of files instead.

11. Action Part: Extra Processing



The “extra processing” part of the Action panel is only visible when you’ve set the action type to “search-and-replace”, rename files, or “collect data”. When you mark the “extra processing” checkbox, an extra set of controls for entering search terms appears.

“Extra processing” is simply a fancy name for an additional search-and-replace. This search-and-replace is not run on a file, but on each replacement text in a search-and-replace action, or on each text to be collected in a “collect data” action. It is most useful when the main action searches using regular expressions, and replaces or collects text using backreferences. The extra processing step is run after backreferences have already been substituted in the replacement text or text to be collected, giving you a chance to reformat them.

An example: when collecting data from URLs in the log files of a web server, you’ll get back URL-encoded data. E.g. spaces appear as plus symbols, and plus symbols appear as %2B. With an extra processing step, you can search-and-replace the plus symbols back into spaces, etc. making the results a lot more readable.

Do not confuse extra processing with entering a list of search terms in the main part of a search-and-replace action. Each search-and-replace in the main part of the action is run on the entire file, or the entire section. The extra processing is only run on the replacement text, just before the main action makes a substitution. If the main action uses a list, the extra processing is applied to each replacement made by all items in that list.

Examples: Search through printable content in Word .docx files, Padding replacements, Rename files based on HTML title tags, Collect XML Data with entities replaced, Convert Windows to UNIX paths and Extract Google search terms from web logs

Named Capturing Groups Carry Over

When using regular expressions, named capturing groups carry over from the file sectioning and the main part of the action to extra processing. If the sectioning regex or main regex used a capturing group, you can use a backreference to that capturing group in the regular expression and/or the replacement text used for extra processing.

12. Action Part: Context

When you preview or execute an action (but not when you use Quick Execute), PowerGREP displays the search matches it found on the Results panel. To make it easier to distinguish between all the search matches the Results panel has options to display search matches along with their context rather than just the search matches themselves. This option only works if you use asked PowerGREP to collect additional context around each search match while executing the action.

The “list files” and “merge files” action types do not allow context to be collected, because these two action types never display any search matches on the Results panel. They list file names only. The “rename files” action types always collects the file’s full path as context. So for these three action types the “context” part of the Action panel is invisible.

The “simple search”, “search”, and “collect data” action types all allow context to be collected, unless you’ve turned on the option to group identical matches. When grouping identical matches each unique search match is stored only once, regardless of how often it occurs in the file(s) you’re searching through. Since identical matches may have different context, there’s no way for PowerGREP to collect that context when grouping identical matches.

Disabling Context



The default is "no context". This is the fastest option because PowerGREP doesn’t have to spend time locating the context or any memory storing it. The Results panel won’t show any context.

Note that you don’t need to select “no context” when using Quick Execute. Since that command tells PowerGREP not to display any search matches on the Results panel, it also disables context automatically, regardless of which context options you selected on the Action panel.

Simple Context (Lines as Context)



The “simple search” action type offers only two context options. Instead of the “context type” drop-down list you get a "use lines as context" checkbox. Ticking the checkbox is the same as selecting “use lines as context” in the “context type” drop-down list and turning on “show line numbers”. Clearing the checkbox is the same as selecting “no context”.

Extra Context Before The Match

Number of extra blocks or lines of context you want before the match.

If expand to whole lines is checked, extra lines of context are added. Otherwise, extra blocks according to the context type are added.

Extra Context After The Match

Number of extra blocks or lines of context you want after the match.

If expand to whole lines is checked, extra lines of context are added. Otherwise, extra blocks according to the context type are added.

Regular Context (Lines as Context)

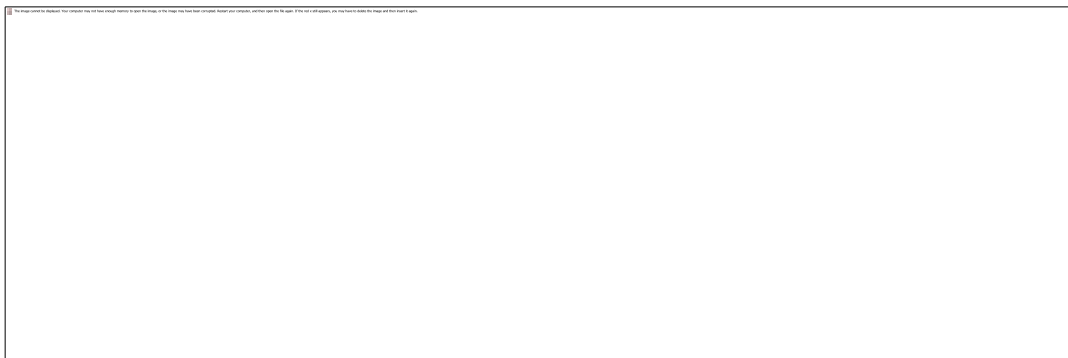


All the other action types that support context provide additional context types. Select “use lines as context” to get the same context options as for simple searches, plus one more:

Show Line Numbers

Turn on to scan the file for lines and show line numbers with each line of context or with each match. Turn off to show sequential context numbers.

All Context Types



Even more context options are available if you choose another context type.

Context Type

When displaying search matches on the Results panel, PowerGREP can display extra context around each search match. This context is the text that appears in the file immediately before and after the search match. This setting determines how much context PowerGREP will collect from the file, if any.

Context is only used on the Results panel. It is never saved to target files.

Collecting context slows down the search and takes up extra memory, but makes it easier to inspect the search matches on the Results panel. If you don't collect context, you need to double-click on a search match in the results to open the file it was found in on the Editor panel in order to see its context.

- **No context:** Do not collect any context. Only the search matches themselves will be shown in the results.
- **Use sections as context:** When using file sectioning, you can use the section that the matches were found in as context. This is the way PowerGREP 3 used to work, which had no separate context settings.
- **Use lines as context:** Scan the file for line breaks, and use line a search match was on for context. If the search match spans multiple lines, the lines it starts and ends on are both used for context.
- **Fixed-length records:** Split the file in chunks of equal length. Use this for unstructured binary files and for files that consist of fixed-length records.
- **Split along delimiters:** Split the file at each match of the context search term(s). The text between the main search match and the context delimiters and precede and follow the main search match is used for context.
- **Search for context:** Use each match of the context search term(s) as a block of context. If a main search match falls entirely within a context search match, that context match is used for context. If not, the context search matches in which the main search match begins and ends are used for context. If the main and context searches do not always produce overlapping matches, some matches will be displayed without context.
- **Search and collect context:** As above, but with the ability to specify a backreference in the context collect box to use only part of the context regex match as context. This requires the context search to be a regular expression.

Count All Context

Turn on to count all context blocks in the file, including context that doesn't appear in the results. Turn off to show sequential context numbers in the results, ignoring any unused context.

Expand to Whole Lines

Turn on to expand the context so that all matches are always shown with full lines of context. This makes it easier to interpret the results if you're used to working with the files with a line-based application such as a plain text editor. Turn off to allow the context to be a partial line.

13. Action Part: Collect Between

The “search”, “collect data”, and “merge files” action types save search matches to one or more target files, at least when “target file creation” is set to anything except “do not save results to file”. If an action finds three matches „1”, „2”, and „3” you may want to save them as “1, 2, 3” into the target file rather than as “123”. The “collect between” part of the Action panel is where you can tell PowerGREP what, if anything, PowerGREP should place between the collected matches. In this example we’re putting a comma between all the matches.

In addition or instead of putting text between matches, you can add a header and footer to each target file created by PowerGREP. When collecting matches from multiple source files (files that were searched through) into a single target file, you can also add a header and footer to the blocks of matches that were found in the same source file. You can use path placeholders to add references to the files the matches were found in.

The appearance of the “collect between” part of the action changes based on what you select in the “between collected text” drop-down list and whether you tick “collect headers and footers”. If you’re not using any options that require custom text, no text box appears:

 A screenshot of the 'collect between' section in the PowerGREP interface. It shows a single, empty rectangular text box with a thin black border, indicating that no custom text is required for this configuration.

If you’re using only one delimiter between the matches without any headers or footers, you get one box to type in that delimiter:

 A screenshot of the 'collect between' section in the PowerGREP interface. It shows a single, empty rectangular text box with a thin black border, intended for entering a single delimiter character or string.

If you’re using multiple delimiters or headers and footers then a list with different items appears. All the items in the list are used when you execute the action. The item that you select is the one that you can edit in the text box:

 A screenshot of the 'collect between' section in the PowerGREP interface. It shows a list of options (not clearly legible) and a text box below it for editing the selected item. The text box is currently empty.

Between Collected Text

Choose which text, if any, you want to place between text that was collected for different search matches:

- **Nothing:** Don’t put anything between text collected for different matches.

- **Line break:** Use a separate line to collect the text for each search match.
- **Text between matches and files:** Put the same text between all matches, regardless of which files they were collected from.
- **Text between matches only:** Put text between the matches collected from the same file. Do not put any text between matches collected from different files (other than file headers and footers).
- **Text between files only:** Do not put any text between matches collected from the same file. When saving text from multiple files into a single file, put text between matches collected from different files.
- **Different text between matches and files:** Put one bit of text between text collected for matches from the same file, and another bit of text between matches collected from different files. The text to be put between matches from different files is only used when saving text from multiple files into a single file.

Collect Headers and Footers

Add a header and footer to the files into which the collected text is saved. When collecting into a single file, you can also add a header and footer to the block of text collected from each file searched through.

Select Text to Edit

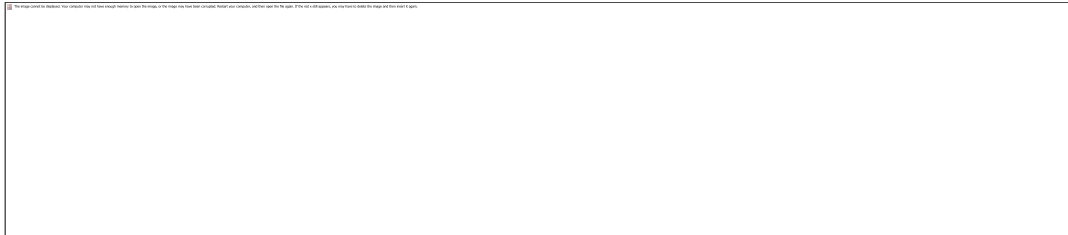
Select the text you want to edit in the edit box to the right of this list. Which matches are available depends on the choices you make for “between collected text”, “collect headers and footers”, and “target file creation”.

- **Between matches:** Text to be put between collected matches, whether they’re from the same file or from different source files.
- **Between matches from the same file:** Text to be put between matches collected from a single source file.
- **Between matches from different files:** Text to be put between the blocks of matches collected from different source files, when they are being saved into a single target file.
- **Source file header and footer:** Text to be put before and after a block of text collected from a single file. Can use path placeholders to insert the path to the source file. Used when creating a separate target file for each file, as well as when collecting text from multiple sources into a single target file.
- **Target file header and footer:** Text to be put at the start and the end of each file created by the collect data action. Only used when collecting text from multiple source files into a single target file.

Examples: Compile indices of files and Generate a PHP navigation bar

14. Action Part: Target and Backup Files

Near the bottom of the Action panel, you can select how PowerGREP should save, modify or copy files while you execute the action. To run an action without modifying any files at all, simply use the Preview button. Previewing an action will never modify any files or do anything else you might regret.



Target Files

Target File Creation

The available target types depend on the kind of action you have prepared. For a “list files” or file name search action, five target types are available:

- **Do not save results to file:** The results are simply listed on the Results panel. No target and backup options need to be specified.
- **Save list of matching files to file:** The list of files displayed on the Results panel is saved into a text file of your choice. You can choose the encoding and line break style of the file, as well as the delimiter put between each file name. Specify a single file as the target location.
- **Copy matching files:** Files listed in the results are copied into a folder or archive of your choice. You can use the target file destination type to compress or decompress files, leaving the originals.
- **Move matching files:** Files listed in the results are copied into a folder or archive, and then removed from the original location. You can use the target file destination type to compress or decompress files, removing the originals.
- **Delete matching files:** Files listed in the results are permanently deleted. This action cannot be undone. You will not be asked for backup options. Be careful!

A “search-and-replace” or “search-and-delete” action offers three target types:

- **Modify original files:** Make the replacements in the files searched through.
- **Copy only modified files:** If a file has one or more search matches, make a copy of it and modify the copy.
- **Copy all searched files:** Make a copy of all files searched through, and modify the copy if the file has one or more search matches.

Five target types are available for “search” and “collect data” actions:

- **Do not save results to file:** Display the results in PowerGREP only. No target and backup options need to be specified.

- **Save results into a single file:** Save the text collected from all the files into a single new file. Specify a single file as the target location.
- **Modify original files:** Save the text collected in each file to the file, overwriting the original file. The end result is a search and replace that deletes unmatched parts of the file.
- **Save one file for each searched file:** Create a new file for each file that has one or more search matches. Save the text collected from the file into the new file.
- **Path placeholders:** Use path placeholders to dynamically build a full target path for each file searched through. If the placeholders result in the same target path for multiple files searched through, the text collected from all those files will be combined into the target file.

When you set the action type to “rename files” you can choose to leave the original files in place when renaming:

- **Rename or move files:** Rename or move each file to the path created by searching-and-replacing through the file’s path.
- **Copy files:** Copy each file to the path created by searching-and-replacing through the file’s path.

The “merge files” action type offers three choices:

- **Merge into a single file:** Merge all the files into a single new file.
- **Merge based on search matches:** Search through the files to be merged and determine the target file to merge each file into based on the search match in each file. You can specify the target in the main action, like you would specify the replacement text in a search-and-replace action.
- **Path placeholders:** Use path placeholders to dynamically build a full target file to merge each file into.

Target File Destination Type

Select the type of place you want the target files to be created. Except when using path placeholders, target files will have the same name as the original files.

- **Single folder:** Place all files into a single folder.
- **Folder tree:** Place all files into a specific folder. If you marked a folder to also include its subfolders, those subfolders will be recreated under the target folder.
- **Compressed archive:** Place all files into a single .zip or .7z archive. If you marked a folder to also include its subfolders, those subfolders will be recreated inside the archive. If the archive already exists, the files will be added to it. If the archive already contains a file with the same name, a backup copy of that file is created.
- **Numbered archive:** Similar to the “compressed archive” option, except that if the archive already exists, a new archive will be created with a number added to its name.
- **Path placeholders:** Use path placeholders to dynamically build a full target path for each file searched through. You should take care to use a unique target path for each file. Otherwise the targets will overwrite each other.

Target File Location

Specify a target location of the type selected in the target file destination type list. Click the (...) button to interactively select or build the target location.

Target File Text Encoding

Select the Unicode encoding or the Windows code page the target file(s) should use. Choose “same as original file” if unsure.

Target File Line Break Style

Select the character sequence to be used to separate lines in the target file. Choose “same as original file” if unsure.

Order of Collected Matches

When collecting data into a single file without both grouping identical matches and grouping results for all files, this option determines in which order the collected matches are saved into that file.

- **No particular order:** Collect items as search matches are found. This is the fastest option. Since PowerGREP searches multiple files concurrently, text collected from one file may be mixed with text collected from other files. If you set “between collected text” to put text between matches from different files, or you specify source file headers and footers, PowerGREP will keep each file’s matches together even if you specify “no particular order”. This makes sure the headers, footers, and text between files appear logically.
- **Keep each file’s matches together:** Collect matches as each file is processed. Text collected from one file will never be mixed with text collected from any other files. Since PowerGREP searches multiple files and drives concurrently, the blocks of text collected from different files are not ordered in relation to those files.
- **Sort files alphabetically, A..Z:** Keep each file’s matches together. Create the target file after all files have been processed, collecting text in alphabetical order of the files.
- **Sort files alphabetically, Z..A:** The previous option, in reverse.
- **Sort files by increasing totals:** Keep each file’s matches together. Create the target file after all files have been processed, starting with the file with the fewest search matches until the file with the most search matches.
- **Sort files by decreasing totals:** The previous option, in reverse.
- **Newest file to oldest file:** Keep each file’s matches together. Create the target file after all files have been processed, starting with the file that was modified the shortest time ago until the file that was modified the longest time ago.
- **Oldest file to newest file:** The previous option, in reverse.

Backup Files

Whenever you specify a target type that may cause files to be overwritten, you should specify how PowerGREP should create backup copies of files that are overwritten. While you can tell PowerGREP not to create backup copies at all, this is not recommended. PowerGREP needs the backup copies to be able to undo the action in the Undo History. If no backup files are created, PowerGREP will not add the action to the Undo History at all.

Backup File Naming Style

When a target file already exists, PowerGREP can make a backup copy of the file before overwriting it. The name of the backup copy is based on the name of the target file.

- **No backups:** Do not create backups at all. You will not be able to undo the action if it overwrites files.
- **Single *.bak:** Append a ".bak" extension. The backup of FileName.ext is FileName.ext.bak. If the backup file already exists, it will be overwritten. This option gives backup files a unique file type in Windows Explorer.
- **Single *.~*:** Insert a tilde into the file's extension. The backup of FileName.ext is FileName.~ext. If the backup file already exists, it will be overwritten.
- **Multi .bak, .bak2, ...:** Append a ".bak" extension. If the backup file already exists, append a number to the extension to make the file name unique.
- **Multi "Backup N of ...":** Prepend "Backup 1 of " to the file's name. The backup of FileName.ext is "Backup 1 of FileName.ext". If the file already exists, the number is incremented to make the file name unique. This option preserves the extension, making it easy to open backup files in the original application.
- **Same file name:** Use the same file name for the backup as the original. This option requires you to place backup files into a separate folder or .zip archive. If the backup file already exists, it will be overwritten.
- **Path placeholders:** Use path placeholders to dynamically build a full backup path for each file that needs to be backed up.
- **Hidden history:** Store backup files in a hidden __history subfolder of each folder in which files are overwritten. This option makes sure backup files don't clutter up your folders.

Backup File Destination Type

Select the type of place you want the backup files to be created.

- **Same folder as original:** Place the backup file in the same folder as the file it is a backup of.
- **Subfolder of original folder:** Place the backup file in a subfolder of the folder that holds the file that it is a backup of.
- **Single folder:** Place all backup files into a single folder.
- **Folder tree:** Place all backup files into a specific folder. If you marked a folder in the file selection to also include its subfolders, those subfolders will be recreated under the target folder.
- **Compressed archive:** Place all backup files into a .zip or .7z archive. If you marked a folder to also include its subfolders, those subfolders will be recreated inside the .zip archive. If the archive already exists, the files will be added to it.
- **Numbered archive:** Similar to the "compressed archive" option, except that if the archive already exists, a new archive will be created with a number added to its name.

Backup File Location

Specify a target location of the type selected in the backup file destination type list. Click the (...) button to interactively select or build the backup location.

15. Action Parts and Named Capture

If you have some experience with regular expressions, you've certainly come across or even created regular expressions that use capturing groups and backreferences. The regular expression «(one)(two)(three)» matches the text „onetwothree”. If we pair the replacement text \3\2\1 with this regular expression then the actual replacement becomes “threetwoone”. A more useful example might be the regular expression «\b(\d\d)/(\d\d)/(\d\d\d\d)\b» to match a date in dd/mm/yyyy or mm/dd/yyyy format and the replacement text “\2/\1/\3” to flip the day and month numbers.

Some regular expression flavors, including the JGsoft flavor used in PowerGREP, added a new syntax for named capturing group. The only difference between a named capturing group and a traditional numbered one is that you can use a chosen name to reference the group instead of a number that requires you to count how many groups there are in your regular expression. It simply makes your regular expression easier to read and to maintain. The date regex could be written as «\b(?:'day'\d\d)/(?:'month'\d\d)/(?:'year'\d\d\d\d)\b» and the replacement text as “\${month}/\${day}/\${year}”.

PowerGREP takes named capturing groups a step further. Normally, capturing groups can only be referenced by a single regular expression and replacement text. In PowerGREP, named capturing groups are shared between all the regular expressions on the Action panel. Text captured by a named capturing group is preserved until PowerGREP either attempts to match the same regular expression again or PowerGREP attempts to match another regular expression that defines the same capturing group or PowerGREP proceeds with the next file. As long as a capturing group is preserved it can be referenced by backreferences in any other regular expression or replacement text.

PowerGREP uses the regular expressions from the various parts of the Action panel in this order:

1. The “filter files” regular expressions are attempted once to check if they can be matched or not.
2. PowerGREP finds the first match of the “file sectioning” regular expressions. If you don't use file sectioning, the remainder of this list is executed only once using the whole file as a single section.
3. PowerGREP finds the first match of the regexes in the main part of the action restricting its search to the section found in step 2.
4. PowerGREP builds the replacement text or text to be collected for the search match found in step 3.
5. If “extra processing” is used, PowerGREP runs a search-and-replace through the replacement text from step 4.
6. If PowerGREP needs to collect context it does so by applying the context regular expressions as many times as needed, starting from the start of the file or where it last stopped looking for context.
7. If step 3 found a match before the end of the section, PowerGREP goes back to step 3 to search through the remainder of the section.
8. If step 2 found a section before the end of the file, PowerGREP goes back to step 2 to find the next section.

If any of these action parts use a list of regular expressions, the “non-overlapping search” option comes into play. If this option is on, the list of regular expressions for that action part is treated as a single regular expression. Thus each match attempt of that part of the action clears all the named capturing groups defined in any of those regular expressions. If “non-overlapping search” is off then only one regular expression is attempted at a time. Each regex only clears its own named capturing groups. PowerGREP starts with the first regex in the list. It only proceeds with the next one after the previous one cannot find any more matches. This is why you need to turn off “non-overlapping search” when using the “filter files” feature to grab multiple parts of the file to be reused in the remainder of the action.

16. Action Menu

The Action menu lists commands for use with the Action panel. See the Action reference chapter for more information on the Action panel itself.



Clear

Clears all settings on the Action panel. Clearing the action reduces clutter, which makes it easier to start with a completely new action definition.



Open

Loads the file selection and action definition from a PowerGREP action file that you previously saved. Both the current file selection and action will be replaced with those saved in the file. PowerGREP results files also contain file selection information. If you select a results file, only the file selection and action information will be read from the file.

You can quickly reopen a recently opened or saved action file by clicking the downward pointing arrow next to the Open button on the Action toolbar. Or, you can click the right-pointing arrow next to the Open item in the Action menu. A new menu listing the last 16 opened or saved files will appear. Select “Maintain List” to access the last 100 files.



Save

Save the current file selection into a PowerGREP file selection file. You will be prompted for the file name each time.

All settings you made in both the File Selector and the Action panel will be saved. If you want to save the action definition only, without the file selection, consider adding the action to a PowerGREP Library instead.

Action files are appropriate for actions that you execute repeatedly, in exactly the same way. Action files can be executed from the command line. You can even generate them with other applications. Use libraries to store boilerplate action definitions for later adaptation.



Favorites

If you often open the same files, you should add them to your favorites for quick access. Before you can do so, you need to save the action to a file. PowerGREP’s window caption will then indicate the name of the action file. Click the downward pointing arrow next to the Favorites button on the Action toolbar, or the right-pointing arrow next to the Favorites item in the Action menu. Then select “Add Current Action” to add the current action file to the favorites. Pick a file from the menu to open it.

If you click the Favorites button or menu item directly, a window will pop up where you can organize your action favorites. If you have many favorites, you can organize them in folders for easier reference later.



Add to Library

Adds the current action definition to the PowerGREP Library. Unlike saving an action, which saves both the action definition and file selection, only the action definition itself is added to the library. You don't need to save the action before adding it to the library. A copy of the entire action definition is stored in the library file itself.

Only valid action definitions can be added to libraries. If something is amiss, error message will appear. Correct the error, and try adding the action to the library again.



Preview

Click the Preview button on the Action toolbar, or press F9 on the keyboard, to execute the action without creating or modifying any files. Previewing an action is always perfectly safe. It will never do anything that you might regret later.

You should make a habit of using the Preview button rather than the Execute button, even when displaying search matches or when you set the target type to “do not save results to file”. In those situations, the Preview and Execute buttons do exactly the same. Still, you should use the Preview button, just in case you made a mistake when preparing the action you are about to execute.

That said, as long as you tell PowerGREP to keep backup copies, any action, except deleting files, can be undone.

When you preview an action, PowerGREP will show detailed search results on the Results panel.



Execute

The Execute item in the Action menu executes the action for real. If the target type calls for files to be created or modified, executing the action will do so. PowerGREP will show detailed search results on the Results panel.

On the Action toolbar, the Execute button is not labeled “Execute”. Instead, it is labeled “Search”, “Copy Files”, “Move Files”, “Delete Files”, “Replace” or “Collect” depending on the action type and target type you've chosen. The “Search” label is used for all actions that will *not* modify any files. All other labels indicate that the action *will* create and/or modify files.

You can speed up executing an action for real after you've previewed it by turning on the option to search only through files with results. If you know none of the files were modified since you did the preview, turn on this option so PowerGREP doesn't needlessly search files without matches again.



Quick Execute

The Quick Execute item in the Action menu executes the action for real, without keeping individual match results. The Results panel will only show how many matches were found in each file. Files will be created or modified according to the target type.

Just like the Execute button, the Quick Execute button on the Action toolbar changes its label depending on the action type and target type you've chosen. The "Quick Search" label is used for all actions that will *not* modify any files. All other labels indicate that the action *will* create and/or modify files.

Quick Execute is significantly faster than Execute or Preview, and will use far less of your computer's memory. That's because it doesn't have to keep track of each individual match to be able to show you all the details on the Results panel. If you don't plan to inspect the search results, Quick Execute is the way to go.

When preparing a new action that you plan to execute on a large number of files, or some very large files, you should first preview the action on just a couple of the files. When you're confident the action works the way it should, expand the file selection to all the files, and use Quick Execute to execute it for real.



Pause

Pauses the action or sequence that is being previewed or executed. The Results panel shows the portion of the results that have already been collected. Pausing an action can be useful if PowerGREP is running an action that is taking a long time to complete and is slowing down your computer. If you need your computer to do something else, you can pause PowerGREP, do the other thing, and then resume PowerGREP.

You cannot close PowerGREP while an action is paused. Pausing an action is instant and leaves everything in a suspended state. PowerGREP keeps its locks on files if it had any files open that it was searching through or writing results to. If you want to close PowerGREP, either resume the action and wait for it to complete, or abort the action.

You don't need to pause an action if you merely want to pause the Results panel to get a better look of the results collected so far. To pause the results display without pausing the action itself, turn off the Automatic Update option in the Results menu. Then you can look at the results PowerGREP gathered so far while allowing it to continue to find the rest in the background.



Resume

Resumes a paused action or sequence.



Abort

Aborts the action or sequence that is being previewed or executed. The Results panel shows the portion of the results that had already been collected. Aborting the action does *not* automatically undo its effects. Files

that had already been modified will not be reverted. Files that had already been created will not be deleted. The partially executed action will be added to the Undo History, where you can undo its effects.

If an action doesn't seem to be doing what you intended, click the Abort button, inspect the results gathered so far, undo the action's effects, correct the action definition, and execute it again.

17. Sequence Reference

The Sequence panel makes it possible to execute multiple related or unrelated actions in a single sequence. PowerGREP sequences are mainly useful if you regularly run the same same set of PowerGREP actions. If you're going to execute a particular set of actions only once, it is usually easier to run them one after the other using the File Selector and Action panels. The Sequence panel does not offer any functionality that the File Selector and Action panels don't offer, other than the ability to easily run the same sequence of actions more than once.



Use a Single Action Whenever Possible

PowerGREP actions can be very powerful. The Action panel provides a ton of options. A single action can use up to five sets of search terms for specific purposes. Each set of search terms can be a list of any number of regular expressions. If you're used to working with a much more limited grep tool, you may not realize how much you can do with just one PowerGREP action, and you may make things too complicated by using the Sequence panel when it's not needed.

As a rule of thumb, you only need to use the Sequence panel to run multiple actions if those actions need to work on different sets of files, or on different different subsets of a set of files. The Action panel always uses the files marked in the File Selector. The Sequence panel can associate a different file selection with each of the actions in the sequence.

Suppose you want to replace A with B on drive C: and replace X with Y on drive D:. When doing this just once, you can mark drive C: in the File Selector and then use the Action panel to replace A with B. When that's done, mark drive D: in the File Selector and then replace X with Y on the Action panel. If you want to be able to repeat this with one click or by invoking PowerGREP from the command line once, you need to use the Sequence Panel. Mark drive C: in the File Selector, and prepare an action to replace A with B on the Action panel. But instead of executing the action, add it to the sequence by clicking the New Step button on the Sequence panel. Then mark drive D: in the File Selector, prepare another action to replace X with Y, and click the New Step button again. Now you can save this sequence or add it to a library to quickly run both replacements in the future.

But if you want to replace A with B and X with Y in a single set of files, you should do so as a single action on the Action panel. Simply set the search type on the Action panel to “list of literal text” or “list of regular expressions”. Then you can add both the A->B and X->Y replacements to the list on the Action panel. Turn off non-overlapping search if you want to make the second replacement work on the results of the first replacement as would happen when you run the two replacements as separate actions. The benefit of using a single action is that each file is opened and saved only once.

A PowerGREP Sequence executes the actions in the sequence independently. There is no way to make a later step in the action use the search matches found by an earlier step in the action, even if both steps work on the same set of files. This is just like executing two actions on the Action panel: the second action doesn’t know anything about the first. If you want to do something like searching for something in a file, and then using that match in a search-and-replace, you need to do that in a single action. PowerGREP’s way of allowing named capturing groups to carry over from one part of the action to the next makes this possible.

The only thing that can be carried over from an earlier step in the sequence to a later step is the file selection, including the ability to make a step process only the files that a previous step searched through, found matches in, did not find matches in, or created as a target file.

Building Up a PowerGREP Sequence

Before adding a step to the sequence, use the Action panel to prepare the file selection and action for the step. Then click the New Step button on the Sequence panel. Change the settings on the Action panel as needed for the second step, and click the New Step button again. You can add as many steps to the sequence as you like. Steps are executed strictly from top to bottom. Use the Up and Down toolbar buttons to change their order.

To permanently remove a step from the sequence, click the Delete button on the toolbar. To temporarily disable a step, clear the checkbox next to the step in the list on the Sequence panel.

To edit a step’s action, use the items under the Action button on the Sequence toolbar or in the Action submenu in the Sequence menu. First click Sequence to Action to copy the action from the selected step to the Action panel. Edit the action on the Action panel. Then use Action to Sequence to copy the action from the Action panel to the selected step, replacing the old action.

By default, all steps in the sequence have their “file selection” option set to “File Selector”. That means the step will search through the files marked on the File Selector panel at the time you execute the sequence. This allows you to create sequences that are not tied to a specific set of files. Whenever you want to execute the sequence on a new set of files, just load the sequence on the Sequence panel, prepare a new file selection in the File Selector, and execute the sequence.

You can make the sequence to work on a specific set of files, ignoring the File Selector. Select the first step in the sequence and then click on File Selector to Sequence under Files on the Sequence toolbar. This copies the file selection to the step. The “file selection” setting for the first step then indicates “own file selection”.

If you want another step in the sequence to work on the same set of files, select that step. Click on the “file selection” drop-down list and choose “file selection from other step”. Then select the step that has its own file selection in the “step” drop-down list. This list indicates steps by the index numbers that they have in the sequence. Those numbers are also shown in the first column of the list of steps. The benefit of copying the

file selection to only the first step that uses it and then making all other steps reference it is that you can easily change the set of files that all those steps work on simply by copying a new file selection to the first step.

A step can also work on a subset of the files that were searched through by a previous step. Instead of choosing “file selection from another step” in the “file selection” drop-down list, you can choose:

- Files matched by other step: Search through the files in which another step in the sequence already found matches.
- Files not matched by other step: Search through the files that another step searched through without finding any matches.
- Target files from other step: Search through the files that were created or overwritten by another step.

Executing a PowerGREP Sequence

Click the Preview, Execute, or Quick Execute button on the Sequence toolbar to preview or execute the sequence. The Preview and Execute buttons keep detailed results for each step. Quick Execute only counts search matches and is therefore faster and uses less memory. Previewing a sequence does not modify any files at all. Since the steps in the sequence always run independently, previewing a sequence will not give the same results as executing it if a later step searches through files that would have been created or modified by an earlier step if you had executed the sequence rather than previewing it.

If any of the actions in the sequence create backup files, then the sequence is added as a single item to the Undo History. There you can delete all the backup files created by the sequence or use them to undo the changes made by the sequence. If multiple steps in the sequence modify the same file, only one backup copy is made of that file. Undoing a sequence always undoes all steps in the sequence.

While the sequence runs, PowerGREP updates the Results panel with the results of the step that is being executed. When the last step has completed, the Results panel shows the results of the last step. To see the results of an earlier step, select the step on the Sequence panel and click on Sequence to Results under the Results button on the Sequence toolbar or in the Results submenu in the Sequence menu. This copies the results of the selected step to the Results panel.

If you double-click a step in the sequence, the file selection, action, and results of that step are all copied to the File Selector, Action, and Results panels.

Examples: Replace in file names and contents and Apply an extra search-and-replace to target files

18. Sequence Menu

The Sequence menu lists commands for use with the Sequence panel. See the Sequence reference chapter for more information on the Sequence panel itself.



Clear

Removes all steps from the Sequence panel.



Open

Loads a sequence from a PowerGREP Sequence Action file or a sequence with results from a PowerGREP Sequence Results file.

You can quickly reopen a recently opened or saved sequence file by clicking the downward pointing arrow next to the Open button on the Sequence toolbar. Or, you can click the right-pointing arrow next to the Open item in the Sequence menu. A new menu listing the last 16 opened or saved files will appear. Select “Maintain List” to access the last 100 files.



Save

Saves the whole sequence into a PowerGREP Sequence Action file. Any results that may appear on the Sequence panel are not saved. You will be prompted for the file name each time.

Sequence files are appropriate for sequences that you execute repeatedly, in exactly the same way. Sequence files can be executed from the command line. You can even generate them with other applications. Use the Library panel to store boilerplate sequence definitions for later adaptation.



Save Results

This command is only available after you’ve executed the sequence. It saves the whole sequence and the results of all its steps into a PowerGREP Sequence Results file. PowerGREP results files use a special XML-based file format. While you could process the XML file with other applications, the primary purpose of saving results files is to be able to inspect the results in PowerGREP at a later time.



Favorites

If you often open the same files, you should add them to your favorites for quick access. Before you can do so, you need to save the sequence to a file. PowerGREP’s window caption will then indicate the name of the sequence file. Click the downward pointing arrow next to the Favorites button on the Sequence toolbar, or

the right-pointing arrow next to the Favorites item in the Sequence menu. Then select “Add Current Sequence” to add the current sequence file to the favorites. Pick a file from the menu to open it.

If you click the Favorites button or menu item directly, a window will pop up where you can organize your sequence favorites. If you have many favorites, you can organize them in folders for easier reference later.



Add to Library

Adds the whole sequence definition as a single item to the PowerGREP Library. Any results that may appear on the Sequence panel are not added to the library.



New Step

Adds a new step to the sequence. The settings on the Action panel are copied to the new step. The settings on the File Selector are *not* copied to the step. Use the File Selector to Sequence menu item if you want to copy the file selection also.



Delete Step

Removes the selected step from the sequence.



Move Step Up

Moves the selected step one position upwards in the sequence, so it will be executed sooner. Steps are always executed from top to bottom, one after the other.



Move Step Down

Moves the selected step one position downwards in the sequence, so it will be executed later. Steps are always executed from top to bottom, one after the other.



File Selector to Sequence

The File Selector to Sequence item in the Files submenu of the Sequence menu copies the file selection you’ve made in the File Selector to the selected step in the sequence. The selected step will use that file selection next time you execute the sequence.



Open File Selection

The Open File Selection item in the Files submenu of the Sequence menu loads a file selection from a PowerGREP File Selection, Action, or Results file into the selected step in the sequence. The step will use that file selection next time you execute the sequence.



Sequence to File Selector

The Sequence to File Selector item in the Files submenu of the Sequence menu copies the file selection from the selected step in the sequence to the File Selector. If you have already executed the sequence, you will get the file selection that this step actually used, even if you configured the step to use another step's file selection. If you have not yet executed the sequence, the Sequence to File Selector command is only available for steps that have their own file selection.



Save File Selection

The Save File Selection item in the Files submenu of the Sequence menu saves the the file selection from the selected step in the sequence into a PowerGREP File Selection file. If you have already executed the sequence, this saves the file selection that this step actually used, even if you configured the step to use another step's file selection. If you have not yet executed the sequence, the Save File Selection command is only available for steps that have their own file selection.



Action to Sequence

The Action to Sequence item in the Action submenu of the Sequence menu copies the settings on the Action panel to the selected step in the sequence. The step will use that action next time you execute the sequence..



Open Action

The Open Action item in the Action submenu of the Sequence menu loads loads an action from a PowerGREP Action or Results file into the selected step in the sequence. The step will use that action next time you execute the sequence.



Sequence to Action

The Sequence to Action item in the Action submenu of the Sequence menu copies the action from the selected step in the sequence to the Action panel.



Save Action

The Save Action item in the Action submenu of the Sequence menu saves the action from the selected step in the sequence into a PowerGREP Action file.



Sequence to Results

The Sequence to Results item in the Results submenu of the Sequence menu copies the results from the selected step in the sequence to the Results panel.



Save Results

The Save Results item in the Results submenu of the Sequence menu saves the file selection, results from the selected step in the sequence into a PowerGREP Results file.

Clear Step Results

The Clear Step Results in the Results submenu of the Sequence menu removes the results for the selected step from the Sequence panel. If a step produced a lot of results that you aren't interested in, you can clear them to reduce PowerGREP's memory usage. It's not necessary to manually clear the results before executing the sequence again.

Clear Sequence Results

The Clear Sequence Results in the Results submenu of the Sequence menu removes the results for the entire sequence from the Sequence panel. If a sequence produced a lot of results that you aren't interested in, you can clear them to reduce PowerGREP's memory usage. It's not necessary to manually clear the results before executing the sequence again.



Preview

Click the Preview button on the Sequence toolbar, or press Alt+F9 on the keyboard, to execute the sequence without creating or modifying any files. Previewing an sequence is always perfectly safe. It will never do anything that you might regret later.

If later steps in the sequence work on files that are modified by earlier steps when executing rather than previewing the sequence, then previewing the sequence does not show the same results. Because the preview did not actually modify the files in the earlier steps, the later steps see the original files rather than the modified files. PowerGREP does not have a feature to preview the effects of modifications made on the same file by multiple steps without actually modifying the file.

 **Execute**

The Execute item in the Sequence menu executes the sequence for real. If the target type of an action in the sequence calls for files to be created or modified, executing the sequence will do so. PowerGREP will show detailed search results on the Results panel.

 **Quick Execute**

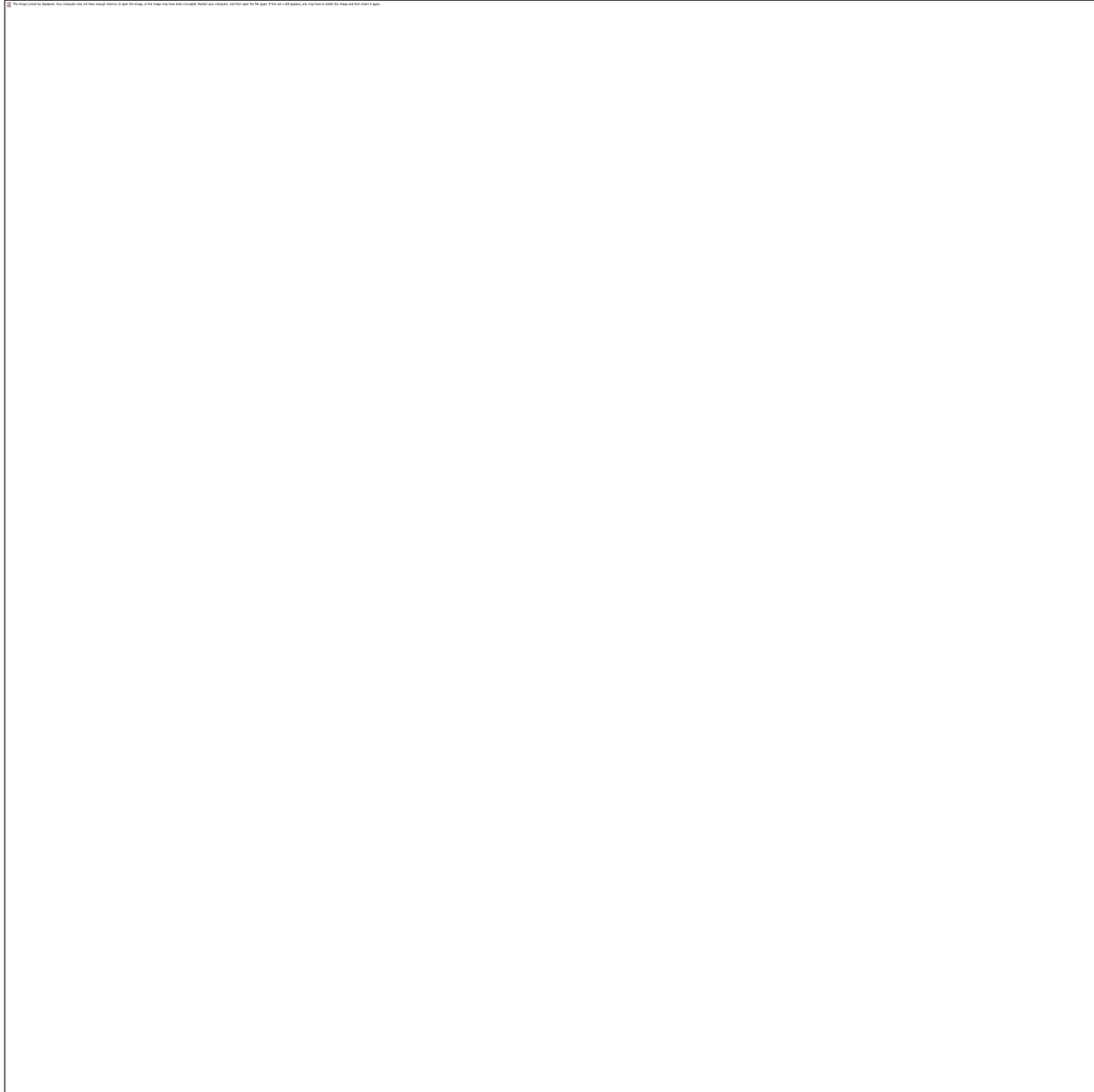
The Quick Execute item in the Sequence menu executes the sequence for real, without keeping individual match results. The Results panel will only show how many matches were found in each file. Files will be created or modified according to the target type of each action in the sequence.

Quick Execute is significantly faster than Execute or Preview, and will use far less of your computer's memory. That's because it doesn't have to keep track of each individual match to be able to show you all the details on the Results panel. If you don't plan to inspect the search results, Quick Execute is the way to go.

When preparing a new sequence that you plan to execute on a large number of files, or some very large files, you should first preview the sequence on just a couple of the files. When you're confident the sequence works the way it should, expand the file selection to all the files, and use Quick Execute to execute it for real.

19. Library Reference

A PowerGREP Library is a convenient place to store PowerGREP file selections, actions, and sequences for later reuse. You can open and save libraries on the Library panel in PowerGREP. In the default layout, you can access the library panel by clicking on its tab just below the menu bar.



To add an file selection to the library, use the Add to Library item in the File Selection menu or File Selection toolbar. To add an action to the library, use the Add to Library item in the Action menu or Action toolbar. To add a sequence to the library, use the Add to Library item in the Sequence menu or Sequence toolbar. You don't need to save the file selection, action, or sequence before adding it to the library. A copy of the entire file selection, action, or sequence definition is stored in the library file itself.

While the Save item in the Action menu saves a .pga file that includes both the file selection and the action definition, the Add to Library item in the Action menu only adds the action definition itself. To store both the file selection and the action in the library, use the Add to Library item in both the File Selection and the Action menus.

When you open a library, you will see a list of one-line descriptions of all the items in the library. The icon next to each item indicates whether it is a file selection, action, or sequence. Click on an item to make PowerGREP show the complete description along with a summary of the item itself. This is the main advantage of storing file selections, actions, and sequences in a library rather than saving them into separate files. You can easily seek out the item you want to use by comparing the descriptions.

To reuse a file selection, action, or sequence from the library, click the Use button on the Library toolbar. If you selected a file selection, it is copied to the File Selector panel. If you selected an action, it is copied to the Action panel. If you selected a sequence, it is copied to the Sequence panel. All settings on the destination panel are completely replaced with those of the item from the library.

If you use an item from the library, and then edit that item on the File Selection, Action, or Sequence panel, the item stored in the library is *not* automatically updated. You can adapt items you used from the library to the situation at hand without messing up your carefully collected library. If you do want to update the item in the library, simply use the Add to Library menu item again. If you didn't change the description on the Action or Sequence panel, PowerGREP will ask you if you want to replace the action or sequence already in the library. If you did change the description, the new action or sequence is added to the library alongside the old action or sequence.

If you want to add an action from the library to the sequence you're preparing on the Sequence panel, use the Use Item in Sequence command in the Library menu. This appends the action to the sequence. You can also combine multiple sequences. Load the first sequence into the Sequence panel. Then select the second sequence in the library and use the Use Item in Sequence command in the Library menu. This appends the sequence to the one on the Sequence panel instead of replacing it.

PowerGREP automatically and regularly saves library files. You only need to use the Save item in the Library menu or the Save button on the Library toolbar when you want to save a copy of the library under a new name, or when you want to give a new library a name.

Library files are automatically synchronized between multiple instances of PowerGREP. If you open the same library in two or more instances, any modifications you make to the library in one instance will automatically appear in all other instances. There is no risk of data loss when you edit a library in more than one PowerGREP instance at the same time.

20. Library Menu

The Library menu lists commands for use with the Library panel. See the Library reference chapter for more information on the Library panel itself.



New

Starts with a blank, untitled PowerGREP library. The library is not saved until you click the Save button to give it a name.



Open

Opens a previously saved PowerGREP library. You can quickly reopen a library you recently worked with by clicking the downward pointing arrow next to the Open button on the Library toolbar. Or, you can click the right-pointing arrow next to the Open item in the Library menu. A new menu listing the last 16 opened or saved files will appear. Select “Maintain List” to access the last 100 files.

You can open the same library in multiple instances of PowerGREP without risk. Library files are automatically synchronized between multiple instances of PowerGREP. There is no risk of data loss when you edit a library in more than one instance at the same time.



Save

Saves the library under a new name. PowerGREP automatically and regularly saves library files. You only need to use the Save command when you want to save a copy of the library under a new name.



Favorites

If you often use the same library files, you should add them to your favorites for quick access. Click the downward pointing arrow next to the Favorites button on the Library toolbar, or the right-pointing arrow next to the Favorites item in the Library menu. Then select “Add Current Library” to add the current action file to the favorites. Pick a file from the menu to open it.

If you click the Favorites button or menu item directly, a window will pop up where you can organize your library favorites.



Use Item

If you selected a file selection in the library, it is copied to the File Selector panel. If you selected an action, it is copied to the Action panel. If you selected a sequence, it is copied to the Sequence panel. All settings on the destination panel are completely replaced with those of the item from the library.



Use Item in Sequence

If you selected an action, it is added to the sequence on the Sequence panel. If you selected a sequence, it is appended to the sequence on the sequence panel.



Delete Item

Deletes the selected item from the library. This cannot be undone.

21. Results Reference

While PowerGREP previews or executes an action, the Results panel shows a progress meter. While PowerGREP collects the list of files to search through, the progress meter stays at 0% while counting up the number of files it is about to search through. While searching through files the meter indicates the percentage of files already searched through at the left, and an estimate of the remaining time the action needs until completion at the right.

If you have configured PowerGREP to use multiple threads in the action preferences and the file selection includes files from two or more drives, it is possible for the progress meter to appear running backwards or back-and-forth. Because PowerGREP will use one thread to get the list of files from each drive, it is possible that PowerGREP starts searching through the files on one drive while it is still getting the list of files on another drive. That causes the thread searching through files to advance the progress meter as it completes part of the job while the other thread reduces the progress meter as it increases the size of the job by adding files to the list. A “still globbing” label on the progress meter indicates this situation. Once the list of files has been gathered for all drives the progress meter will advance normally.

By default, the results are updated at regular intervals as PowerGREP finds more search matches. If you find this distracting, turn off the Automatic Update button. You can then click the Update Results button to see the results gathered so far. to make PowerGREP update the results regularly. Updating the results slows down the search somewhat, but does allow you to inspect the results before the action has completed.

Changing Display Options

The Results panel provides six sets of options to rearrange the display of the results, without executing the action again. When Automatic Update is active, changing an option immediately rearranges the results. If not, you need to click the Update Results button to rearrange them according to the new options. Automatic update is more convenient when working with small result sets. Manual update is faster when working with large result sets, since it allows you to change multiple options before updating the results.

Display Files and Matches

- **Do not show files or matches:** Do not show any file names or matches. Only totals will be shown.
- **File names only:** Display the names of the files searched through and their totals. Do not display matches.
- **Target file names only:** Display the names of the target files that were created or modified and their totals. Do not display matches.
- **Matches without context:** Display search matches without extra information. File names and file totals are shown when grouping per file.
- **Matches with context numbers:** Display search matches along with the number of the line or the context block the match was found on.
- **Matches with context:** Display the context that was collected for the matches, and highlight the matches.
- **Matches with context and context numbers:** Same as “matches with context” and also showing the number of the line or the context block the match was found on.

- **Aligned matches with context:** Same as “matches with context”, but results for matches with less context before them are indented so all highlighted matches line up (if you turn off word wrap) as in a concordance.
- **Aligned matches with context and numbers:** Same as “aligned matches with context” and also showing the number of the line or the context block the match was found on.

Note: When grouping unique matches, the context choices determine how matches are shown when double-clicking a unique match.

Group Search Matches

If you selected to display files and/or matches, you can also select how the files and matches should be displayed.

- **Do not group matches:** Show all matches, without indicating file names.
- **Per file:** Show file names, and all matches for each file.
- **Per file, with or without matches:** Show file names and all matches. Also show file names of files without matches.
- **Per file, then per unique match:** Show file names, and unique matches for each file. Per file, per match, with or without matches: Show file names and unique matches. Also show file names of files without matches.
- **Per unique match:** Show each unique match once, regardless of the files in which it was found. Do not indicate file names.
- **Per unique match, listing files:** Show each unique match once. List the names of the files the match was found in below each match.

Display Totals

- **Do not show totals:** Never indicate totals.
- **Show totals before the results:** Show the number of matches and files before the results. When grouping per file, show the number of matches in that file between the file name and the file’s matches.
- **Show totals after the results:** Show the number of matches and files after the results. When grouping per file, show the number of matches in that file after the file’s matches.
- **Show totals for grouped matches.:** Do not show overall or per-file totals. When grouping unique matches, show the number of occurrences before each match.
- **Totals before results, and grouped matches.:** Show overall and per-file totals before the results, and show the number of occurrences of each match when grouping unique matches.
- **Totals after results, and grouped matches.:** Show overall and per-file totals after the results, and show the number of occurrences of each match when grouping unique matches.

Sort Files

- **First searched to last searched:** Files are added to the bottom of the results as PowerGREP searches through them. Because PowerGREP processes files, disk drives, and network servers in parallel, this order is not deterministic.

- **Last searched to first searched:** Files are added to the top of the results as PowerGREP searches through them. Because PowerGREP processes files, disk drives, and network servers in parallel, this order is not deterministic.
- **Alphabetically, A..Z:** Sort files in ascending alphabetic order of their full path names.
- **Alphabetically, Z..A:** Sort files in descending alphabetic order of their full path names.
- **By increasing totals:** Sort files by the number of matches found in each file, from least to most.
- **By decreasing totals:** Sort files by the number of matches found in each file, from most to least.
- **Newest to oldest:** Sort files by the date and time they were last modified, from newest to oldest.
- **Oldest to newest:** Sort files by the date and time they were last modified, from oldest to newest.

Sort Matches

- **Show in original order:** Show matches in the order they have in the file they were found in.
- **Alphabetically, A..Z:** Sort matches alphabetically, from A to Z.
- **Alphabetically, Z..A:** Sort matches alphabetically, from Z to A.
- **By increasing totals:** When grouping unique matches, sort them from least occurrences to most occurrences.
- **By decreasing totals:** When grouping unique matches, sort them from most occurrences to least occurrences.

If a line or block of context has more than one match, each line or block of context is shown only once with all matches highlighted if you choose “show in original order”. If you choose any other sort option, lines or blocks of context with multiple matches are shown multiple times, each time with one search match highlighted, in the correct sorting position for each match.

Display Replacements

Determines how search matches and their replacements are displayed after a search-and-replace or collect action.

- **Search match only:** Display search matches, without their replacements.
- **Replacement only:** Display replacements instead of search matches.
- **In-line match and replacement:** Display both matches and replacements, next to each other.
- **Separate match and replacement:** Display matches and replacements separately. When showing context, the context is duplicated.

22. Results Menu

The Results menu lists commands for use with the Results panel. See the Results reference chapter for more information on the Results panel itself.



Clear

Clears the Results panel, and unloads the information gathered by the last action from memory. Clearing the results also clears the results information from the File Selector, without clearing the file selection.



Open

Loads a previously saved PowerGREP results file. The results file also contains the action definition and file selection that produced the results. These will be loaded from the results file into the Action panel and File Selector respectively.

You can quickly reopen a recently opened or saved results file by clicking the downward pointing arrow next to the Open button on the Results toolbar. Or, you can click the right-pointing arrow next to the Open item in the Results menu. A new menu listing the last 16 opened or saved files will appear. Select “Maintain List” to access the last 100 files.



Save

Saves the results shown on the Results panel along with the action definition and file selection that produced those results into a PowerGREP results file. PowerGREP results files use a special XML-based file format. While you could process the XML file with other applications, the primary purpose of saving results files is to be able to inspect the results in PowerGREP at a later time.

If you want to process search matches found by PowerGREP with other software, running a “collect data” action with the appropriate target settings is usually more useful. That way, you collect only the raw search matches, grouped, sorted and delimited the way you want (if at all).



Favorites

If you often open the same files, you should add them to your favorites for quick access. Before you can do so, you need to save the results to a file. PowerGREP’s window caption will then indicate the name of the results file. Click the downward pointing arrow next to the Favorites button on the Results toolbar, or the right-pointing arrow next to the Results item in the Results menu. Then select “Add Current Results” to add the current results file to the favorites. Pick a file from the menu to open it.

If you click the Favorites button or menu item directly, a window will pop up where you can organize your results favorites. If you have many favorites, you can organize them in folders for easier reference later.

By default, the Favorites button is not visible on the toolbar. To make it visible, click on the downward pointing arrow at the far right end of the Results toolbar. A menu will pop up where you can toggle the visibility of all toolbar buttons.



Export

Saves the results as shown on the Results panel into a plain text file, an HTML file, or an RTF file. The file will contain exactly the same text as shown on the Results panel, including file headers, totals, etc. If you export to an HTML file or an RTF file, it will use the same color coding as the results in PowerGREP. This can be useful if you want to make the results part of a web site or a report you're writing in a word processor.

If you want to process search matches found by PowerGREP with other software, running a “collect data” action with the appropriate target settings is usually more useful than exporting the results to a text file. That way, you collect only the raw search matches, grouped, sorted and delimited the way you want (if at all).



Print

Print the results as shown on the Results panel. A print preview will appear. The print preview allows you to configure the printer and page layout before printing.



Update Results

When automatic results update is off (see next item), use the Update Results command to check PowerGREP's progress while previewing or executing an action, and after you've changed the display options on the Results panel.



Automatic Update

Automatic update is a toggle. When on, results are regularly updated while PowerGREP executes an action, and results are rearranged immediately when you change a display option on the Results panel. Using automatic update is only recommended when working with small results sets.



Next Match

Jump to the next highlighted match in the results.



Previous Match

Jump to the previous highlighted match in the results.



Next File

Jump to the next file in the results.



Previous File

Jump to the previous file in the results.



Make Replacement

Replaces the search matches in the selected text or the search match that the text cursor is on with the replacement text that was prepared for each match while executing the last PowerGREP action. The color of the replaced matches changes to indicate they have been replaced.

You can only make replacements after previewing or executing a search-and-replace action, since PowerGREP needs to know which replacement text to use. Quick Replace does not allow you to replace individual matches, since Quick Replace discards information about individual matches.



Revert Replacement

Reverts the search matches in the selected text or the search match that the text cursor is on by replacing them with the original text that was matched while executing the last PowerGREP action. The color of the reverted matches changes to indicate they have been reverted.

You can only revert replacements after previewing or executing a search-and-replace action. Quick Replace does not allow you to revert individual replacements, since Quick Replace discards information about individual matches.



Make Replacements in This File

Replaces the search matches in the file that the cursor points to in the results with the replacement text that was prepared for each match while previewing or executing the last PowerGREP action. The color of the replaced matches changes to indicate they have been replaced.



Revert Replacements in This File

Reverts the search matches in the file that the cursor points to in the results with the original text that was matched while previewing or executing the last PowerGREP action. The color of the reverted matches changes to indicate they have been reverted.



Make Replacements in All Files

Replaces all search matches shown in the results with the replacement text that was prepared for each match while previewing or executing the last PowerGREP action. The color of the replaced matches changes to indicate they have been replaced.

This command is enabled only if PowerGREP was able to hold the details of all replacements in all files in memory during the last action. If PowerGREP ran out of memory and only partial results are shown, the command to make replacements in all files is disabled to make sure you do not get the mistaken impression that you can make replacements in all files. The commands for replacing only the selected search matches or the search matches in the selected files will still be available to allow you to make the replacements that PowerGREP was able to keep in memory.



Revert Replacements in All Files

Reverts all search matches shown in the results with the original text that was matched while previewing or executing the last PowerGREP action. The color of the reverted matches changes to indicate they have been reverted.

This command is enabled only if PowerGREP was able to hold the details of all replacements in all files in memory during the last action. If PowerGREP ran out of memory and only partial results are shown, the command to revert replacements in all files is disabled to make sure you do not get the mistaken impression that you can use this command to revert the replacements in all files. The only way to revert all replacements in this situation is to use the Undo History to restore all the files from their backups. The commands for reverting only the selected replacements or the replacements in the selected files will still be available to allow you to revert the replacements that PowerGREP was able to keep in memory.



Folding

Use the Fold and Unfold items in the Folding submenu of the Results menu to collapse or expand the results for the file that the cursor is on in the results. You can also do this by clicking the small plus or minus in the left margin of the results. Use the Fold All and Unfold All items in the Folding submenu of the Results menu or their corresponding buttons on the Results toolbar to collapse or expand the results of all files.



Go to Bookmark

Move the cursor to one of ten bookmarks that you previously set in the results.



Set Bookmark

Set one of ten bookmarks at the item that the text cursor points to in the results. An item can be a highlighted match, context around a match, or a file name. The bookmark appears in the margin next to the item. If an

item occurs more than once in the results, then the bookmark is shown next to each occurrence of the item. The bookmark remains associated with the item if you rearrange the results display by changing the display options using the drop-down lists on the Results panel. Bookmarks are also saved into the .pgr file when you use the Save item in the Results menu.

You can remove a bookmark by setting the same bookmark again on the same item. If you set a different bookmark on the same item, then the other bookmark that was already set on the item is removed, and the different bookmark is set.

If two items are shown on the same line in the results, such as two search matches found on the same line in a file, then it is possible to set two bookmarks on the same line, one on each item. Only one of the bookmarks will be shown in the margin. But both bookmarks will be set and respond to the Go to Bookmark command. If you rearrange the results so that the two items are no longer shown on the same line, such as by sorting search matches alphabetically, then both bookmarks will appear in the margin.



Font and Text Direction

Configure the text layout or select a previously configured text layout to change the font, text direction, cursor behavior, and spacing used by the Results panel. The Configure Text Layout topic in the reference section about PowerGREP's Preferences screen has much more information on this.



Word Wrap

Toggles word wrap on or off. When on, lines in the results that are too long to fit the width of the Results panel are wrapped across multiple lines. When off, you will need to use the horizontal scroll bar to see the remainder of long lines.



Edit File

Click on a file name in the results, and select the Edit File command to open that file in PowerGREP's built-in file editor. The editor can edit both text and binary files, and will highlight search matches.

If you prefer to use an external editor or application to view or edit the file, first configure the editor or application in the external editors preferences. You can then click on the downward pointing arrow next to the Edit button on the toolbar, or the right-pointing arrow next to the Edit item in the Results menu, to open the selected file with that application. The applications that are associated with that file type in Windows Explorer are also listed in the Edit submenu.

If you configured an external editor as the default editor, then the Edit File command will invoke that editor instead of using PowerGREP's built-in editor. This saves you having to go through the Edit File submenu.



Locate in File Selector

Use the File Selector to locate the file that the text cursor points to in the results.



Include in Next Action

Use the File Selector to mark the file that the text cursor points to in the results so that this file is included in the next action.



Exclude from Next Action

Use the File Selector to mark the file that the text cursor points to in the results so that this file is excluded from the next action.



Open File in EditPad

Click on a file name in the results, and select the Edit File command to open that file in EditPad. EditPad is a most convenient text editor. Just like PowerGREP, EditPad has been designed by Jan Goyvaerts and is sold by Just Great Software Co. Ltd. EditPad is available at <http://www.editpadpro.com/>.

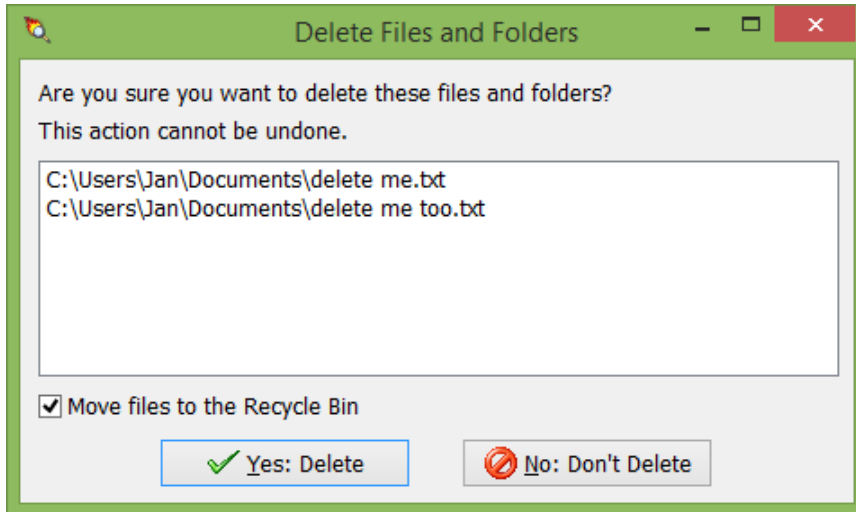


Delete Files

The Delete Files submenu of the Results menu allows you to delete four sets of files:

1. Delete Selected Files and Folders: Delete the files that you have selected in the results. A file is selected if the selected text in the results includes its file name, one of its search matches, or some context of one of its matches.
2. Delete Matched Files: Delete all the files in which search matches were found during the previously executed action.
3. Delete Unmatched Files: Delete all the files that were searched through but did result in any matches during the previously executed action.
4. Delete Target Files: Delete all target files that were created during the previously executed action. Note that this is not the same as undoing the action. PowerGREP's undo history restores backup files. Deleting target files in the Results does not.

All four options ask for confirmation before actually deleting any files. The confirmation lists the files that will be deleted and gives you the option between moving the files to the Windows Recycle Bin or permanently deleting the files. Neither choice allows you to undo deleting the files in PowerGREP. If you choose to move the files to the Recycle Bin, you can recover the files manually from the Recycle Bin icon on your Windows desktop, at least until you make the Recycle Bin empty.



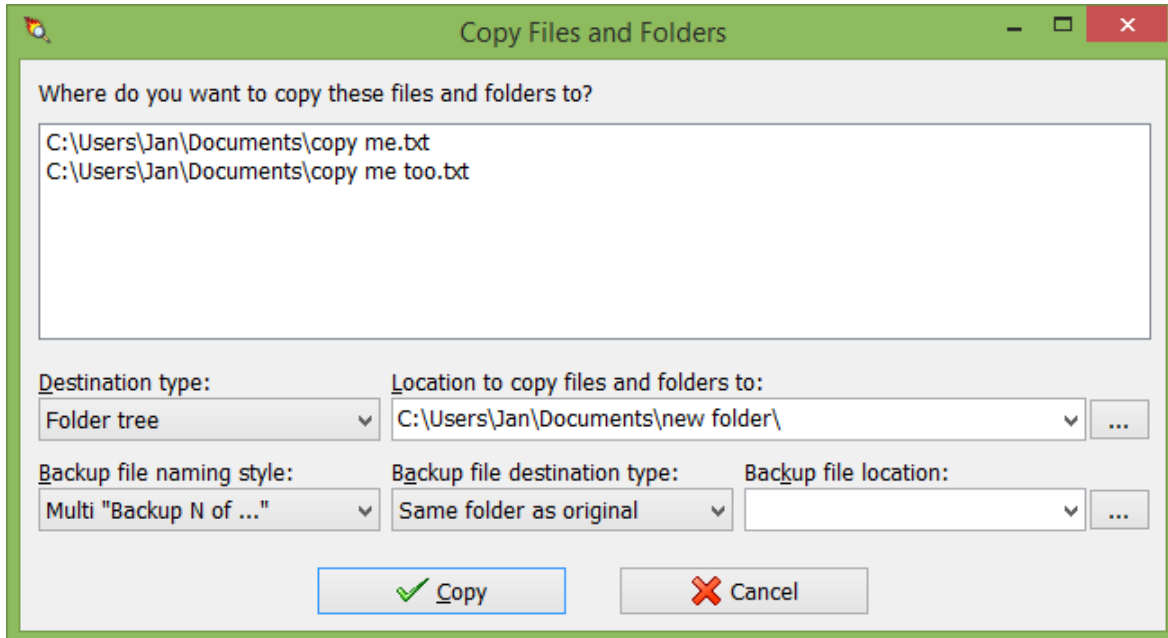
Copy Files

The Copy Files submenu of the Results menu allows you to copy four sets of files:

1. Copy Selected Files and Folders: Copy the files that you have selected in the results. A file is selected if the selected text in the results includes its file name, one of its search matches, or some context of one of its matches.
2. Copy Matched Files: Copy all the files in which search matches were found during the previously executed action.
3. Copy Unmatched Files: Copy all the files that were searched through but did result in any matches during the previously executed action.
4. Copy Target Files: Copy all target files that were created during the previously executed action.

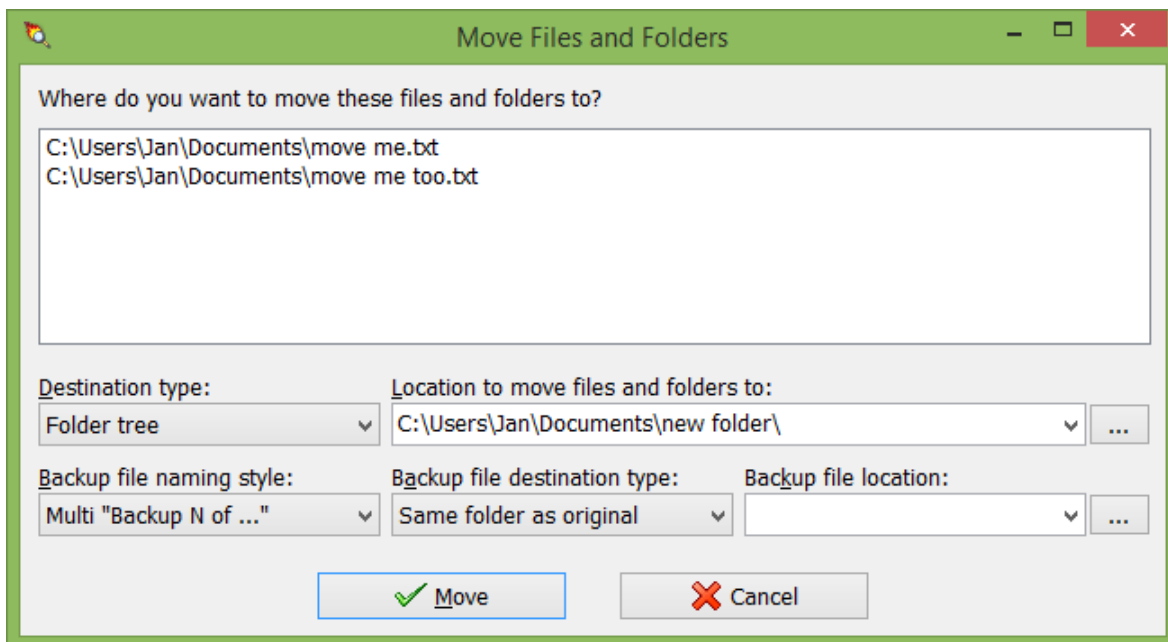
All four options show a screen listing all files that you are about to copy. You can specify the destination for the copied files and whether backups should be created for any files that are overwritten. These are the same target destination and backup options as on the Action panel.

After the files are copied a new item appears in the Undo History. There you can undo copying the files if you choose to create backups, or if no files were overwritten during the copy operation. The Undo History also allows you to clean up backup files when you're sure you don't want to undo the operation.



Move Files

The Move Files submenu of the Results menu offers the same options as the Copy Files menu. It shows the same screen with options. The only difference is that the files are moved rather than copied to their new locations. Move operations are also added to the Undo History.



23. Editor Reference

PowerGREP sports a full-featured built-in file editor for editing both text files and binary files. PowerGREP's editor is built on the same technology as EditPad, a popular text editor designed by Jan Goyvaerts and sold by Just Great Software Co. Ltd., just like PowerGREP.

The key advantage of PowerGREP's built-in editor is that it highlights search matches when you preview or execute an action (but not when you use Quick Execute). If you've previewed a search-and-replace action, you can replace individual matches in the editor with just one click. If you executed the search-and-replace, you can revert individual replacements to the original text with just one click. Replacing and reverting individual matches is much more comfortable than answering yes/no for each replacement while the action is executed, as most other grep tools do. The editor supports unlimited undo and redo, so mistakes are easy to fix. The highlighting is automatically updated to reflect replaced matches and reverted replacements.

Cursor Movement Keys

Arrow key	Moves the text cursor (blinking vertical bar).
Ctrl+Left arrow [text]	Moves the text cursor to the start of the previous word or the end of the previous line, whichever is closer.
Ctrl+Right arrow [text]	Moves the text cursor to the start of the next line or the end of the current line, whichever is closer.
Ctrl+Up/Down arrow	Scrolls the text one line up or down.
Page up/down	Moves the text cursor up or down an entire screen.
Ctrl+Page up/down	Scrolls the text one screen up or down.
Home	Moves the text cursor to the beginning of the line.
Ctrl+Home	Moves the text cursor to the start of the entire text.
End	Moves the text cursor to the end of the line.
Ctrl+End	Moves the text cursor to the end of the entire text.
Shift+Movement key	Moves the text cursor and expand or shrink the selection towards the new text cursor position. If there was no selection, one is started. Pressing Ctrl as well, will move the text cursor correspondingly.
Alt+Shift+Movement [text]	The same as when Alt is not pressed, except that if word wrap is off, the selection will be rectangular instead of flowing along with the text.

Editing Commands

Enter	Inserts a line break.
Shift+Enter	Inserts a line break.
Ctrl+Enter	Inserts a page break.
Delete	Deletes the current selection if there is one and selections are not persistent. Otherwise, the character to the right of the caret is deleted.
Ctrl+Delete	Deletes the current selection if there is one. Otherwise, the part of the current word to the right of the text cursor is deleted [text].

	Deletes the current selection [hex].
Shift+Ctrl+Delete	All the text on the current line to the right of the text cursor is deleted [text]. Deletes the current selection [hex].
Backspace	Deletes the current selection if there is one and selections are not persistent. Otherwise, the character to the left of the caret is deleted.
Ctrl+Backspace	Deletes the current selection if there is one and selections are not persistent. Otherwise, the part of the current word to the left of the text cursor is deleted [text]. Deletes the current selection [hex].
Shift+Ctrl+Backspace	Deletes the current selection if there is one and selections are not persistent. Otherwise, all the text on the current line to the left of the text cursor is deleted [text]. Deletes the current selection [hex].
Ctrl+Z	Undo the last edit
Ctrl+R	Redo the last undone edit
Alt+Backspace	Alternative shortcut for Undo
Alt+Shift+Backspace	Alternative shortcut for Redo
Insert	Toggles between insert and overwrite mode.
Tab [text]	If there is a selection, the entire selection is indented. Otherwise, a tab is inserted.
Tab [hex]	Makes the text cursor switch between the hexadecimal side and text side.
Shift+Tab [text]	If there is a selection, the entire selection is unindented (outdented). Otherwise, if there is a tab, or a series of spaces the size of a tab, to the left of the text cursor, that tab or spaces are deleted.
Ctrl+A	Select all
Ctrl+Y	Delete the current line
Shift+Ctrl+Y	Duplicate the current line

Clipboard Commands

Ctrl+X	Cut: Delete the selected text and put it on the clipboard.
Shift+Ctrl+X	Cut Append: Delete the selected text, and append it to the text already on the clipboard.
Ctrl+C	Copy: Put the selected text on the clipboard, replacing any data held by the clipboard.
Shift+Ctrl+C	Copy Append: Append the selected text to the text already on the clipboard.
Ctrl+V	Paste: Insert the text held by the clipboard.
Shift+Ctrl+V	Swap with Clipboard: Replace the text on the clipboard with the selected text, and vice versa.
Shift+Delete	Alternative shortcut for Cut
Ctrl+Insert	Alternative shortcut for Copy
Shift+Insert	Alternative shortcut for Paste

Mouse Actions

Dragging means to move the mouse before releasing the mouse button you pressed. If you move the mouse pointer to the edge of the editor space while dragging, the text will start to scroll automatically. Modifier keys

like shift or control must be pressed before pressing the mouse button and kept depressed until the mouse button is released.

Left click	Moves the text cursor to the spot where you clicked.
Shift+Left click	Moves the text cursor and expands or shrinks the selection. If there is no selection, the text between the old and new cursor positions becomes selected. If you click outside of the selection, the selection plus the text between the selection and the new cursor position becomes selected. If you click inside the selection, the new selection is the text between the original start of the selection and the new cursor position.
Left click+drag	When clicking outside the selection, a new selection is created from the point where you press the mouse button until the point where you release it. When clicking inside the selection, the selected text deleted and inserted again at the spot (outside the selection) where you release the mouse button.
Shift+Left click+drag	Expands or shrinks the selection like Shift+Left click, but then the text cursor is moved and the selection adjusted until you release the mouse button.

If you press Alt while changing the selection with the mouse, and word wrap is off, the selection becomes rectangular.

Rotate wheel	Scrolls the text a single line up or down.
Shift+Wheel	Moves the text cursor a line up or down, like pressing the up or down arrow keys on the keyboard.
Ctrl+Wheel	Scrolls the text an entire screen up or down.
Shift+Ctrl+Wheel	Moves the text cursor a screen up or down, like pressing page up or down on the keyboard.

24. Editor Menu

The Editor menu lists commands for use with PowerGREP's built-in file editor. See the Editor reference chapter for more information on the file editor itself.



New

Start with a blank, untitled file.



Open

Open a file in the file editor. You can quickly reopen a recently opened or saved results file by clicking the downward pointing arrow next to the Open button on the Results toolbar. Or, you can click the right-pointing arrow next to the Open item in the Results menu. A new menu listing the last 16 opened or saved files will appear. Select "Maintain List" to access the last 100 files.



Save

Save the file you are editing in the file editor. The Save command only becomes enabled when you've modified the file.

Before saving, PowerGREP will create a backup copy of the file using the naming style you set in the Action & Results Preferences. The Undo History will keep track of the backup copy in a separate action.



Save As

Save the file you are editing under a new name. If you later save the file again, it will again be saved under the new name. The existing copy of the file under the old name will remain.

If a file with the new name already exists, PowerGREP will create a backup copy of the existing file using the naming style you set in the Action & Results Preferences. The Undo History will keep track of the backup copy in a separate action.



Favorites

If you often open the same files, you should add them to your favorites for quick access. Before you can do so, you need to save the Editor to a file. PowerGREP's window caption will then indicate the name of the Editor file. Click the downward pointing arrow next to the Favorites button on the Editor toolbar, or the

right-pointing arrow next to the Editor item in the Editor menu. Then select “Add Current Editor” to add the current Editor file to the favorites. Pick a file from the menu to open it.

If you click the Favorites button or menu item directly, a window will pop up where you can organize your Editor favorites. If you have many favorites, you can organize them in folders for easier reference later.

By default, the Favorites button is not visible on the toolbar. To make it visible, click on the downward pointing arrow at the far right end of the Editor toolbar. A menu will pop up where you can toggle the visibility of all toolbar buttons.



Print

Print the file you are editing. A print preview will appear. The print preview allows you to configure the printer and page layout before printing. You can limit the printout to the selected part of the file, and/or to a particular range of pages.



Next Match

Highlights the next search match in the file. If there are no further search matches after the cursor position, the cursor is moved to the end of the file.



Previous Match

Highlights the previous search match in the file. If there are no search matches before the cursor position, the cursor is moved to the start of the file.



Next File

Closes the file you are editing, prompting to save if needed, and then opens the next file with search results. The order of the files is determined by the “sort files” drop-down list on the Results panel.



Previous File

Closes the file you are editing, prompting to save if needed, and then opens the previous file with search results. The order of the files is determined by the “sort files” drop-down list on the Results panel.



Make Replacement

Replaces the search matches in the selected text or the search match that the text cursor is on with the replacement text that was prepared for each match while executing the last PowerGREP action. The color of the replaced matches changes to indicate they have been replaced.

You can only make replacements after previewing or executing a search-and-replace action, since PowerGREP needs to know which replacement text to use. Quick Replace does not allow you to replace individual matches, since Quick Replace discards information about individual matches.



Revert Replacement

Reverts the search matches in the selected text or the search match that the text cursor is on by replacing them with the original text that was matched while executing the last PowerGREP action. The color of the reverted matches changes to indicate they have been reverted.

You can only revert replacements after previewing or executing a search-and-replace action. Quick Replace does not allow you to revert individual replacements, since Quick Replace discards information about individual matches.



Make All Replacements

Replaces all the search matches in the file that you have open in the Editor with the replacement text that was prepared for each match while executing the last PowerGREP action. The color of the replaced matches changes to indicate they have been replaced.



Revert All Replacements

Reverts all the search matches in the file that you have open in the Editor with the original text that was matched while executing the last PowerGREP action. The color of the reverted matches changes to indicate they have been reverted.



Fold

Folds the selected lines. The first line in the selection remains visible, while the others are hidden. They are “folded” underneath the first line. A square with a plus symbol in it appears to the left of the folded line.

If you did not select part of the text, and the text cursor is inside a foldable range indicated by a vertical line in the left margin, the Fold command folds that folding range.

While folded lines are invisible, they are still fully part of the file, and still take part in all editing actions. E.g. if you select a block that includes one or more folded sections and copy the block to the clipboard, all selected lines, including lines hidden by folding, are copied to the clipboard. Folding only affects the display.

In the editor preferences you can configure PowerGREP to add automatic folding points based on indentation and/or file navigation schemes. Automatic folding points appear as unfolded ranges that you can fold with the mouse or the Fold command.



Unfold

The Unfold command only becomes enabled when you've placed the text cursor on the first (still visible) line of a folded section. It unfolds a section previously folded with the Fold command. Clicking the square with the plus symbol in it is another way of unfolding a folded section. The unfolded section remains marked as a folding point. A square with a minus marks the first line, with a vertical line extending down from it to mark the previously folded range. You can easily re-fold it by clicking the square with the minus.

If you've made a selection, then the Unfold command unfolds all folding points inside the selection.



Fold All

Folds all folding ranges that you have previously unfolded or that were automatically added based on indentation and/or a file navigation scheme as configured in the editor preferences.



Unfold All

Unfolds all sections that you have folded with the Fold and Fold All commands.



Font and Text Direction

Configure the text layout or select a previously configured text layout to change the font, text direction, cursor behavior, and spacing used by PowerGREP's built-in editor. The Configure Text Layout topic in the reference section about PowerGREP's Preferences screen has much more information on this.



Word Wrap

Toggles word wrap on or off. When on, lines in the results that are too long to fit the width of the Results panel are wrapped across multiple lines. When off, you will need to use the horizontal scroll bar to see the remainder of long lines. Word wrap must be off to enable rectangular selections.



Line Numbers

Toggles showing line numbers in the left margin on or off.

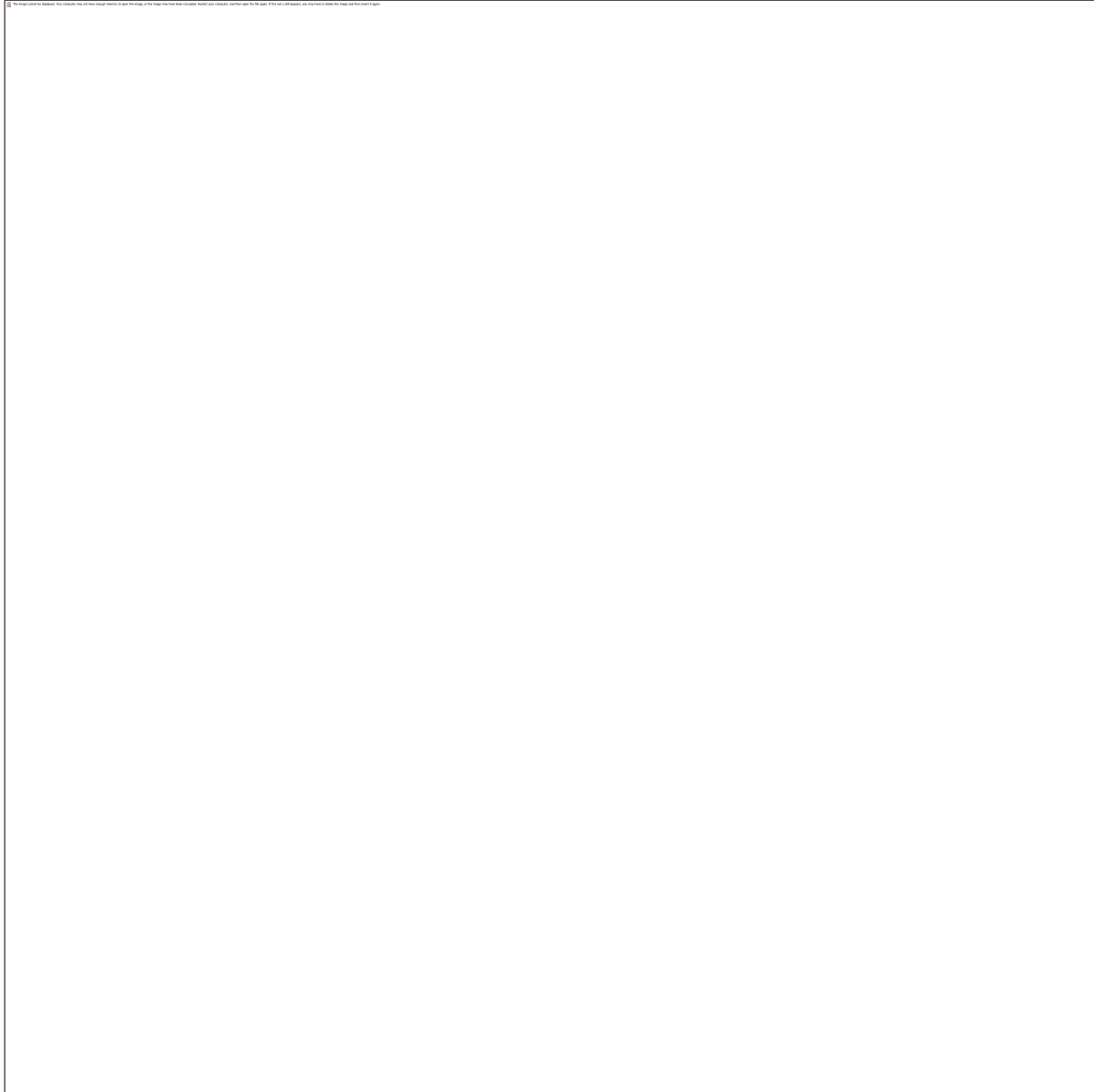


Auto Indent

Turn on automatic indent if you want the next line to automatically start at the same column position as the previous line whenever you press Enter on the keyboard while in the editor. The editor will accomplish this by counting the number of spaces and tabs at the beginning of the previous line and inserting them at the beginning of the new line you created by pressing Enter. This is most useful when editing source code and other structured files.

25. Undo History Reference

Whenever you execute an action that creates or modifies one or more files, PowerGREP automatically adds the action to the undo history. In the default layout, you can access the undo history by clicking on the Undo History tab.



To be able to undo an action, backup copies must have been created of files that were overwritten. Be sure to set the backup options you want before executing an action. You can easily delete backup files that are no longer needed in the undo history.

The undo history lists all actions that modified files since you last cleaned the undo history. The most recent actions are listed at the top. Since the same file may have been modified by more than one action, you should always undo actions from top to bottom, when you want to undo multiple actions.

To undo an action, simply click the Undo Action button. All files that were modified will be replaced with their backup copies. The backup copies are deleted in the process. If you want to execute an action again, whether you undid it or not, click the Use Action button. PowerGREP will extract the file selection and action definition from the undo history, ready to be executed again.

Actions that have been undone, and actions that cannot be undone because the backup files were deleted, stay behind in the undo history. To remove them, either delete individual actions from the undo history, or use the Clean History item in the Undo History menu menu to remove all actions that have been undone or cannot be undone.

PowerGREP Undo Manager

PowerGREP automatically saves the undo history. If you did not select an undo history file, PowerGREP will save a file called "PowerGREP Undo History.pgu" in your "My Documents" folder. If for some reason the undo history cannot be saved, PowerGREP will prompt you for another location to save the undo history. PowerGREP will not allow undo information to be lost. It will keep on prompting until it manages to save the undo history.

If you run more than one PowerGREP instance at the same time, the undo history is automatically synchronized between all instances. PowerGREP's undo manager handles this in the background. While one or more instances of PowerGREP is running, an application called PowerGREPUndoManager.exe will be running in the background. When you close the last PowerGREP instance, the undo manager closes automatically.

26. Undo History Menu

The Undo History menu lists commands for use with PowerGREP's undo history. See the undo history reference chapter for more information on the undo history itself.



Undo Action

Undo the action you selected in the undo history. All files that were created by the action will be deleted. All files that were modified or overwritten by the action will be replaced with their backup copies. The backup copies are deleted in the process.

The action will remain in the undo history. It will be indicated as “already undone”.



Use Action or Sequence

If the item selected in the undo history was the result of executing a sequence, the action definition and file selection are copied to the Action and File Selector, respectively. You can then use the Preview, Execute or Quick Execute command in the Action menu to execute the same action again.

If the item in the undo history was the result of executing a sequence, then the sequence is copied to the Sequence panel. You can then use the Preview, Execute or Quick Execute command in the Sequence menu to execute the same sequence again.



Select History

By default, PowerGREP saves the undo history in a file called "PowerGREP Undo History.pgu" in your "My Documents" folder. If you would rather use a different file or folder to save the undo history, use the Select History command. PowerGREP will continue to use the newly selected undo history file until you select another one, even after you close and restart PowerGREP.

If you select an undo history file that does not yet exist, PowerGREP will create the new file, and keep the existing undo history. No undo information will be lost by selecting a new undo history file.



Clean History

Removes all actions from the undo history that have already been undone, or that cannot be undone because their backup files were deleted. Cleaning the undo history reduces clutter. It does not affect any files.

Delete Action

Deletes the selected action from the undo history. If the action had not yet been undone, you will no longer be able to undo it automatically with PowerGREP. If the action's backup files had not yet been deleted, they will remain behind.



Delete Backup Files

Deletes all backup files created by the action. You will not be able to undo the action after deleting the backup files. Use this command to reclaim disk space when you're certain the action did what you wanted, and you don't want to undo it.

The action will not be removed from the undo history. Its "undoable" status will be indicated as "no".

27. Change PowerGREP's Appearance

PowerGREP's interface is completely modular. The application consists of nine panels. You can activate each of the panels through the View menu, whether you closed the panel or not. You can freely arrange all seven panels to best suit the way you like to work with PowerGREP.

The default layout is optimized for a computer with a single monitor and average screen resolution. If your computer has a single large resolution monitor, you can make use of the additional space by docking side-by-side some of the panels that are tabbed by default. If your computer has more than one monitor, take advantage of both monitors by making one or more panels float. Then drag the floating panels off to the second monitor.

How to Rearrange PowerGREP's panels

To move a panel, use the mouse to drag and drop its caption bar (for a panel docked to the side, or a floating panel) or its tab (for a tabbed panel). While you drag the panel, squares appear at the four edges of PowerGREP's window. While dragging over another panel, five squares appear in the center of that panel. Drop the panel onto one of the four squares at the edges of PowerGREP's window to dock the panel to that edge. Drop the panel onto one of the four outer squares in the center of another panel to dock the dragged panel to one of the four sides of the panel you're dropping it onto. Drop the panel on the center square of another panel to arrange the two panels inside a tabbed container.

To make a panel float freely, drag it away from PowerGREP or simply double-click its caption or tab. Floating a panel is very useful if your computer has more than one monitor. Move the floating panel to your second monitor to take full advantage of your multi-monitor system. If you drag a second panel onto the floating panel, you can dock both panels together in a single floating container. This way you can conveniently display several panels on the second monitor.



View Assistant

Show the PowerGREP Assistant. In the default view, the PowerGREP Assistant is visible along the bottom of the PowerGREP window. The assistant displays helpful hints as well as error messages while you work with PowerGREP.



View File Selector

Show the File Selector. In the default view, the File Selector is visible along the left side of the PowerGREP window. The File Selector displays a tree of folders and files, and enables you to select which files PowerGREP will work on.



View Action

Show the Action panel where you define the action that PowerGREP will execute. In the default view, you can access the Action panel by clicking on the Action tab near the top of the PowerGREP window.



View Sequence

Show the Sequence panel where you define a sequence of actions for PowerGREP to execute. In the default view, you can access the Sequence panel by clicking on the Sequence tab near the top of the PowerGREP window.



View Library

Show the Library panel where you can store PowerGREP actions for later reuse. In the default view, you can access the Library panel by clicking on the Library tab near the top of the PowerGREP window.



View Results

Show the Results panel where PowerGREP displays detailed results after executing an action. In the default view, you can access the Results panel by clicking on the Results tab near the top of the PowerGREP window.



View Editor

Show or hide PowerGREP's built-in file editor. With the editor you can edit any kind of text or binary file. The editor also highlights matches if the file was searched through during the last action you executed. In the default view, you can access the Editor panel by clicking on the Editor tab near the top of the PowerGREP window.



View Undo History

Show the Undo History. The Undo History keeps track of all actions that overwrote one or more files. With the Undo History you can undo an action by restoring all overwritten files from their backup copies. You can also delete backup files that are no longer needed. In the default view, you can access the Undo History by clicking on the Undo History tab near the top of the PowerGREP window.



View Forum

Show the PowerGREP forum where you can discuss PowerGREP and regular expressions with other PowerGREP users, and obtain technical support from Just Great Software.

Large Toolbar Icons

If you have a high resolution monitor, the icons on PowerGREP's various toolbars may be too small to discern properly. Select Large Toolbar Icons in the View menu to make them 50% larger. Select the same command to restore the toolbar icons to their default size.

Lock Toolbars

PowerGREP's toolbars can be docked anywhere and made to float by dragging them with the mouse. If you find yourself doing this by accident, you can lock the toolbars in place by turning on the Lock Toolbars item in the View menu.

Office 2003 Display Style

If you find PowerGREP's looks a bit bland, select Office 2003 Display Style from the View menu to make PowerGREP mimic the looks of Microsoft Office 2003. This will make PowerGREP rather colorful. Select the item again to restore the default looks. On Windows XP, the default looks will use the Windows XP theme you selected in your computer's display settings.



Restore Default Layout

Use the Restore Default Layout item in the View menu to quickly reset the PowerGREP window to its default layout, with the File Selector docked to the left, the Assistant docked to the bottom, and the other panels arranged in tabs.

This layout keeps each panel sufficiently large to be workable on a computer with a single medium resolution monitor.

Side by Side Layout

The side by side layout arranges the panels into four columns. The File Selector has its own space at the left, the Action, Sequence, and Editor panels are docked together as the second column, the Results, Library, Undo History, and Forum panels are combined into tabs as the third column, and the Assistant is permanently visible at the right.

If you have a large resolution widescreen monitor, use this layout to work more comfortably by keeping panels that are often used in combination visible at the same time, such as Action/Library and

Editor/Results. The columns optimally take advantage of the extra screen width. If you don't use the Sequence panel, close it after choosing this layout to have more space for the Action panel.

Dual Monitor Tabbed Layout

This option is only available if your computer has more than one monitor. Use this layout if your computer has two average resolution monitors to take advantage of the second monitor. It arranges the Action, Library, Editor, and Undo History panel in tabs, with the File Selector and Assistant docked to the left and the right. The Sequence, Results, and Forum panels are arranged in a floating tabbed window that is automatically placed on the other monitor (versus the one PowerGREP's main window is on).

Dual Monitor Side by Side Layout

This option is only available if your computer has more than one monitor. Use this layout if your computer has two high resolution monitors to put your computer's screen size to maximum use. This layout keeps all the frequently used panels visible at all times. It arranges the File Selector, Action, Sequence, and Assistant panels side by side. The Results and Editor panels are arranged side by side in a floating window that is automatically placed on the other monitor (versus the one PowerGREP's main window is on). The Library, Undo History, and Forum panels are tabbed with the Sequence panel.

Custom Layouts

If you don't like the predefined layouts, you can freely rearrange the panels as described at the top of this chapter. Then use the Save Layouts item in the Custom Layouts submenu of the View menu to give your layout a name. You can then restore this layout at any time by selecting it from the Custom Layouts menu. You can add as many layouts as you like and switch between them at any time.

28. Share Experiences and Get Help on The User Forums

When you click the Login button you will be asked for a name and email address. The name you enter is what others will see when you post a message to the forum. It is polite to enter your real, full name. The forums are private, friendly and spam-free, so there's no need to hide behind a pseudonym. While you can use an anonymous handle, you'll find that people (other PowerGREP users) are more willing to help you if you let them know who you are. Support staff from Just Great Software will answer technical support questions anyhow.

The email address you enter will be used to email you whenever others participate in one of your discussions. The email address is never displayed to anyone, and will never be used for anything other than the automatic notifications. PowerGREP's forum system does not have a function to respond privately to a message. If you don't want to receive automatic email notifications, there's no need to enter an email address.

If you select "never email replies", you'll never get any email. If you select "email replies to conversations you start", you'll get an email whenever somebody replies to a conversation that you started. If you select "email replies to conversations that you participate in", you'll get an email whenever somebody replies to a conversation that you started or replied to. The From address on the email notifications will be forums@jgsoft.com. You can filter the messages based on this address in your email software.

PowerGREP's forum system uses the standard HTTP protocol which is also used for regular web browsing. If your computer is behind an HTTP proxy or SOCKS proxy, click the Proxy button to configure the proxy connection.

If you prefer to be notified of new messages via an RSS feed instead of email, log in first. After PowerGREP has connected to the forums, you can click the Feeds button to select RSS feeds that you can add to your favorite feed reader.

Various Forums

Below the Login button, there's a drop-down list where you can select which particular forum you want to participate in. The "Just Great Software news" forum is read-only. Announcements about new PowerGREP releases and other Just Great Software products will appear there.

The "PowerGREP discussion" forum is for discussing anything related to the PowerGREP software itself. This is the place for technical support questions, feature requests and other feedback regarding the functionality and use of PowerGREP.

The "regular expressions" forum is for discussing regular expressions in general. Here you can talk about creating regular expressions for particular tasks, and exchange ideas on how to implement regular expressions with whatever application or programming language you work with.

Searching The Forums

Before starting a new conversation, please check first if there's already a conversation going on about your topic. In the top right corner of the Forum pane, there's an edit box on the toolbar that you can use to search for messages. When you type something into that box, only conversations that include at least one message

containing the word or phrase you typed in will be shown. The filtering happens in real time as you type in your word or phrase.

Note that you can enter only one search term, which will be searched for literally. If you type “find me”, only conversations containing the two words “find me” next to each other and in that order will be shown. You cannot use boolean operators like “or” or “and”. Since the filtering is instant, you can quickly try various keywords.

If you find a conversation about your subject, start with reading all the messages in that conversation. If you have any further comments or questions on that conversation, reply to the existing conversation instead of starting a new one. That way, the thread of the conversation will stay together, and others can instantly see what you’re talking about. It doesn’t matter if the conversation is a year old. If you reply to it, it will move to the top automatically.

Conversations and Messages

The left hand half of the Forum pane shows two lists. The one at the top shows conversations. The bottom one shows the messages in the selected conversation. You can change the order of the conversations and messages by clicking on the column headers in the lists. A conversation talks about one specific topic. In other forums, a conversation is sometimes called a thread.

If you want to talk about a topic that doesn’t have a conversation yet, click the New button to start a new conversation. A new entry will appear in the list of conversations with an edit box. Type in a brief subject for your conversation (up to 100 characters) and press Enter. Please write a clear subject such as “scraping an HTML table in Perl” rather than “need help with HTML” or just “help”. A clear subject significantly increases the odds that somebody who knows the answer will actually click on your conversation, read your question and reply. A generic scream for help only gives the impression you’re too lazy to type in a clear subject, and most forum users don’t like helping lazy people.

After typing in your subject and pressing Enter, the keyboard focus will move to the box empty box where you can enter the body text of your message. Please try to be as clear and descriptive as you can. The more information you provide, the more likely you’ll get a timely and accurate answer. If your question is about a particular regular expression, don’t forget to attach your regular expression or test data. Use the forum’s attachment system rather than copying and pasting stuff into your message text.

If you want to reply to an existing conversation, select the conversation and click the Reply button. It doesn’t matter which message in the conversation you selected. Replies are always to the whole conversation rather than to a particular message in a conversation. PowerGREP doesn’t thread messages like newsgroup software tends to do. This prevents conversations from veering off-topic. If you want to respond to somebody and bring up a different subject, you can start a new conversation, and mention the new conversation in a short reply to the old one.

When starting a reply, a new entry will appear in the list of messages. Type in a summary of your reply (up to 100 characters) and press Enter. Then you can type in the full text of your reply, just like when you start a new conversation. However, doing so is optional. If your reply is very brief, simply leave the message body blank. When you send a reply without any body text, the forum system will use the summary as the body text, and automatically prepend [nt] to your summary. The [nt] is an abbreviation for “no text”, meaning the summary is all there is. If you see [nt] on a reply, you don’t need to click on it to see the rest of the message.

This way you can quickly respond with “Thank you” or "You're welcome" and other brief courtesy messages that are often sadly absent from online communication.

When you're done with your message, click the Send button to publish it. There's no need to hurry clicking the Send button. PowerGREP will forever keep all your messages in progress, even if you close and restart PowerGREP, or refresh the forums. Sometimes it's a good idea to sleep on a reply if the discussion gets a little heated. You can have as many draft conversations and replies as you want. You can read other messages while composing your reply. If you're replying to a long question, you can switch between the message with the question and your reply while you're writing.

Directly Attach PowerGREP Actions and Other Files

One of the greatest benefits of PowerGREP's built-in forums is that you can directly attach file selections, actions, sequences, libraries, and results. Simply click the Attach button and select the item you want to attach. PowerGREP automatically copies the settings from the relevant panel in PowerGREP into the attachment. You can add the same item more than once. E.g. if you attach your action, then make some changes on the Action panel, and select to attach the action again, your message will have two PowerGREP Action attachments, each storing the settings on the Action panel as you had it when you added the attachment.

To attach a screen shot, press the Print Screen button on the keyboard to capture your whole desktop. Or, press Alt+Print Screen to just capture the active window (e.g. PowerGREP's window). Then switch to the Forum tab, click the Attach button, and select Clipboard. You can also attach text you copied to the clipboard this way.

It's best to add your attachments while you're still composing your message. The attachments will appear with the message, but won't be uploaded until you click the Send button to post your message. If you add an attachment to a message you've written previously, it will be uploaded immediately. You cannot attach anything to messages written by others. Write your own reply, and attach your data to that.

To check out an attachment uploaded by somebody else, click the Use or Save button. The Use button loads the attachment directly into PowerGREP. This may replace your own data. E.g. the Action panel can hold only one action definition. Selecting a PowerGREP Action attachment and clicking Use replaces everything you have on the Action panel with the settings from the attachment. If you click the Save button, PowerGREP prompts for a location to save the attachment. PowerGREP does not automatically open attachments you save.

PowerGREP automatically compresses attachments in memory before uploading them. So if you want to attach an external file, there's no need to compress it using a zip program first. If you compress the file manually, everybody who wants to open it will have to decompress it manually. If you let PowerGREP compress it automatically, decompression will also be automatic.

Taking Back Your Words

If you regret anything you wrote, simply delete it. There are three Delete buttons. The one above the list of conversations deletes the whole conversation. You can only delete a conversation if nobody else participated in it. The Delete button above the edit box for the message body deletes that message, if you wrote it. The

Delete button above the list of attachments deletes the selected attachment, if it belongs to a message that you wrote.

If somebody already downloaded your message before you got around to deleting it, it won't vanish magically. The message will disappear from their view of the forums the next time they log onto the forums or click Refresh. If you see messages disappear when you refresh your own view of the forums, that means the author of the message deleted it. If you replied to a conversation and the original question disappears, leaving your reply as the only message, you should delete your reply too. Otherwise, your reply will look silly all by itself. When you delete the last reply to a conversation, the conversation itself is also deleted, whether you started it or not.

Changing Your Opinion

If you think you could better phrase something you wrote earlier, select the message and then click the Edit button above the message text. You can then edit the subject and/or body text of the message. Click the Send button to publish the edited message. It will replace the original. If you change your mind about editing the message, click the Delete button. Make sure to click it only once! When editing a message, clicking Delete will revert the message to what it was before you started editing it. If you click Delete a second time (i.e. while the message is no longer being edited), you'll delete the message from the forum.

If other people have already downloaded your message, their view of the message will magically change when they click Refresh or log in again. Since things may get confusing if people respond to your original message before they see the edited message, it's best to restrict your edits to minor errors like spelling mistakes. If you change your opinion, click the Reply button to add a new message to the same conversation.

Updating Your View

When you click the Login button, PowerGREP automatically downloads all new conversations and message summaries. Message bodies are downloaded one conversation at a time as you click on the conversations. Attachments are downloaded individually when you click the Use or Save button.

PowerGREP keeps a cache of conversations and messages that persists when you close PowerGREP. Attachments are cached while PowerGREP is running, and discarded when you close PowerGREP. By caching conversations and messages, PowerGREP improves the responsiveness of the forum while reducing the stress on the forum server.

If you keep PowerGREP running for a long time, PowerGREP will not automatically check for new conversations and messages. To do so, click the Refresh button.

Whenever you click Login or Refresh, all conversations and messages will be marked as "read". They won't have any special indicator in the list of conversations or messages. If the refresh downloads new conversation and message summaries, those will be marked "unread". This is indicated with the same "people" icon as shown next to the Login button.

29. Forum RSS Feeds

When you're connected to the user forum, you can click the Feeds button to select RSS feeds that you can add to your favorite feed reader. This way, you can follow PowerGREP's discussion forums as part of your regular reading, without having to start PowerGREP. To participate in the discussions, simply click on a link in the RSS feed. All links in PowerGREP's RSS feeds will start PowerGREP and present the forum login screen. After you log in, wait a few moments for PowerGREP to download the latest conversations. PowerGREP will automatically select the conversation or message that the link points to. If PowerGREP was already running and you were already logged onto the forums, the conversation or message that the link points to is selected immediately.

You can choose which conversations should be included in the RSS feed:

- All conversations in all groups: show all conversations in all the discussion groups that you can access in PowerGREP.
- All conversations in the selected group: show the list of conversations that PowerGREP is presently showing on the Forum tab.
- All conversations you participated in: show all conversations that you started or replied to in all the discussion groups that you can access in PowerGREP.
- All conversations you started: show all conversations that you started in all the discussion groups that you can access in PowerGREP.
- Only the selected conversation: show only the conversation that you are presently reading on the Forum tab in PowerGREP.

In addition, you can choose how the conversations that you want in your RSS feed should be arranged into items or entries in the feed:

- One item per group, with a list of conversations: Entries link to groups as a whole. The entry titles show the names of the groups. The text of each entry shows a list of conversation subjects and dates. You can click the subjects to participate in the conversation in PowerGREP. Choose this option if you prefer to read discussions in PowerGREP itself (with instant access to attachments), and you only want your RSS reader to tell you if there's anything new to be read.
- One item per conversation, without messages: Entries link to conversations. The entry titles show the subjects of the conversations. The entry text shows the date the conversation was started and last replied to. If your feed has conversations from multiple groups, those will be mixed among each other.
- One item per conversation, with a list of messages: Entries link to conversations. The entry titles show the subjects of the conversations. The entry text shows the list of replies with their summaries, author names, and dates. You can click a message summary to read the message in PowerGREP. If your feed has conversations from multiple groups, those will be mixed among each other.
- One item per conversation, with a list of messages: Entries link to conversations. The entry titles show the subjects of the conversations. The entry text shows the list of replies, each with their full text. If your feed has conversations from multiple groups, those will be mixed among each other. This is the best option if you want to read full discussions in your RSS reader.
- One item per message with its full text: Entries link to messages (responses to conversations). The entry titles show the message summary. The entry text shows the full text of the reply, and the conversation subject that you can click on to open the conversation in PowerGREP. If your feed lists multiple conversations, replies to different conversations are mixed among each other. Choose this option if you want to read full discussions in your RSS reader, but your RSS reader does not mark old entries as unread when they are updated in the RSS feed.

30. File Selector Preferences

In the File Selector section of the Preferences screen, you can configure PowerGREP's File Selector and the way PowerGREP handles files, folders and the network.



Remember File Selection and Action between PowerGREP Sessions

Turn on “remember the File Selection and Action themselves” if you often continue working with the same files and/or action definition when restarting PowerGREP. PowerGREP will then automatically store the file selection and action definition when you close it, and reload it when you start PowerGREP. If you select "new instance" from the PowerGREP menu, the new instance will take over the file selection and action definition.

Alternatively, you can turn on “remember the names of the File Selection and Action files last opened”. Then PowerGREP will not save the actual file selection and action definition. Instead, PowerGREP will remember the file selection file and the action file you last opened. The next time you start PowerGREP, it will reload those files.

There's a clear difference between the two above options when you open an action file, make some changes to it, and close PowerGREP without saving the action file. If you select “remember the File Selection and Action themselves”, PowerGREP will reload the modified action. If you select “remember the names of the File Selection and Action files last opened”, PowerGREP will reload the original action file.

If you select “do not remember the File Selection or Action”, PowerGREP will not automatically remember the file selection and action definition. New instances will start out with a blank file selection and action definition. Choose this option if you usually don’t continue working with the last set of files or action. Then you don’t have to manually clear the file selection and action definition each time.

File Selector

Turn on "search through hidden files and folders" to make hidden files and folders visible in the File Selector. This option also affects system files and folders. PowerGREP completely ignores hidden and system files and folders unless you turn this option on.

Turn on "automatically get a list of network servers and shares" if you want PowerGREP to show all network servers it can find when you expand the Network node in the File Selector. You may want to turn off this option if your computer is connected to a very large network. A long list of servers clutters the File Selector.

When you turn off the option to automatically scan the network, you can still access the network by directly typing in a UNC path in the Path field in the File Selector. E.g. to access the network share “share” on the server “server”, type \\server\share. That share will then appear under the Network node in the File Selector until you close PowerGREP.

When automatically scanning the network, PowerGREP can either search the whole network, or only the servers that are on the same Windows workgroup or domain as your computer. If you usually only access computers in your own workgroup or domain, you should turn on this option. You can still access servers outside your computer’s domain by typing in a UNC path such as \\server\share.

Files and Folders Excluded from All Actions

In the box labeled “files excluded from all actions” you can enter a semicolon-delimited list of file masks or regular expressions, just like you do in the File Selector. All files of which the names match one of these file masks will be completely invisible to PowerGREP. They will not appear in the File Selector, and will never be searched through. The purpose of making files invisible this way is to reduce clutter in the File Selector.

By default, the permanent exclusion masks are set to *.bak;*.~*;Backup*;Working copy* which match all backup files created by PowerGREP, as well as temporary working copies of files created by EditPad Pro. If you’d ever need to search through backup files or working copies, you will need to adjust the permanent exclusion file masks first.

The box “folders excluded from all actions” works in the same way, except that the file masks you specify are tested against individual folder names. Folder names listed here will be completely invisible to PowerGREP. By default only __history is excluded. This is the folder name that PowerGREP uses for backup files when you select the “hidden history” backup option.

Exclude Files and Folders Using Regular Expressions instead of File Masks

Turn on to use a semicolon-delimited list of regular expressions to permanently exclude files. Turn off to use wildcards to permanently exclude files.

Regular expressions are matched partially. E.g. the regular expression “joe” will exclude all files that contain “joe” anywhere in the file name. The equivalent file mask using wildcards is “*joe*”. When using wildcards, “joe” excludes files exactly named “joe” only.

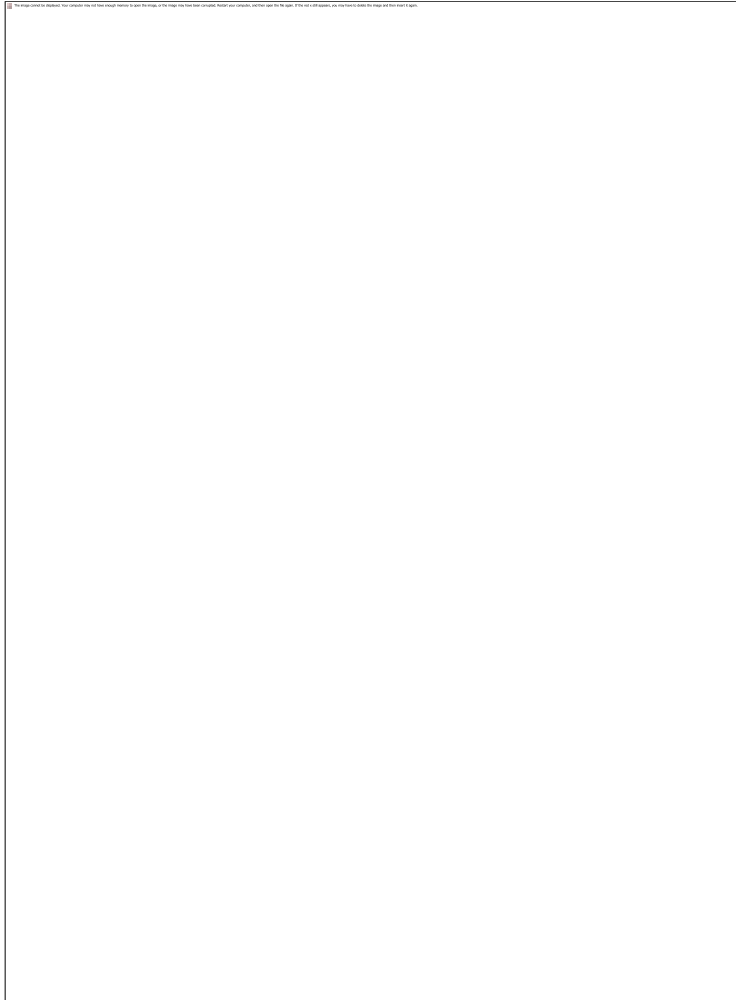
Folder to Use for Open and Save Dialog Boxes

When you invoke a command to open or save a file, PowerGREP will show a list of files. If you previously opened or saved a file, the file list will show the folder containing that file. If not, the file list will show the folder you configured in the Operation Preferences.

Select “the most recently used folder” to make the file list show the folder you most recently opened or saved a file from in PowerGREP, even when you don’t have a file open. Select “my documents folder” to make the file list default to a specific folder. By default, this is your Windows “My Documents” folder, but you can select any folder you like.

31. File Formats Preferences

PowerGREP supports a number of proprietary file formats. You can configure how PowerGREP should deal with them in the File Formats section of the Preferences screen.



Proprietary File Formats

For each file format supported by PowerGREP, you can specify a semicolon-delimited list of file masks. Files that match a format's file mask will be considered to be encoded in that proprietary format. The default settings list the default extensions used by the software that creates files in these formats. If you use non-standard extensions, you can specify them here.

Note that if you don't specify any file masks for a particular file format, PowerGREP will never recognize any files as using that format. This is not a good idea, since it makes it harder to enable the format again later. Turn on the option "search through raw binary data" instead. PowerGREP will then ignore the fact that it can decode the file format, and process the file as if it used an unsupported proprietary file format.

Decode and Convert to Text Prior to Searching

Decode the file format and extract the text from the files. Search through the decoded textual representation of the files.

Use IFilter, If Available, Instead of PowerGREP's Built-in Decoder

PowerGREP can decode all the proprietary file formats in the list above without the help of an external program or IFilter. If you have an IFilter installed for a particular file format, and you find that the IFilter gives better results than PowerGREP's built-in decoder, you can turn on this option to use the IFilter instead. If you turn on this option for a format for which you don't have an IFilter, PowerGREP will fall back to its built-in decoder.

Whether the IFilter is available not only depends on whether you have one installed, but also depends on the IFilter file masks below. If the IFilter file masks tell PowerGREP not to use the IFilter for a file, then PowerGREP will use its built-in decoder, regardless of the option to prefer the IFilter over the built-in decoder.

Search Through Raw Binary Data When a File Cannot Be Decoded

This checkbox determines what happens when a file cannot be decoded by the IFilter or by PowerGREP's built-in decoder. Turn on to search through the undecoded binary data of files that cannot be decoded. Turn off to skip files that cannot be decoded.

Search through Raw Binary Data

Do not decode the files at all. Search through them as binary files of an unspecified format if the Search Binary Files option is turned on in the File Selector.

Search through The File's Raw XML Content

This option appears for file formats that are technically ZIP archives containing XML files and other files. Select this option to search through the file as if it were a ZIP archive. You should only do this if you are familiar with the XML schema used by this file format. You will be able to expand the file's node in the File Selector and access the XML files inside. The action will process the XML files and other files in the document as if they were separate files. The file will still be considered a document, so it will still be searched even if you turn off Search Archives in the File Selector.

Always Skip

Never search through files that use this proprietary file format.

IFilter

Many other applications use proprietary file formats for which PowerGREP does not have built-in decoders. PowerGREP can decode such files if the application that uses them provides an IFilter, or if you have installed a 3rd party IFilter that supports the file formats you work with.

Decode Binary Files Using IFilter

The search technology built into Windows provides a mechanism known as IFilter that applications can use to provide plain text versions of their proprietary file formats. PowerGREP is also capable of using the IFilters provided by other applications.

Turn on this option if you want PowerGREP to be able to search through files in proprietary formats for which it does not have built-in decoders, and for which you have software installed that provides an IFilter. If you have specified IFilter file masks, the IFilter is only used for binary files passing both "only use..." and "never use..." tests. The decoding result and speed will greatly depend on that software.

Turn off this option if you do not want your search results to depend on any other software that may or may not be installed on this computer. Some IFilters may be very slow or not return nicely formatted text.

Transform Plain Text Files Using IFilter

Plain text files include web pages, source code, XML files, and any other file that shows readable text in Notepad. PowerGREP can search those files directly. No decoding is needed.

Though plain text files can be searched without IFilter, Windows Search does include IFilters for many text file formats. E.g. the IFilter for HTML strips out HTML tags. This is useful for search engines that shouldn't index HTML tags.

Turn on this option if you want PowerGREP to display and search through plain text files the way Windows Search does. Formatting codes like HTML tags will be stripped. You will not be able to make changes to the files. If you have specified IFilter file masks, the IFilter is only used for text files passing both "only use..." and "never use..." tests.

Turn off this option to make sure that PowerGREP does not use IFilter for plain text files. This allows you to search through the raw content of plain text files, and allows you to search-and-replace through these files.

Turning off this option is strongly recommended.

Only Use IFilter for Files Matching These File Masks

If you specify one or more file masks here, PowerGREP will only use IFilter for files that match one of these file masks.

If a file matches one of these file masks, but also matches one of the file masks that you never want to use IFilter for, then the IFilter is not used. If you have chosen to use the IFilter for a particular proprietary file

format, and you have specified file masks here that do not include the file mask for that proprietary file format, then the IFilter is not used.

By default, PowerGREP uses the IFilter only for PowerPoint and MS Word documents. PowerGREP cannot decode PowerPoint documents on its own. While PowerGREP can decode MS Word documents, it cannot handle MS Word documents with Unicode content. You can add additional file masks if you get good results searching those files with Windows Search but not with PowerGREP.

Never Use IFilter for Files Matching These File Masks

If you specify one or more file masks here, PowerGREP will never use IFilter for files that match one of these file masks, even if they match one of the file masks you specified for "Only use IFilter...".

PDF and Excel files are excluded by default because PowerGREP's built-in decoder is much more reliable. Adobe's IFilter for PDF files has bugs that may cause PowerGREP to crash. Microsoft's IFilter for Excel files does not preserve the layout of rows and columns like PowerGREP's built-in decoder.

Conversion Cache

Decoding proprietary file formats is quite CPU-intensive. Usually, decoding the file takes several times longer than actually searching through it. Therefore, PowerGREP keeps a cache of the decoded files. If you search through the file a second time, and the file is in the cache, PowerGREP can skip the decoding step and simply search through the cached copy of the file. If you run the same search twice, and all decoded files are still in the cache, you'll notice that the second search runs many times faster than the first one. This is particularly handy when fine-tuning search terms or narrowing down search results with successive searches. The cache is shared by all instances of PowerGREP that you run on your computer. If you start multiple instances, you can run multiple searches on the same set of files simultaneously.

By default, PowerGREP will store its cache in the folder that Windows designates for applications to store temporary files. You can select a different folder if you like. If you select a different folder on the same drive, PowerGREP will move its cache. If you select a folder on another drive, PowerGREP will clear its cache. You should select a folder on a local hard disk drive.

While PowerGREP can share the cache with other instances running on your own computer, it cannot share the cache with PowerGREP instances running on other computers. Though PowerGREP will store its cache on a network drive if you tell it to, it is up to you to make sure that only one computer will access the cache at any time. If two people using PowerGREP on their own computers access the cache at the same time, the cache will become corrupted.

The cache must be large enough to store all the files you're working with. If only half the files you're searching through fit into the cache, then the second half of the files will be cached when the search completes. When PowerGREP runs out of cache space, it will first discard the cached conversions of the files you least recently search through. If you search through 10 files of equal size, and only 5 fit into the cache, file 6 will cause file 1 to be discarded, file 7 discards file 2, etc. If you then repeat the search, file 1 will discard file 6, etc. In this situation, PowerGREP will never actually read any of the files from the cache. PowerGREP does not change the order in which it searches files depending on whether they're cached or not.

If the cache is large enough, all files will be in the cache when the search completes. If you repeat the search, PowerGREP will be able to read all the files from the cache.

You can specify how many megabytes of disk space you allow PowerGREP to use for its cache. The default is 1024 megabytes which equals one gigabyte. You can make the running cache as large as you like, within the amount of free space on your hard disk. If your hard disk has 100 GB of free space, and you want PowerGREP to repeatedly search a big file server overnight, you can set aside 90,000 MB for the cache and have plenty of room to spare on your hard disk.

The “running” size is used as long as at least one PowerGREP instance is running on your computer. When you close the last instance, PowerGREP will trim the cache down to the “not running” size, if you set it smaller than the “running” size. If you tend to work with the same set of files over and over, you should set the “not running” cache to be the same as the “running” cache. Then all converted copies will still be available the next day, and your searches will run at full speed. If you have limited free hard disk space and have other applications that use a lot of temporary disk space, trimming the cache is probably a good idea.

If you set the “not running” cache size to zero, PowerGREP will clear the cache entirely when you close the last instance. If you set the “running” size to zero, PowerGREP will never cache any files in the first place. You should only disable the cache if you simply don’t have enough disk space to cache all converted files, or if you never search the same file twice.

Only files in proprietary formats are cached. These are the files that match any of the file masks you’ve specified in the File Formats Preferences, and for which you’ve selected the “decode and convert to text prior to searching” option. Files converted using an IFilter are also cached. When estimating the required size of the cache, you only need to count the files in proprietary formats that PowerGREP decodes or that are converted using an IFilter.

If you run PowerGREP silently to execute actions in an automated fashion, you can use the `/nocache` command line parameter to disable the conversion cache for actions that are executed silently. That way automated actions don’t clutter up the conversion cache.

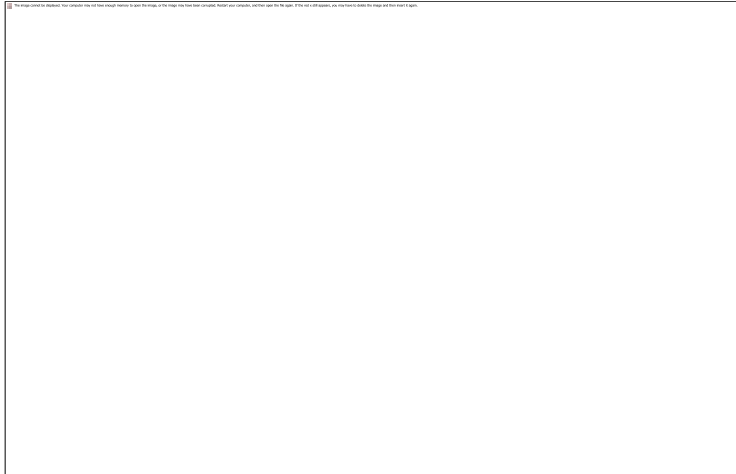
PowerGREP Conversion Manager

If you run more than one PowerGREP instance at the same time, the conversion cache is automatically shared between all instances. PowerGREP’s conversion manager handles this in the background. While one or more instances of PowerGREP is running, an application called `PowerGREPConversionManager.exe` will be running in the background. When you close the last PowerGREP instance, the conversion manager closes automatically.

If you set both the “running” and “not running” cache sizes to zero, then the conversion manager is not used at all.

32. Archive Formats Preferences

PowerGREP supports a wide range of compressed file formats. You can configure how PowerGREP should deal with them in the Archive Formats section of the Preferences screen.



Compressed File Formats Supported by PowerGREP

These are the compressed file formats or archive formats that PowerGREP can decompress. You can search through files in all these formats. PowerGREP can also create or update some of these archives too. Select the format you want to configure.

Formats that PowerGREP can decompress, create, and update:

- **ZIP archives:** ZIP format used by PKZip, WinZIP, and a host of other compression utilities. This is the most popular archive format. PowerGREP uses the original Deflate algorithm for maximum compatibility when creating ZIP archives.
- **ZIPX archives:** Same as the ZIP format. The .zipx extension is used to indicate that newer compression methods are used. PowerGREP uses the PPMd algorithm when creating ZIPX files.
- **7-zip archives:** 7z format used by 7-zip. This format yields the smallest files of all the archive formats that PowerGREP can create.
- **TAR uncompressed:** Uncompressed tarball (.tar file).
- **TAR GZip archives:** Tarball compressed into a GZip file (.tar.gz or .tgz file).
- **TAR BZip2 archives:** Tarball compressed into a BZip2 file (.tar.bz2 file).
- **TAR XZ archives:** Tarball compressed into an XZ file (.tar.xz file).
- **GZip file:** Single file compressed with GZip
- **BZip2 file:** Single file compressed with BZip2
- **XZ file:** Single file compressed with XZ

Formats that PowerGREP can read and write, but which should be treated as document files, even if they are technically archives:

- **Zipped documents:** Office Open XML (MS Office 2007) and OpenDocument Format (OpenOffice) are technically ZIP archives containing multiple XML and other files.

- **CHM files:** HTML Help files consist of compressed HTML files and other files.

Formats that PowerGREP can decompress only:

- **ARJ archives:** Files compressed with ARJ
- **CAB archives:** Microsoft Cabinet
- **DEB packages:** Debian Linux installation packages
- **FAT images:** Disk images of FAT volumes such as floppy disks
- **ISO and UDF images:** CD and DVD images
- **LHA and LZH archives:** Files compressed with LHARC
- **RAR archives:** Files compressed with WinRAR
- **RPM packages:** Red Hat Linux installation packages
- **WIM images:** Windows Imaging disk images
- **XAR archives:** Files compressed with XAR

File Masks

Enter a list of file masks, delimited by semicolons, that match the files that should be treated as files of the selected compressed file format.

Use IFilter to Decode The Compound Document into Plain Text

If this checkbox is enabled, then the compressed file format you've selected in the list is technically a compressed file holding multiple data files inside it, but conceptually a single document file.

Turn on this option to have PowerGREP use the IFilter installed on your computer to decode files of this type into plain text. Depending on the IFilter you have installed, this will enable PowerGREP to search through the document's text as it would appear in the applications that support these file formats. PowerGREP cannot make changes to files decoded with IFilters. If you don't have an IFilter installed for these files, then PowerGREP ignores this option.

Turn off this option to have PowerGREP decompress the files inside the compressed file, and search through those, as if the file was an ordinary archive. PowerGREP can also modify files in zipped documents this way.

You should turn on this option, unless you are familiar with the file format, and you want to work with the documents at a low level.

Document Files That Use Archive File Formats

Some modern document formats are technically .zip archives, though people see them as documents. Microsoft Office Open XML files (*.docx, *.xlsx), XML Paper Specification files (*.xps) and OpenOffice files using the OpenDocument Format (*.odb, *.odc, *.odf, *.odg, *.odi, *.odm, *.odp, *.ods and *.odt) are all technically .zip archives containing one or more XML files and other supports files (such as image files). However, those XML and image files logically constitute one document, unlike unrelated XML files that you might store in a .zip archive.

Therefore, PowerGREP treats such compressed file formats as documents. They ignore the Search Archives option in the File Selector menu. File masks in the File Selector will affect the document file itself, and not the XML and image files that are technically inside the document. So you can include or exclude MS Word 2007 files from your search by typing *.docx in the Include Files or Exclude Files box in the File Selector. Whether the .docx file is included or excluded determines whether all or none of the XML files inside it will be searched through.

If you turn on the IFilter option, and you have an IFilter installed, then PowerGREP uses the IFilter to create a single file with a plain text representation of the document. That plain text representation is what PowerGREP searches through and what it shows in the Editor if you open the file. Office 2007 includes IFilters for all its file formats. OpenOffice and LibreOffice include IFilters for OpenDocument Format files. We have successfully tested PowerGREP with OpenOffice 4.1.0 and LibreOffice 4.2.4. OpenOffice 3.x.x does not correctly install the IFilter, making it unusable by PowerGREP (or any other application).

If you turn off the IFilter option or no IFilter is available, then PowerGREP searches through the individual XML files that the document consists of and show the XML code in the search results or the built-in editor.

If you work with other file formats that are conceptually documents, but technically zip archives, you can add their extensions to the “zipped documents” file masks in the Archive Format Preferences.

Examples: Search through printable content in Word .docx files, Search through printable content in XPS files and Search through printable content in OpenDocument Format files

Self-Extracting Archives

If the .exe files you’re working with are actually self-extracting archives, you can add *.exe to the file masks of the archive format that was used to create the self-extracting archives. E.g. if you used the WinZIP self-extractor, add *.exe to the file masks for “ZIP archives”. If you used the SFX option in WinRAR or 7-zip, add *.exe to “RAR archives” or “7-zip archives”.

You can add each file mask to only one of the compressed file formats supported by PowerGREP. If you add *.exe to more than one format, only the first format in the list that you added it to will be used for .exe files. Since *.exe matches all executable files, PowerGREP will attempt to open all of them as archives, which can slow down your search.

There is no option to make PowerGREP attempt all the archive formats that support self-extracting archives on each .exe file. Though doing so makes sense for a decompression tool that only needs to open a few archives at a time. But if a PowerGREP search includes thousands of .exe files that aren’t archives, testing them for each archive format would slow down your search significantly.

When working with self-extracting archives in different formats, or if you also want to run binary searches through .exe files that aren’t self-extracting archives, then you need to use more specific file masks. Just like the File Selector, PowerGREP’s preferences support file masks with folder names. If your self-extracting archives created with WinZIP are in a folder called “zips” and those created with WinRAR are in a folder called “rars”, add *zips*.exe to the file masks for ZIP archives and *rars*.exe to the file masks for RAR archives. Then PowerGREP tries to open .exe files in any folder called “zips” or its subfolders as ZIP archives and .exe files in any folder called “rars” or its subfolders as RAR archives. Executable files outside these two folders won’t be treated as archives at all.

Even if your self-extracting archives always use the same file format, if you can use a file mask such as `*zip*.exe` to differentiate archives from regular `.exe` files, by all means do so. This ensures PowerGREP does not waste time needlessly trying to open regular `.exe` files as archives.

Archive Formats Using The Same Extension

You can add each file mask to only one of the compressed file formats supported by PowerGREP. If a file matches the file masks of two archive formats, PowerGREP only tries the first one in the list of archive formats. This can be a problem if some of your archives use the same file extension for different kinds of archives.

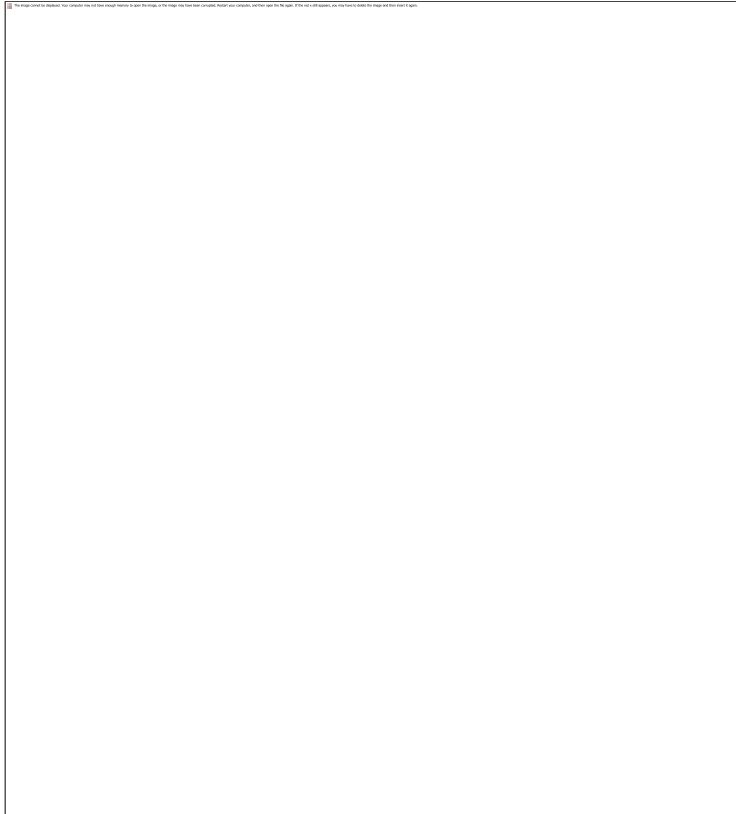
Both Java and the Konquerer web browser use the `.war` extension for "web archives". Java `.war` files are technically `.zip` files while Konquerer `.war` files are technically `.tar.gz` files. PowerGREP's default preferences include the `*.[ejw]ar` file mask for the "ZIP archives" compressed file format. That means it tries to open `.war` files as if they were `.zip` files, which will succeed with Java archives and fail with Konquerer archives.

If all your `.war` files are Konquerer archives, you can easily change the file masks for ZIP archives to `*.zip;*. [ej]ar;*.xpi` and those for TAR GZip to `*.tgz;*.tar.gz;*.war`. If you have a mix of both, you'll need to come up with file masks that differentiate the two.

The file masks that you can use in the preferences are just as flexible as the file masks in the File Selector. If your Java archives are in a folder "java" or subfolders thereof, and your Konquerer archives are in a folder "linux" or subfolders thereof, you can use the file masks `*java*.war` and `*linux*.war` to separate them.

33. Action Preferences

In the Action section in the Preferences screen, you can configure some aspects of the appearance of the Action panel and the way PowerGREP executes actions.



Search Term Editors

The Action panel shows one or more edit boxes for entering search terms. Most of the settings for these edit boxes are combined into a “text layout”. If you have previously configured text layouts in the Results or Editor sections of the preferences, you can select a previously configured text layout from the drop-down list. If not, click the Configure Text Layout button to specify font, text direction, cursor behavior, word selection options, extra spacing, etc.

Visualize Spaces and Tabs

Turn on to visualize spaces as small dots, and tabs as chevrons. Turn off to display spaces and tabs as invisible whitespace.

Visualize Line Breaks

Turn on to show a symbol for each hard line break in the file. This makes it easy to differentiate between permanent line breaks and automatic word wrapping as well as different line break styles. CRLF indicates Windows-style line breaks and LF indicates UNIX-style line breaks.

Action Execution Options

PowerGREP recognizes a number of match placeholders and path placeholders. Match placeholders allow you to insert search matches and search match numbers. Path placeholders are substituted with various parts of the name and path of the file being searched through. If the placeholders conflict with text you're searching for, you can disable them here. PowerGREP will then treat the placeholders as any other literal text.

When PowerGREP creates or modifies a file, PowerGREP will set the last modification date of the new or modified file to the current date and time. If you turn on the option "give target files the same time stamp as the source file", each target file will be given the same last modification date as the source file it is based on. The source file is the file that was searched through by PowerGREP.

The option "delete files to the Windows Recycle Bin" controls how files are deleted when you set the target type of a "list files" action to "delete matching files". When you turn on the option, PowerGREP will try to move the files to the Windows Recycle Bin. Moving files to the recycle bin is a slow operation, and does not reclaim disk space. It does make it possible to recover mistakenly deleted files. If you turn off the option, or if some files cannot be placed into the recycle bin, the files are deleted permanently. Permanently deleted files cannot be recovered, and the disk space they used is reclaimed.

Overwrite Files That Have The Read-Only Attribute Set

Turn off to make PowerGREP respect the read-only attribute. Attempting to overwrite a file with the read-only attribute set will result in an error.

Turn on to make PowerGREP override the read-only attribute. If a target file exists and has the read-only attribute set, PowerGREP will remove the attribute, overwrite the file, and set the attribute again.

This option only applies to the read-only attribute. If a file is read-only because it is locked by another application or because you do not have the security privileges to overwrite the file, then attempting to overwrite it will result in an error regardless of the choice you made for this option. PowerGREP can only remove the read-only attribute if your security privileges allow you to do so.

Multi-Threaded Execution

The settings in this section are important if your computer as a multi-core CPU. All modern computers have dual core CPUs and quad core is becoming increasingly common. PowerGREP can use all the CPU cores your computer has to speed up searches by searching through multiple files at the same time. There is a trade-off though. If you allow PowerGREP to use all your CPU cores, other applications may slow down significantly, particularly if your hard disk can't keep up.

Minimum number of execution threads

If your computer has more than one CPU or a multi-core CPU, you can tell PowerGREP to search through multiple files in parallel. If you set this number higher than 1, PowerGREP will search through as many files as you specified in parallel, even if they are on the same drive. Increase this number to speed up searches using complex regular expressions that are limited by CPU speed rather than by disk or network speed. Decrease this number when running PowerGREP in the background, so it doesn't starve other applications for CPU time. Set this to 1 and turn on the two checkboxes below if most of your actions are simple searches that are limited by disk speed rather than CPU speed, so it doesn't starve other applications for disk access. The maximum setting is the number of CPU cores in your PC. The default setting is one less than that. The default maximizes PowerGREP's performance while making sure other applications and PowerGREP itself remain responsive by leaving one CPU core for other tasks.

Use a Separate Thread for Each Drive Letter

Turn on if the drive letters on your computer represent separate physical disks. When searching through files on multiple drives, PowerGREP will process the drives in parallel to speed up the search. If you have many drive letters, PowerGREP may use more threads than the minimum you specified. Turn off if the drive letters on your computer represent partitions on a single physical disk. Searching through files on different partitions on the same disk in parallel may slow down the search. Mechanical hard disks perform very poorly if they have to access multiple files on different partitions simultaneously. SSD drives (flash-based hard disks) do not have that limitation.

Use a Separate Thread for Each Network Share

Turn on if the servers on your network are slow. When searching through files on multiple network shares, PowerGREP will process the shares in parallel to speed up the search. When searching through files on many shares, PowerGREP may use more threads than the minimum you specified. Turn off if the servers on your network are fast enough to send data to your PC faster than your PC can receive it.

Folder to Use for Temporary Files

PowerGREP uses temporary files for many things. This way PowerGREP can work with arbitrarily large files without running out of memory. If the drive on which Windows is installed (the C: drive) isn't the fastest drive on your computer, tell PowerGREP to use a specific folder on another drive to save its temporary files.

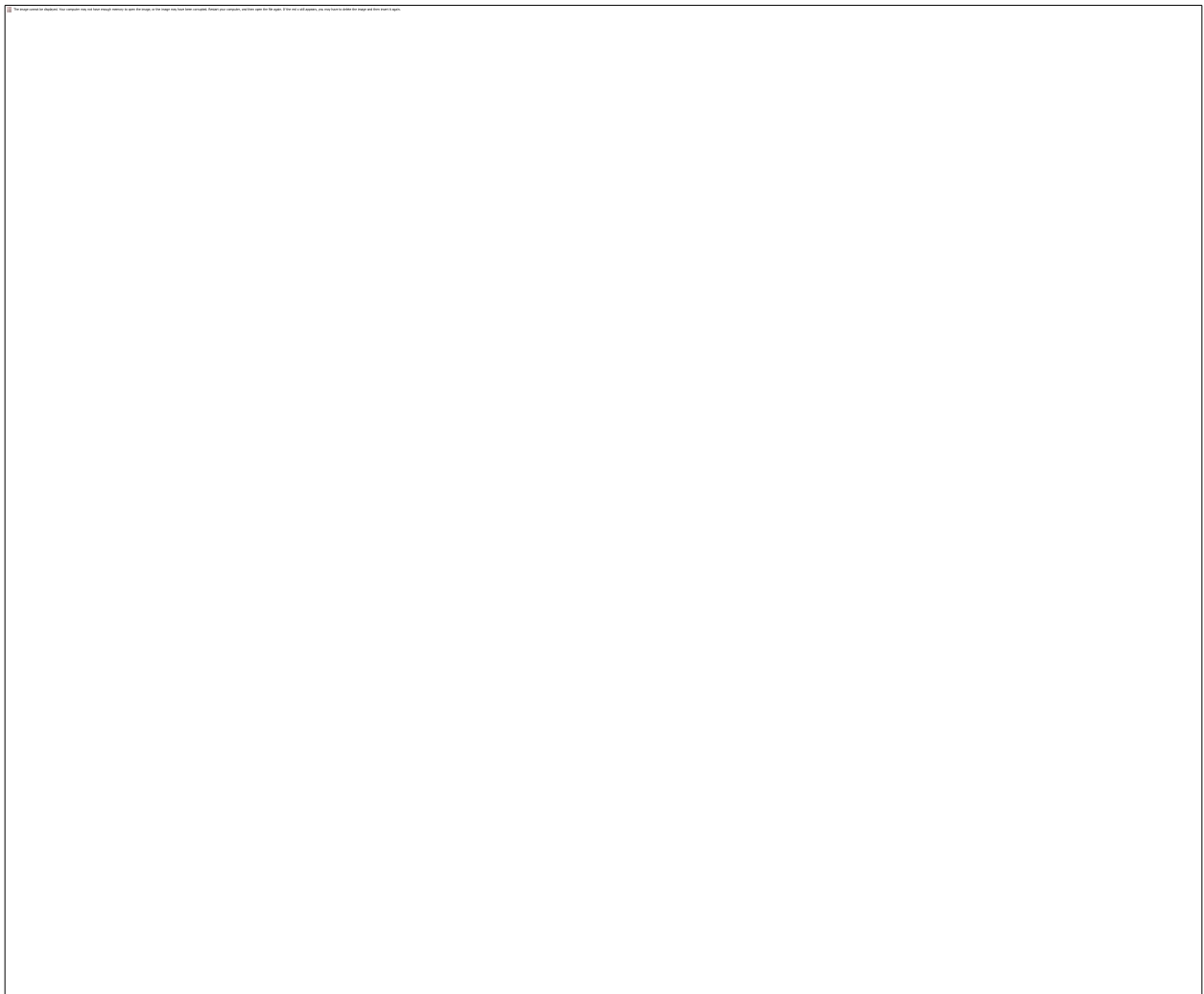
PowerGREP does not use this setting for preparing target files that you want it to save on a local hard disk. Those temporary files are created in the destination folder of the target file. That allows PowerGREP to instantly replace the target with the temporary file instead of having to copy it around.

Smaller temporary files can be kept in memory without saving them to disk. You can choose how much of your PC's RAM PowerGREP is allowed to use for temporary files. Allocating more RAM speeds up actions that need a lot of temporary files, such as searching through compressed archives, but leaves less memory available for other applications. The memory limit is for each instance of PowerGREP. If you run multiple PowerGREP instances at the same time, reduce the limit so your PC has enough actual RAM for all PowerGREP instances.

34. Text Layout Configuration

In PowerGREP, a “text layout” is a combination of settings that control how an edit control displays text and how the text cursor navigates through that text. The settings include the font, text direction, text cursor behavior, which characters are word characters, and how the text should be spaced. You can select a text layout for the edit boxes on the File Selector, Action, Library, and Sequence panels in the Action section of the Preferences or via the Font and Text Direction submenu in the right-click menu of the search term boxes on the Action panel. You can select a different text layout for the Results panel in the Results section of the Preferences or via the Font and Text Direction item in the Results menu. The Editor panel also uses its own text layout, which you can configure in the Editor section of the Preferences or via the Font and Text Direction item in the Editor menu. Finally, you can configure the text layout of the message editor on the user forum in the General section of the Preferences.

Though you can select four different text layouts for different parts of PowerGREP, all four parts offer the same set of preconfigured text layouts. So you can easily make them use the same layout by picking the same preconfigured layout for all of them. You change the preconfigured text layouts via any of the Configure Text Layout buttons in the preferences or the Configure Text Layout item at the bottom of the Font and Text Direction menu items. When you do this the following screen appears.



Select The Text Layout Configuration That You Want to Use

The Text Layout Configuration screen shows the details of the text layout configuration that you select in the list in the top left corner. Any changes you make on the screen are automatically applied to the selected layout and persist as you choose different layouts in the list. The changes become permanent when you click OK. The layout that is selected in the list when you click OK becomes the new default layout.

Click the New and Delete buttons to add or remove layouts. You must have at least one text layout configuration. If you have more than one, you can use the Up and Down buttons to change their order. The order does not affect anything other than the order in which the text layouts configurations appear in selection lists.

PowerGREP comes with a number of preconfigured text layouts. If you find the options on this screen bewildering, simply choose the preconfigured layout that matches your needs, and ignore all the other settings. You can fully edit and delete all the preconfigured text layouts if you don't like them.

- Left-to-right: Normal settings with best performance for editing text in European languages and ideographic languages (Chinese, Japanese, Korean). The default font is monospaced. The layout does respect individual character width if the font is not purely monospaced or if you select another font.
- Proportionally spaced left-to-right: Like left-to-right, but the default font is proportionally spaced.
- Monospaced left-to-right: Like left-to-right, but the text is forced to be monospaced. Columns are guaranteed to line up perfectly even if the font is not purely monospaced. This is the best choice for working with source code and text files with tabular data.
- Monospaced ideographic width: Like monospaced left-to-right, but ASCII characters are given the same width as ideographs. This is the best choice if you want columns of mixed ASCII and ideographic text to line up perfectly.
- Complex script left-to-right: Supports text in any language, including complex scripts (e.g. Indic scripts) and right-to-left scripts (Hebrew, Arabic). Choose this for editing text that is written from left-to-right, perhaps mixed with an occasional word or phrase written from right-to-left.
- Complex script right-to-left: For writing text in scripts such as Hebrew or Arabic that are written from right-to-left, perhaps mixed with an occasional word or phrase written from left-to-right.
- Monospaced complex left-to-right: Like “complex script left-to-right”, but using monospaced fonts for as many scripts as possible. Text is not forced to be monospaced, so columns may not line up perfectly.
- Monospaced complex right-to-left: Like “complex script right-to-left”, but using monospaced fonts for as many scripts as possible. Text is not forced to be monospaced, so columns may not line up perfectly.

Selected Text Layout Configuration

The section in the upper right corner provides a box to type in the name of the text layout configuration. This name is only used to help you identify it in selection lists when you have prepared more than one text layout configuration.

In the Example box you can type in some text to see how the selected text layout configuration causes the editor to behave.

Text Layout and Direction

- Complex script, predominantly left-to-right: Text is written from left to right and can be mixed with text written from right to left. Choose this for complex scripts such as the Indic scripts, or for text in any language that mixes in the occasional word or phrase in a right-to-left or complex script.
- Complex script, predominantly right-to-left: Text is written from right to left and can be mixed with text written from left to right. Choose this for writing text in scripts written from right to left such as Hebrew or Arabic.
- Left-to-right only: Text is always written from left to right. Complex scripts and right-to-left scripts are not supported. Choose this for best performance for editing text in European languages and ideographic languages (Chinese, Japanese, Korean) that is written from left to right without exception.
- Monospaced left-to-right only: Text is always written from left to right and is forced to be monospaced. Complex scripts and right-to-left scripts are not supported. Each character is given the same horizontal width even if the font specifies different widths for different characters. This guarantees columns to be lined up perfectly. To keep the text readable, you should choose a monospaced font.
- ASCII characters with full ideographic width: You can choose this option in combination with any of the four preceding options. In most fonts, ASCII characters (English letters, digits, and punctuation) are about half the width of ideographs. This option substitutes full-width characters for the ASCII characters so they are the same width as ideographs. If you turn this on in combination with “monospaced left-to-right only” then columns that mix English letters and digits with ideographs will line up perfectly.

Text Cursor Movement

- Monodirectional: The left arrow key on the keyboard always moves the cursor to the left on the screen and the right arrow key always moves the cursor to the right on the screen, regardless of the predominant or actual text direction.
- Bidirectional: This option is only available if you have chosen one of the complex script options in the “text layout and direction” list. The direction that the left and right arrow keys move the cursor into depends on the predominant text direction selected in the “text layout and direction” list and on the actual text direction of the word that the cursor is pointing to when you press the left or right arrow key on the keyboard.
 - Predominantly left-to-right: The left key moves to the preceding character in logical order, and the right key moves to the following character in logical order.
 - Actual left-to-right: The left key moves left, and the right key moves right.
 - Actual right-to-left: The actual direction is reversed from the predominant direction. The left key moves right, and the right key moves left.
 - Predominantly right-to-left: The left key moves to the following character in logical order, and the right key moves to the preceding character in logical order.
 - Actual left-to-right: The actual direction is reversed from the predominant direction. The left key moves right, and the right key moves left.
 - Actual right-to-left: The left key moves left, and the right key moves right.

Selection of Words

- Select only the word: Pressing Ctrl+Shift+Right moves the cursor to the end of the word that the cursor is on. The selection stops at the end of the word. This is the default behavior for all Just Great Software applications. It makes it easy to select a specific word or string of words without any extraneous spaces or characters. To include the space after the last word, press Ctrl+Shift+Right once more, and then Ctrl+Shift+Left.
- Select the word plus everything to the next word: Pressing Ctrl+Shift+Right moves the cursor to the start of the word after the one that the cursor is on. The selection includes the word that the cursor was on and the non-word characters between that word and the next word that the cursor is moved to. This is how text editors usually behave on the Windows platform.

Character Sequences to Treat as words

- Letters, digits, and underscores: Characters that are considered to be letters, digits, or underscores by the Unicode standard are selected when you double-click them. Ctrl+Left and Ctrl+Right move the cursor to the start of the preceding or following sequence of letters, digits, or underscores. If symbols or punctuation appear adjacent to the start of a word, the cursor is positioned between the symbol and the first letter of the word. Ideographs are considered to be letters.
- Letters, digits, and symbols: As above, but symbols other than punctuation are included in the selection when double-clicking. Ctrl+Left and Ctrl+Right never put the cursor between a symbol and another word character.
- Everything except whitespace: All characters except whitespace are selected when you double-click them. Ctrl+Left and Ctrl+Right move the cursor to the preceding or following position that has a whitespace character to the left of the cursor and a non-whitespace character to the right of the cursor.
- Words determined by complex script analysis: If you selected the “bidirectional” text cursor movement option, you can turn on this option to allow Ctrl+Left and Ctrl+Right to place the cursor between two letters for languages such as Thai that don’t write spaces between words.

Text Cursor Appearance

Select a predefined cursor or click the Configure button to show the text cursor configuration screen. There you can configure the looks of the blinking text cursor (and even make it stop blinking).

A text layout uses two cursors. One cursor is used for insert mode, where typing in text pushes ahead the text after the cursor. The other cursor is used for overwrite mode, where typing in text replaces the characters after the cursor. Pressing the Insert key on the keyboard toggles between insert and overwrite mode.

Main Font

Select the font that you want to use from the drop-down list. Turn on “allow bitmapped fonts” to include bitmapped fonts in the list. Otherwise, only TrueType and OpenType fonts are included. Using a TrueType or OpenType font is recommended. Bitmapped fonts may not be displayed perfectly (e.g. italics may be clipped) and only support a few specific sizes.

If you access the text layout configuration screen from a print preview, then turning on “allow bitmapped fonts” will include printer fonts rather than screen fonts in the list, in addition to the TrueType and OpenType fonts that work everywhere. A “printer font” is a font built into your printer’s hardware. If you select a printer font, set “text layout and direction” to “left to right only” for best results.

Fallback Fonts

Not all fonts are capable of displaying text in all scripts or languages. If you have selected one of the complex script options in the “text layout and direction” list, you can specify one or more “fallback” fonts. If the main font does not support a particular script, PowerGREP will try to use one of the fallback fonts. It starts with the topmost font at the list and continues to attempt fonts lower in the list until it finds a font that supports the script you are typing with. If none of the fonts supports the script, then the text will appear as squares.

To figure out which scripts a particular font supports, first type or paste some text using those scripts into the Example box. Make sure one of the complex script options is selected. Then remove all fallback fonts. Now you can change the main font and see which characters that font can display. When you’ve come up with a list of fonts that, if used together, can display all of your characters, select your preferred font as the main font. Then add all the others as fallback fonts.

Line and Character Spacing

By default all the spacing options are set to zero. This tells PowerGREP to use the default spacing for the font you have selected.

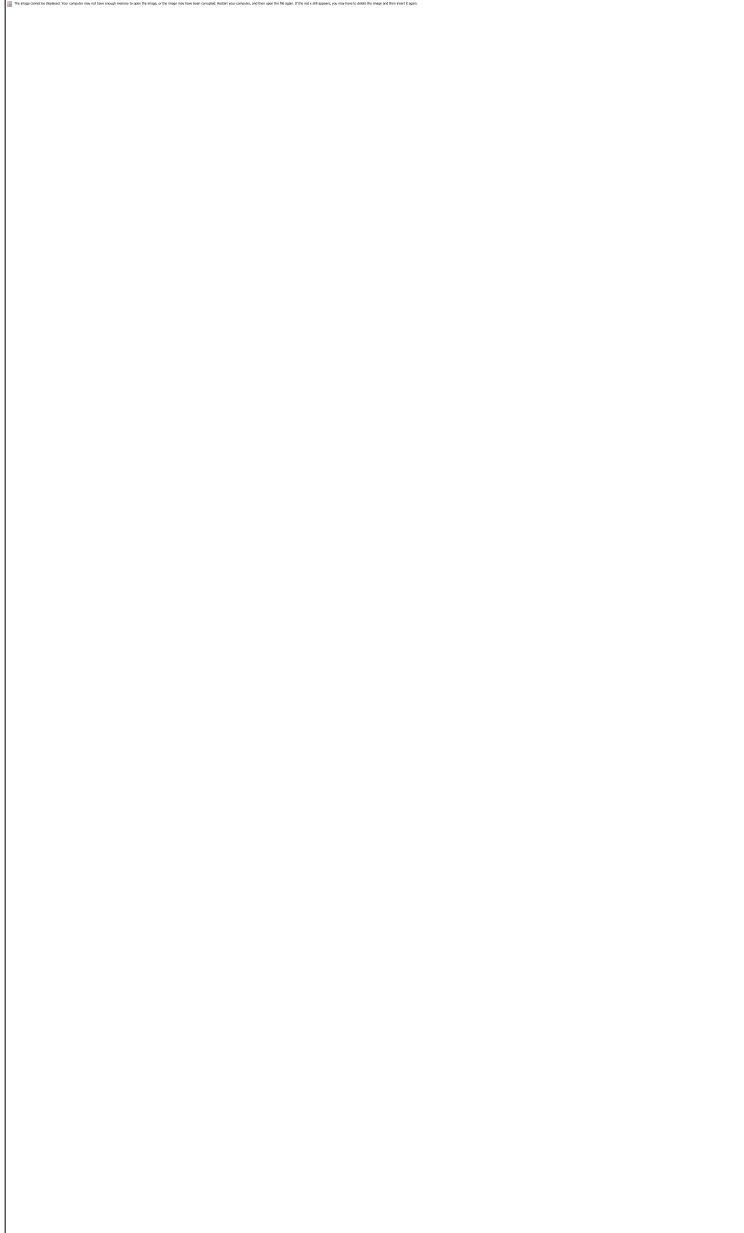
If you find that lines are spaced apart too widely, specify a negative value for “increase (or decrease) the line height”. Set to “add 0 pixels of extra space between lines”.

If you find that lines are spaced too closely together, specify a positive value for “increase (or decrease) the line height” and/or “add ... pixels of extra space between lines”. The difference between the two is that when you select a line of text, increasing the line height increases the height of the selection highlighting, while adding extra space between lines does not. If you select multiple lines of text, extra space between lines shows up as gaps between the selected lines. Adding extra space between lines may make it easier to distinguish between lines.

The “increase (or decrease) the character width by ... pixels” setting is only used when you select “monospaced left-to-right” only in the “text layout and direction” list. You can specify a positive value to increase the character or column width, or a negative value to decrease it. This can be useful if your chosen font is not perfectly monospaced and because of that characters appear spaced too widely or too closely.

35. Text Cursor Configuration

You can access the text cursor configuration screen from the text layout configuration screen by clicking one of the Configure buttons in the “text cursor appearance” section.



Existing Text Cursor Configurations

The Text Cursor Configuration screen shows the details of the text cursor configuration that you select in the list at the top. Any changes you make on the screen are automatically applied to the selected cursor and persist as you choose different cursors in the list. The changes become permanent when you click OK. The cursor that is selected in the list when you click OK becomes the new default cursor.

Click the New and Delete buttons to add or remove cursors. You must have at least one text cursor configuration. If you have more than one, you can use the Up and Down buttons to change their order. The order does not affect anything other than the order in which the text cursor configurations appear in selection lists.

PowerGREP comes with a number of preconfigured text cursors. You can fully edit or delete all the preconfigured text cursors if you don't like them.

- Insertion cursor: Blinking vertical bar similar to the standard Windows cursor, except that it is thicker and fully black, even on a gray background.
- Bidirectional insertion cursor: Like the insertion cursor, but with a little flag that indicates whether the keyboard layout is left-to-right (e.g. you're typing in English) or right-to-left (e.g. you're typing in Hebrew). The flag is larger than what you get with the standard Windows cursor and is shown even if you don't have any right-to-left layouts installed.
- Underbar cursor: Blinking horizontal bar that lies under the character. This mimics the text cursor that was common in DOS applications.
- Overwrite cursor: Blinking rectangle that covers the bottom half of the character. In EditPad this is the default cursor for overwrite mode. In this mode, which is toggled with the Insert key on the keyboard, typing text overwrites the following characters instead of pushing them ahead.
- Standard Windows cursor: The standard Windows cursor is a very thin blinking vertical bar that is XOR-ed on the screen, making it very hard to see on anything except a pure black or pure white background. If you have a right-to-left keyboard layout installed, the cursor gets a tiny flag indicating keyboard direction. You should only use this cursor if you rely on accessibility software such as a screen reader or magnification tool that fails to track any of EditPad's other cursor shapes.

Selected Text Cursor Configuration

Type in the name of the text cursor configuration. This name is only used to help you identify it in selection lists when you have prepared more than one text cursor configuration.

In the Example box you can type in some text to see what the cursor looks like. The box has a word in Latin and Arabic so you can see the difference in cursor appearance, if any, based on the text direction of the word that the cursor is on.

Shape

- Standard Windows Text cursor: The standard Windows cursor is a very thin blinking vertical bar that is XOR-ed on the screen, making it very hard to see on anything except a pure black or pure white background. If you have a right-to-left keyboard layout installed, the cursor gets a tiny flag indicating keyboard direction. You should only use this cursor if you rely on accessibility software such as a screen reader or magnification tool that fails to track any of EditPad's other cursor shapes. The standard Windows cursor provides no configuration options.
- Vertical bar in front of the character: On the Windows platform, the normal cursor shape is a vertical bar that is positioned in front of the character that it points to. That is to the left of the character for left-to-right text, and to the right of the character for right-to-left text.
- Vertical bar with a flag indicating keyboard direction: A vertical bar positioned in front of the character that it points to, with a little flag (triangle) at the top that indicates the direction of the active keyboard layout. When the cursor points to a character in left-to-right text, it is placed to the

left of that character. When the cursor point to a character in right-to-left text, it is placed to the right of that character. The direction of the cursor's flag is independent of the text under the cursor. The cursor's flag points to the right when the active keyboard layout is for a left-to-right language. The cursor's flag points to the left when the active keyboard layout is for a right-to-left language.

- Vertical bar with a flag indicating text direction: A vertical bar positioned in front of the character that it points to, with a little flag (triangle) at the top that points to that character. When the cursor points to a character in left-to-right text, it is placed to the left of that character with its flag pointing to the right towards that character. When the cursor point to a character in right-to-left text, it is placed to the right of that character with its flag pointing to the left towards that character.
- Horizontal bar under the character: In DOS applications, the cursor was a horizontal line under the character that the cursor points to.
- Half rectangle covering half the character: The cursor covers the bottom half of the character that it points to. This is a traditional cursor shape to indicate typing will overwrite the character rather than push it ahead.
- Rectangle covering the whole character: The cursor makes the character invisible. This can also be used to indicate overwrite mode.

Blinking Style

- Do not blink: The cursor is permanently visible in a single color. Choose this option if the blinking distracts you or if it confuses accessibility software such as screen readers or magnification tools.
- Blink on and off: The usual blinking style for text cursors on the Windows platform. The cursor is permanently visible while you type (quickly). When you stop typing for about half a second, the cursor blinks by becoming temporarily invisible. Blinking makes it easier to locate the cursor with your eyes in a large block of text.
- Alternate between two colors: Makes the cursor blink when you stop typing like “on and off”. But instead of making the cursor invisible, it is displayed with an alternate color. This option gives the cursor maximum visibility: the blinking animation attracts the eye while keeping the cursor permanently visible.

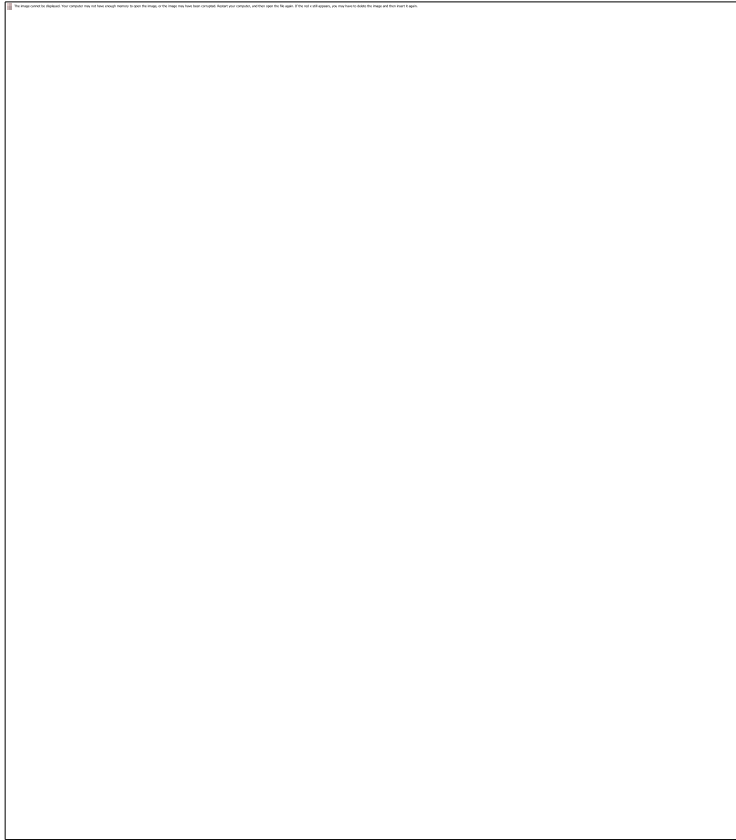
Sizes

- Width: Width in pixels for the vertical bar shape.
- Height: Height in pixels for the horizontal bar shape.
- Flag: Length in pixels of the edges of the flag that indicates text direction.

Colors

- Regular: Used for all shapes and blinking styles except the standard Windows cursor.
- Alternate: Alternate color used by the “alternate between two colors” blinking style.
- Dragging: Color of a second “ghost” cursor that appears while dragging and dropping text with the mouse. It indicates the position the text is moved or copied to when you release the mouse button.

36. Text Encoding Preferences



Computers deal with numbers, not with characters. When you save a text file, each character is mapped to a number, and the numbers are stored on disk. When you open a text file, the numbers are read and mapped back to characters. When saving a file in one application, and opening the that file in another application, both applications need to use the same character mappings.

Traditional character mappings or code pages use only 8 bits per character. This means that only 256 characters can be represented in any text file. As a result, different character mappings are used for different language and scripts. Since different computer manufacturers had different ideas about how to create character mappings, there's a wide variety of legacy character mappings. PowerGREP supports a wide range of these.

In addition to conversion problems, the main problem with using traditional character mappings is that it is impossible to create text files written in multiple languages using multiple scripts. You can't mix Chinese, Russian and French in a text file, unless you use Unicode. Unicode is a standard that aims to encompass all traditional character mappings, and all scripts used by current and historical human languages.

What This Means to You

If you only search through files created on your own Windows computer, or on other Windows computers using the same regional settings, PowerGREP's default settings will suit you perfectly. All your files will all use the same traditional Windows code page and/or Unicode.

In the section “default settings for files not configured below”, select the Windows code page that matches the language you work with. All Windows code pages are an extension of US ASCII, which supports the English alphabet. Code page 1252 is the default for the Americas and Western Europe.

Text files normally should not contain NULL characters. Binary files usually do contain NULL bytes. Turn on "treat files containing NULL characters as binary files" to be able to exclude binary files in the File Selector with the option Search through Binary Files. When you do search through binary files, PowerGREP will display the results in hexadecimal. The file editor edits binary files in hexadecimal mode.

If PowerGREP treats certain text files as binary files, that is because those text files contain spurious NULL characters. You can turn off “treat files containing NULL characters as binary files” to force PowerGREP to treat all files as text files.

On the Windows platform, Unicode files should start with a byte order marker. The byte order marker is a special code that indicates the Unicode encoding (UTF-8, UTF-16 or UTF-32) used by the file. PowerGREP will always detect the byte order marker, and treat the file with the corresponding Unicode encoding. By default, PowerGREP will also write the byte order marker when creating new files. When changing existing files, PowerGREP will write the byte order marker only if it was previously present in the file.

Some applications save Unicode files without byte order markers. Reading a UTF-16 file as if it was encoded with a Windows code page will cause every other character in the file to appear as a NULL character. PowerGREP can detect this situation and read the file as UTF-16. Reading a UTF-8 file as if it was encoded with a Windows code page will cause non-ASCII characters to appear as two or three garbage characters. E.g. the French character é will appear as Å©. PowerGREP can detect if a file contains non-ASCII characters and if all of them are valid UTF-8 sequences, indicating the file is highly likely an UTF-8 file. You should turn on the option “detect UTF-8 and UTF-16 files without a byte order marker”, to prevent UTF-16 files from being treated as binary files, and to make sure UTF-8 files are processed properly.

Text Encoding Definitions

Creating additional text encoding definitions is only necessary when you work with files encoded with different traditional character mappings. This will be the case with files created on different Windows computers using different regional or language settings. It will also be the case with text files created on computers using operating systems other than Windows. Linux systems usually use UTF-8 without a byte order marker (meaning PowerGREP cannot auto-detect the UTF-8 format), or one of the ISO-8859 code pages. Old files created with MS-DOS will use one of the DOS code pages. Files created on IBM mainframes are likely to use one of the EBCDIC encodings.

Click the New button to create a new text encoding definition. Click the Delete button to remove the selected definition. Use the Move Up and Move Down buttons to change the order. PowerGREP looks through the definitions from top to bottom when looking up the encoding to use for a particular file. If the file matches more than one text encoding definition, the topmost definition is used.

Assign each definition a meaningful label. This label is only used in the text encoding preferences. It makes it easy to look up the encoding later in the list.

Select the character mapping to use for files of this type (i.e. files matching the definition's file mask) from the Encoding drop-down list. This list has one extra mapping not available for the default encoding. At the top of the list, you'll find the "binary file" encoding. This is not really an encoding, but tells PowerGREP to treat all files of this type as 8-bit binary files. These files can be excluded in the File Selector with the option Search through Binary Files. When you do search through binary files, PowerGREP will display the results in hexadecimal. The file editor edits binary files in hexadecimal mode.

The file mask is a semicolon-delimited list of file masks, just like the include files masks in the File Selector. If a file's name matches one of the masks in the list, the text encoding definition is used for that file.

If some files of this type are text files, and others are binary files, you can turn on "treat files containing NULL characters as binary files" to make PowerGREP auto-detect binary files. For most text encoding definitions though, you will leave this option off.

If some files of this type are UTF-8 or UTF-16 Unicode files without a byte order marker, while others use another encoding, select the other encoding from the Encoding list, and turn on "detect UTF-8 and UTF-16 Unicode files without a byte order marker". If you're not sure, turn on the option.

On the Windows platform, Unicode files should start with a byte order marker. PowerGREP will write this marker at the start of each Unicode file, unless you've turned off that option for certain text encoding definitions. If an application that claims to support Unicode can't read the Unicode files created by PowerGREP, try turning off the option to write the byte order marker for the files you're trying to open with that application.

If you're not sure whether files of this type should use a byte order marker or not, or if some applications require it and others can't handle it, turn on the option to "preserve the presence of absence of the byte order marker in existing files". When this option is on, and PowerGREP modifies a file, PowerGREP will keep the BOM if it was present in the file, but won't write it if it wasn't present, regardless of whether you turned the "write a byte order marker" option on or off. Note that the "write a byte order marker" option will still determine whether PowerGREP writes the byte order marker to new files that it creates.

Encoding of XML Files

XML files start with an XML declaration that indicates the encoding of the XML file. Before searching a file, PowerGREP will check if it starts with an XML declaration. If so, PowerGREP will process the file using the encoding indicated by the XML declaration.

PowerGREP will always do the XML declaration check for files for which you did not define a text encoding on the Text Encoding page in the Preferences. There is no XML option among the default text encoding settings.

You can turn off the XML declaration check for text encoding definitions. This can be useful when you want to search through XML files that use an encoding not recognized by PowerGREP. Otherwise, PowerGREP will skip those files with an error message indicating the unsupported encoding. Instead, you can select a fixed encoding from PowerGREP's list that is close enough to the one actually used by the XML file.

E.g. PowerGREP supports only a handful of EBCDIC encodings. If you have XML files that use another EBCDIC encoding, you can create a text encoding definition for those XML files. Turn off the XML declaration check, and select the EBCDIC 037 encoding. This way you can still properly search through the parts of the XML file written in English, which could very well be the whole file.

Note that PowerGREP will not automatically add or update the XML declaration when writing XML files. It's up to you if you want to add the declaration to your files, and to make sure it is correct.

Encoding of HTML Files

Most web servers and web browsers do not support the Unicode byte order marker. For HTML file types like HTML, PHP and ASP pages, you should turn off the options to write and preserve the byte order marker, and turn on the option to detect Unicode files without a byte order marker.

HTML files can use a meta tag to set a Content-Type header which can specify the encoding used by the web page. E.g. `<meta http-equiv="Content-Type" content="text/html; charset=win1252">` specifies the Windows 1252 code page. Turn on the “detect HTML Content-Type meta tag” option to make PowerGREP search for this tag at the start of the file, and use the specified encoding if the tag can be found. Note that there's no such option for the default settings. PowerGREP will not look for the HTML meta tag by default.

This is indeed different from the XML declaration check, which PowerGREP does do by default. The reason is that the XML declaration always appears at the very start of the file, so checking its presence is trivial. The meta tag can appear deep in the HTML file's header, so PowerGREP has to search for it.

Note that PowerGREP will not automatically add or update the meta tag when writing HTML files. It's up to you to decide if you want to add the tag to your files, and to make sure it is correct.

Detect ASCII files using `\uFFFF`, ``; or ``; as Unicode files

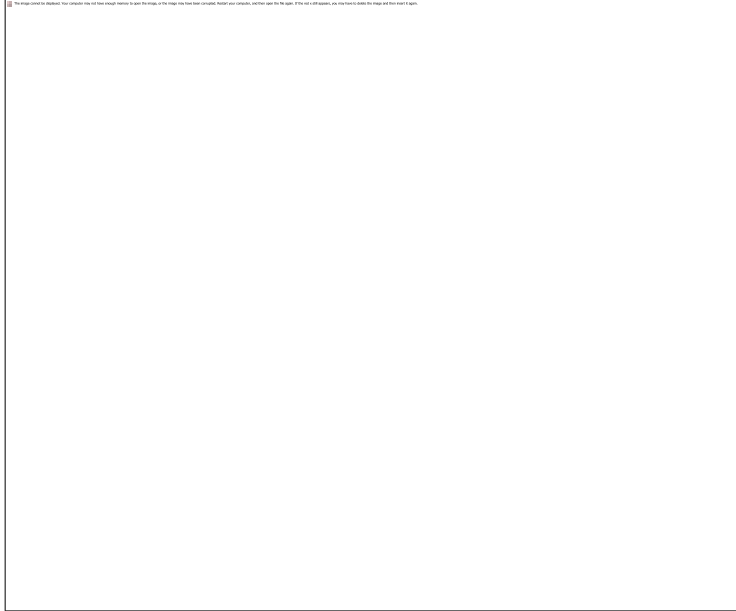
Some files consist only of ASCII characters and specify Unicode characters using numeric Unicode character escapes in the form of `\uFFFF` or using XML entities in the form of ``; and/or ``;

Turn on this option to make PowerGREP treat these files as Unicode and transparently convert the Unicode escapes or XML entities into Unicode characters and back. PowerGREP will display the Unicode characters in the search results, and you'll need to type or paste in those Unicode characters as literal characters if you want to search for them.

Turn this option off to make PowerGREP treat these files as ASCII files. The Unicode escapes and XML entities will appear as such in the search results.

37. Results Preferences

In the Results section in the Preferences screen, you can configure some general aspects of how PowerGREP displays search results.



Results Display Options

When the option Search through Binary Files is off in the File Selector menu, PowerGREP will not search through binary files. To avoid surprises, PowerGREP will show a list on the Results panel of all files it skipped. Turn off the option "indicate skipped binary files" if this list bothers you.

Turn on the option "indicate backup files" if you want to see the name of each backup file listed along with each target file in the results, when that target file caused a file to be overwritten. If you turn off this option, backup copies will still be made, but will not be indicated in the results. This option does *not* affect the Undo History.

By default, PowerGREP indicates files using their full paths in the results. If you turn on the option "show relative paths", PowerGREP will show relative paths instead. Full paths make sure there is no confusion between files with identical names. Relative paths reduce clutter when searching through files in deep folder structures.

The paths will be shown relative to the folder that was marked in the File Selector. If you directly marked a file or folder, no path information will be shown for that file, or the files inside that folder. If you marked a folder for recursion, files inside that folder will be shown with paths relative to that folder.

When search matches are grouped per file in the results, PowerGREP will automatically place the results for each file in a foldable block. You can fold or unfold the file's results by clicking on the + or - button in the left margin. When folded, only the file's path will be visible.

If you turn on the option to fold files by default, the results will initially show only a list of file paths. To see the results of a particular file, you'll need to click the + button to unfold it. If you turn the option off, all results for all files will be visible initially. You can then click the - button to collapse files you're no longer interested in, so there's more space in the results for the remaining files.

Most of the settings for the text viewer on the Results panel are combined into a "text layout". If you have previously configured text layouts in the Action or Editor sections of the preferences, you can select a previously configured text layout from the drop-down list. If not, click the Configure Text Layout button to specify font, text direction, cursor behavior, word selection options, extra spacing, etc.

Results Limits

It's easy to (inadvertently) execute a PowerGREP action that gathers a large amount of search results. Particularly collecting context can eat up a lot of memory. By setting certain limitations you can make sure PowerGREP doesn't eat up all of your computer's memory producing more results than you can even begin to look at.

Maximum Memory Usage to Display Results

All the matches and their context that PowerGREP displays on the Results panel have to fit into your computer's memory. With this setting you can limit the (approximate) amount of memory that PowerGREP will use to display results. This prevents PowerGREP from running out of memory or starving other applications running on your computer for memory.

When an action produces more results than PowerGREP can keep in memory, the action will run to completion. All search matches will be found and processed. The only difference is that the search matches that no longer fit into the allotted memory won't be displayed in the results. PowerGREP then only indicates how many matches were found in each file, just as it does for all files when you use the Quick Execute command in the Action menu.

The memory limit is for each instance of PowerGREP. If you run multiple PowerGREP instances at the same time, reduce the limit so your PC has enough actual RAM for all PowerGREP instances.

Maximum Length of a Line of Context

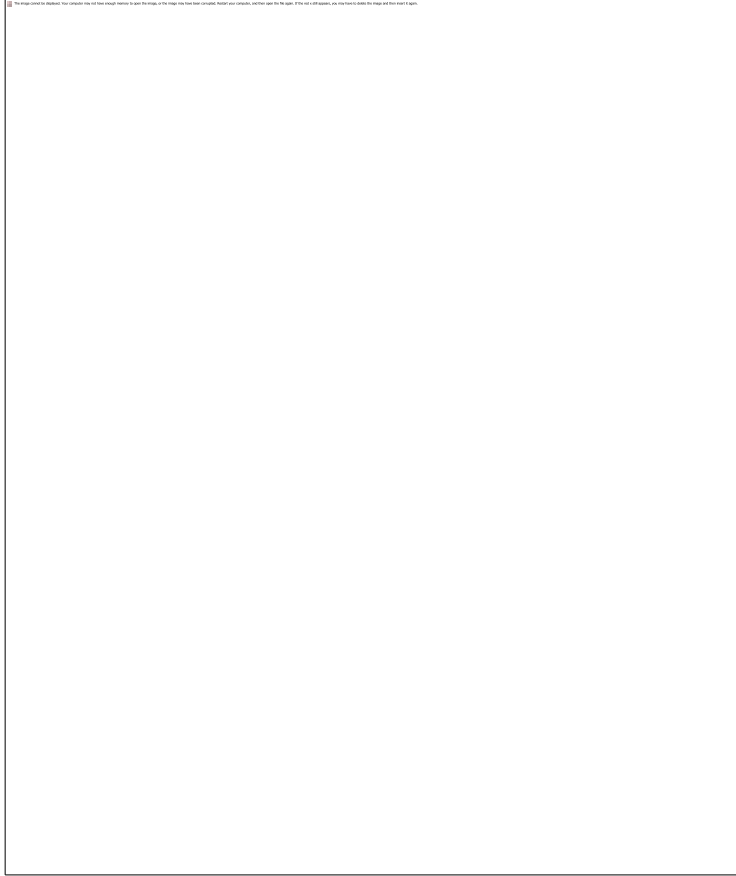
While showing one line of context is usually very convenient, extremely long lines (such as a large XML file with everything on one line) make it hard to see the matches as you'll need to do a lot of horizontal scrolling. The setting for the maximum length of a line of context acts as a safety valve. When showing lines as context, PowerGREP will split up lines that are too long.

This setting does not affect file sectioning. When sectioning files line by line, lines are always searched entirely as one piece, even if they are billions of characters long.

The default setting is 10,000. The minimum setting is 1,000. Anything less than that is easily handled by the Results panel. If you want your lines to fit on screen, use the Word Wrap item in the Results menu instead.

38. Editor Preferences

In the Editor section in the Preferences screen, you can configure the behavior of PowerGREP's built-in file editor.



Next and Previous Buttons Should Also Select The Match in The Results

Turn on to make the Next and Previous File and Match items in the Editor menu move the text cursor on the Results panel to the same file or match, in addition to moving to the next or previous file or match on the Editor panel. Turn off to make the Next and Previous File and Match items in the Editor menu affect the Editor only.

Next and Previous Match Buttons Can Advance to The Next or Previous File

This option determines what the Next and Previous Match items in the Editor menu do when there is no next or previous match to go to in the file that is presently open in the editor. Turn on to make the editor open the next or previous file in the results and highlight the first or last match in that file. Turn off to go to the end or the start of the file that is already open in the editor.

Ctrl+Wheel Scrolls Whole Pages Instead of Zooming

Turn on to make rotating the mouse wheel while holding down the Control button scroll one page with each rotation of the wheel, just like rotating the mouse wheel without any buttons scrolls one line with each rotation. Turn off to make rotating the mouse wheel while holding down the Control button change the font size, effectively zooming the text. Changing the font size with the mouse wheel is temporary. To permanently change the font size, click the Configure Text Layout button in the Preferences.

Visualize Spaces and Tabs

Turn on to visualize spaces as small dots, and tabs as chevrons. Turn off to display spaces and tabs as invisible whitespace.

Visualize Line Breaks

Turn on to show a symbol for each hard line break in the file. This makes it easy to differentiate between permanent line breaks and automatic word wrapping as well as different line break styles. CRLF indicates Windows-style line breaks and LF indicates UNIX-style line breaks.

Tab Size

The width of tabs, as a multiple of the width of a single space. The default is 8, which is the default that most applications use.

Text Layout

Most of the settings for the text editor on the Editor panel are combined into a “text layout”. If you have previously configured text layouts in the Action or Results sections of the preferences, you can select a previously configured text layout from the drop-down list. If not, click the Configure Text Layout button to specify font, text direction, cursor behavior, word selection options, extra spacing, etc.

Text Folding

With the Fold, Unfold, Fold All, and Unfold All commands you can hide parts of the file you’re editing by folding a block of lines under the block’s first line. This can make it easier to get an overview of the different parts of a document. Once you have folded a block, it is indicated with a plus symbol in a square in the left hand margin. If you unfold the block, the indicator changes to a minus symbol and a vertical line indicates the length of the block. This allows you to quickly fold the same block again.

PowerGREP can automatically mark foldable blocks in a file based on the file’s contents. This allows you to quickly fold parts of the file in a logical way, without having to manually select each blocks first. PowerGREP can use file navigation schemes designed for EditPad Pro to mark these foldable blocks. PowerGREP ships with such schemes for a wide variety of file formats. If you want to use a certain file navigation scheme for files with a certain extension, select that scheme from the drop-down list and then make sure the extension is

listed in the semicolon-delimited list of extensions for the scheme. Also make sure the extension isn't listed for any other scheme.

If there's no file navigation scheme for a particular file format, you can still use automatic folding based on the indentation of the lines in the file. PowerGREP does this for any file that doesn't have a file navigation scheme and that matches one of the file masks in the "fold files matching these file masks based on indentation". By default no masks are specified, so no files are folded based on indentation. You could set it to * to fold all files without file navigation schemes based on indentation.

Backups

The backup settings in the Editor section of the Preferences are used whenever you overwrite a file using PowerGREP's built-in editor. These backups are added to the Undo History where you can restore files from their backups or clean up the backups.

The available backup settings are the same as the backup settings on the Action panel. They're described in detail in the Action panel reference. Your backup settings for the editor are also used as the default whenever you clear the Action panel.

39. External Editors Preferences

If you'd rather use one or more external applications rather than PowerGREP's built-in file editor to view or edit files, you can configure them in the External Editors section in the Preferences screen. The editors will appear in the Edit File menu in the File Selector, and in the Edit File menu in the Results panel.



To add an editor, click on the New button. Type in the label that should appear on the editor's menu item in the Label field.

In the command line field, type in the complete command that PowerGREP should execute to launch the editor and open the current file in the editor. You can use all of the path placeholders that are also available in PowerGREP's search-and-replace and "collect data" operations. Most of the time, you will use "%FILE%". %FILE% is replaced with the full path to the file being viewed in the file viewer. The double quotes make sure that filenames with spaces in them are kept together.

Some editors can open a file at a specific position if you supply the correct parameter. You can use two placeholders on the command line to pass the location of a search match on the command line. %START% is replaced by an integer indicating the byte position of the start of the search match, counting all bytes before the match starting from the beginning of the file. The first character in the file has byte position 0 (zero). The second character has position 1 in single byte character set text files, byte position 2 in UTF-16 files, byte position 4 in UTF-32 files, etc. %STOP% is replaced by the byte position at the end of the match, counting all bytes before the match, and all byte in the match. The length of the match equals %STOP%-%START%.

Since many editors support command line parameters for placing the cursor at a specific line or column, but not at a specific byte offset, PowerGREP also provides %LINE% and %COL% placeholders to indicate the start of the match, and %LINESTOP% and %COLSTOP% to indicate the first character after the match. The first character in the file is on line 1 and column 1. The second character is on column 2, regardless of the file's encoding. The line and column placeholders only work when the search results were produced by an

action with the context type set to “use lines as context” and “show line numbers” turned on. When collecting context in a different way, or not at all, PowerGREP does not scan the file for line breaks to calculate line numbers.

The working folder is the folder that is made the current one when the editor is launched. Typically, you will enter %PATH% here, which is the full path to the file being shown in the file viewer, without the file name. You should not put double quotes around the working folder, whether it is likely to contain spaces or not.

You can restrict the editor to be available for certain file types only. This restriction is based on file extensions. For an HTML editor, you could enter *.htm;*.html;*.shtml separating the extensions with semicolons. If you leave the list of extensions blank, then the editor will be available for all files.

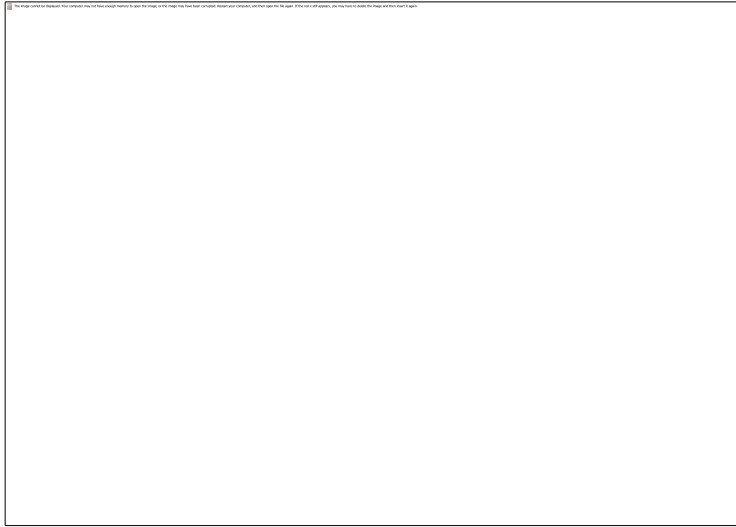
If you turn on the "default editor" checkbox, then PowerGREP will use this editor instead of its built-in editor when you click the Edit button on the toolbar in the Results panel, or when you double-click on a file in the results. You can set more than one editor as being a default editor. If you do so, PowerGREP first checks if there is a default editor with the file's extension listed as an extension the editor recognizes. If there is more than one such editor, the topmost one in the list is used. If no default editor has the extension listed, PowerGREP invokes the topmost editor you marked as default and configured to handle all files (i.e. by not listing any file extensions).

Open All Files Command for This Editor

If all the files in the results have extensions supported by this editor, or if you did not list any extensions for this editor, show an Open All Files command that opens all the files in the results in this editor.

PowerGREP will launch one instance of this editor for each file in the results. You should only enable this command for applications that are capable of reusing existing instances. If there are 10 files in the results, PowerGREP will launch 10 instances. The application should be capable of signaling itself so that 1 instance opens all 10 files, instead of 10 instances cluttering your desktop. To make sure your system doesn't crash with thousands of instances of your editor, PowerGREP won't launch more than 10 instances if the application doesn't close them down.

40. General Preferences



Search with PowerGREP

Turn on to add an item labeled “Search with PowerGREP” to the context menu that appears when you right-click on a folder in Windows explorer. This item will launch PowerGREP with the folder you right-clicked on marked in PowerGREP’s File Selector. Its subfolders are not marked.

Turn off to remove this item if it was previously added.

Search Subfolders with PowerGREP

Turn on to add an item labeled “Search subfolders with PowerGREP” to the context menu that appears when you right-click on a folder in Windows explorer. This item will launch PowerGREP with the folder you right-clicked on and its subfolders marked in PowerGREP’s File Selector.

Turn off to remove this item if it was previously added.

Send To Menu Shortcut

Turn on to add PowerGREP to the Send To submenu of the context menu that appears when you right-click on any file or folder in Windows Explorer. Using PowerGREP’s shortcut in the Send To menu starts PowerGREP and marks the selected files and folders in PowerGREP’s file selector. Subfolders of the selected folders are not marked.

Turn off to remove PowerGREP from the Send To menu.

Send To Menu Shortcut That Includes Subfolders

Turn on to add "PowerGREP (with subfolders)" to the Send To submenu of the context menu that appears when you right-click on any file or folder in Windows Explorer. Using the "PowerGREP (with subfolders)" shortcut in the Send To menu starts PowerGREP and marks the selected files and folders in PowerGREP's file selector, including any subfolders of the selected folders.

Turn off to remove "PowerGREP (with subfolders)" from the Send To menu.

Assistant

The PowerGREP Assistant is the panel in PowerGREP that displays hints about the control that has keyboard focus or that you're pointing to with the mouse. The Preferences screen has its own assistant at the right hand side. Any changes you make to the preferences for the assistant are applied immediately to the assistant on the Preferences screen.

Follow Keyboard Focus and Mouse Pointer

Choose this option to make the Assistant panel display the hint of the control under the mouse pointer. When the mouse pointer is on top of the Assistant, it displays the hint of the control that has keyboard focus.

Follow Keyboard Focus Only

Choose this option to make the Assistant panel always display the hint of the control that has keyboard focus. Mouse movement does not change the display of the Assistant.

Assistant Font

Select the font used by the PowerGREP Assistant which displays helpful hints while you work with PowerGREP.

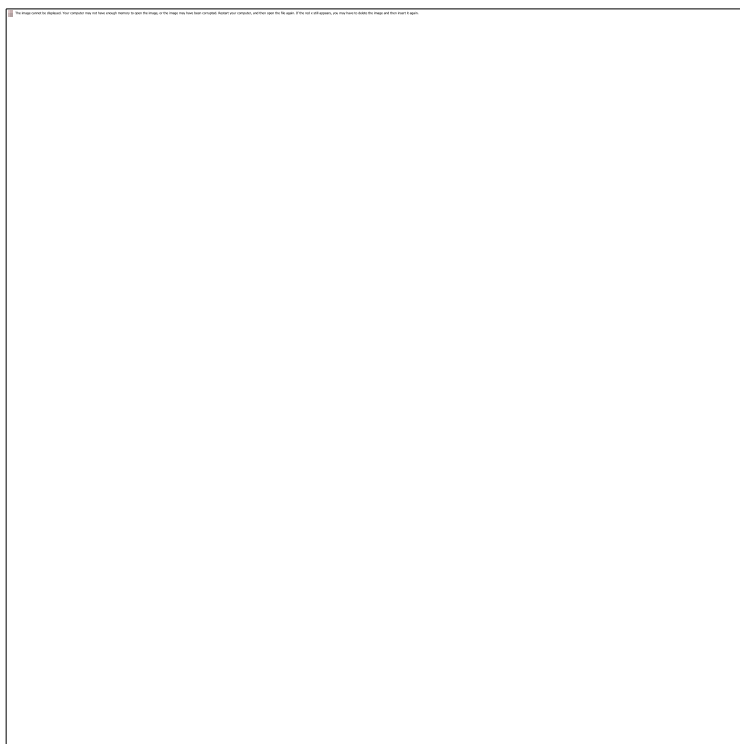
Forum

The settings for the text editor on the Forum panel are combined into a "text layout". If you have previously configured text layouts in the Action, Results, or Editor sections of the preferences, you can select a previously configured text layout from the drop-down list. If not, click the Configure Text Layout button to specify font, text direction, cursor behavior, word selection options, extra spacing, etc.

41. Color Configuration

In the Preferences screen, you can configure the colors used by all edit boxes in PowerGREP to your own taste and eyesight.

Regex Colors

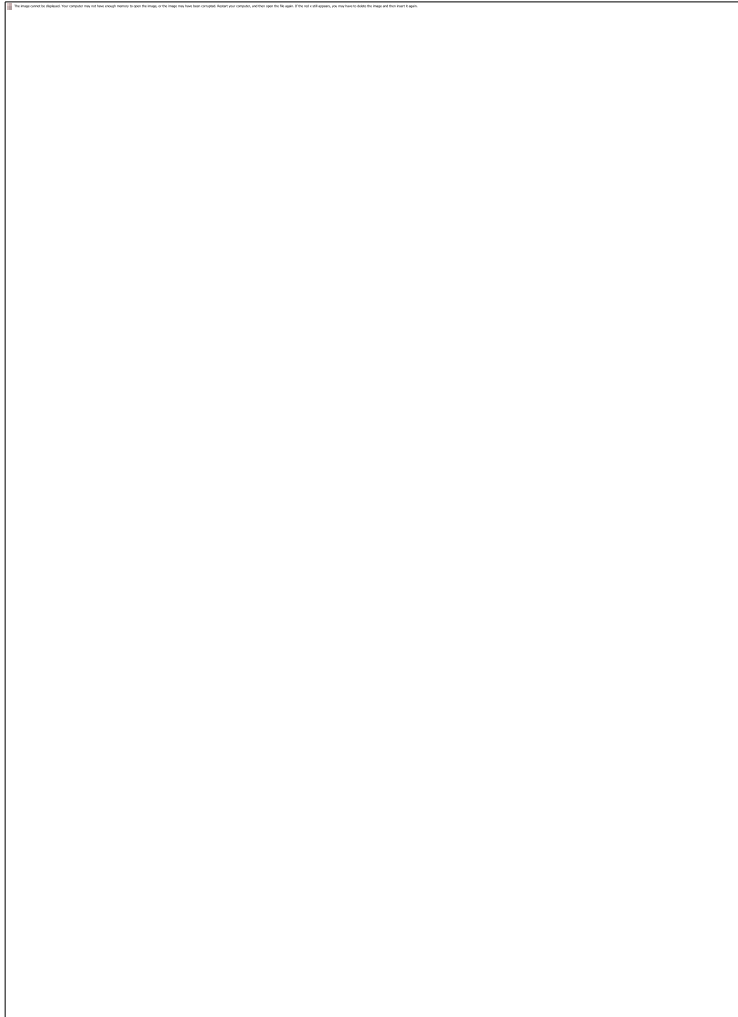


In the Regex Colors section, you can configure the colors for search term edit boxes. These are all edit boxes on the Action panel, the file masks boxes on the File Selector, and the description and details boxes on the Library panel.

You can select one of the preset configurations at the top of the screen. To customize the colors, click on an item in the list of individual colors. Then click the Background Color and Text Color buttons to change the item's colors. You can also change the font style with the bold, italic and underline checkboxes. At the bottom, a sample edit box will show what the color configuration looks like. You can edit the text in the example box to further test the colors.

The “literal text” and “selected text” colors are used by all search term edit boxes. All the other colors are used for syntax highlighting regular expressions. You can quickly disable syntax highlighting by selecting the “no regex syntax coloring” preset configuration.

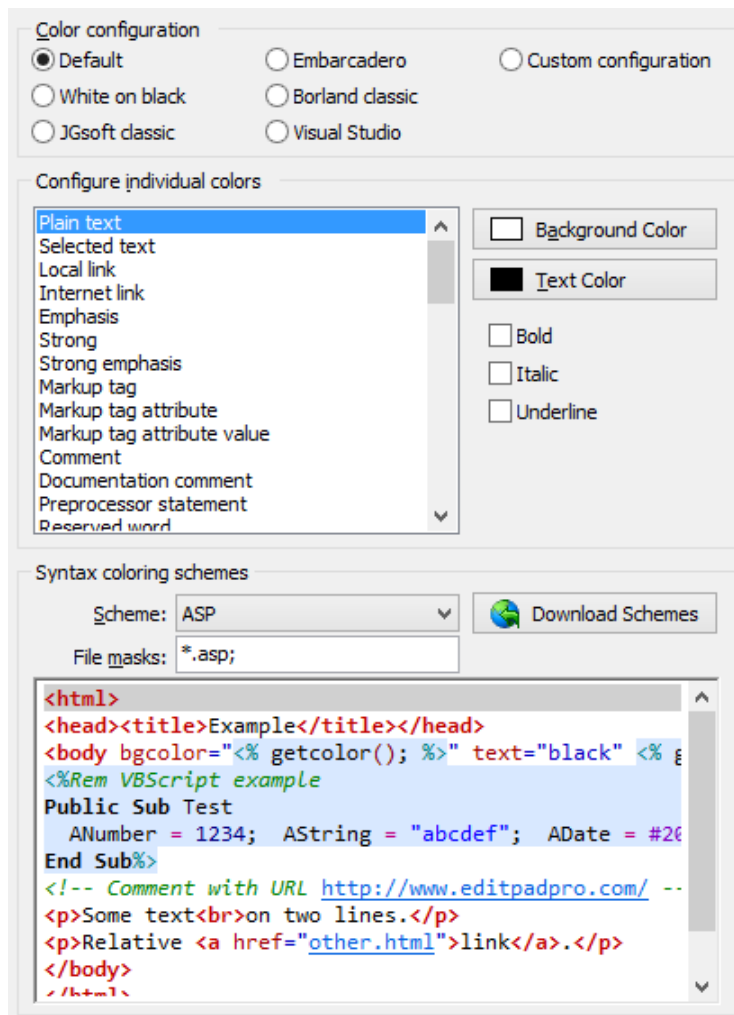
Results Colors



In the Results Colors section, you can configure the colors used to display the results. These colors are also used to highlight matches in the built-in file editor

You can select one of the preset configurations at the top of the screen. To customize the colors, click on an item in the list of individual colors. Then click the Background Color and Text Color buttons to change the item's colors. You can also change the font style with the bold, italic and underline checkboxes. At the bottom, a sample edit box will show what the color configuration looks like.

Syntax Colors



In the Syntax Colors sections, you can configure the colors used by the built-in file editor for syntax coloring. Colors for match highlighting are set in the Results Colors section.

You can select one of the preset configurations at the top of the screen. To customize the colors, click on an item in the list of individual colors. Then click the Background Color and Text Color buttons to change the item's colors. You can also change the font style with the bold, italic and underline checkboxes.

The individual colors have logical names that are used by the various syntax coloring schemes that PowerGREP supports. This way, the same parts of a file are colored the same across different file types. To see a sample, select a syntax coloring scheme from the Scheme drop-down list. You can edit the text in the example box to further test the colors.

The extension of a file determines which syntax coloring scheme the file editor will use. Specify a semicolon-delimited list of file extensions listing all the file types that you want to use a particular coloring scheme for.

To download additional syntax coloring schemes shared by other PowerGREP and EditPad Pro users, click the Download Schemes button. PowerGREP will show a list of available syntax coloring schemes. Simply select a scheme and click the Install button to download it.

If for some reason PowerGREP cannot connect to the internet directly, you can download coloring schemes using your web browser at <http://www.editpadpro.com/cscs.html>.

To create your own syntax coloring schemes, or edit the schemes you downloaded, you will need the JGsoft Custom Syntax Coloring Scheme Editor. You can find the download link in the email message you received when purchasing PowerGREP. If you lost it, you can have the email resent by entering your email address at <http://www.powergrep.com/download.html>.

42. Match Placeholders

Match placeholders can be used to insert search matches or match counts in the search text or replacement text. For this to work, the option “Expand match placeholders and path placeholders” must have been enabled in the action & results preferences. You can easily insert match placeholders by right-clicking on an edit box on the Action panel and selecting Insert Match Placeholder in the placeholders menu.

Placeholder	Meaning and Examples	Availability
%ACTIONDATE%	Date on which PowerGREP started executing the action.	Everywhere
%ACTIONTIME%	Time of the day (hours, minutes, and seconds) on which PowerGREP started executing the action.	Everywhere
%ACTIONDAY%	Day of the month on which PowerGREP started executing the action.	Everywhere
%ACTIONMONTH%	Number of the month on which PowerGREP started executing the action.	Everywhere
%ACTIONYEAR%	Year during which PowerGREP started executing the action.	Everywhere
%ACTIONHOUR12%	Hour based on a 12-hour clock on which PowerGREP started executing the action.	Everywhere
%ACTIONHOUR24%	Hour based on a 24-hour clock on which PowerGREP started executing the action.	Everywhere
%ACTIONAMP%	Replaced with AM or PM depending on whether PowerGREP started executing the action before noon or after noon.	Everywhere
%FILEN% and %FILENZ%	The number of the file being searched through. Counts all files that are searched through, including those without matches. Files are numbered in no particular order if the Action Preferences are set to process files in multiple threads. Files are numbered in alphabetical order if the Action Preferences are set to use only a single thread. %FILEN% starts counting at one, and %FILENZ% at zero.	Everywhere
%FILENA%	The sequential letter of the file being searched through. The first file is “a”, the second “b”, etc. Type the placeholder in upper or lowercase to determine the case of the letter.	Everywhere
%FILEDATE%	Date on which the file currently being processed by the action was last modified.	Everywhere
%FILETIME%	Time of the day (hours, minutes, and seconds) on which the file currently being processed by the action was last modified.	Everywhere
%FILETIME%	Time of the day (hours, minutes, and seconds) on which the file currently being processed by the	Everywhere

	action was last modified.	
%FILEDAY%	Day of the month on which the file currently being processed by the action was last modified.	Everywhere
%FILEMONTH%	Number of the month on which the file currently being processed by the action was last modified.	Everywhere
%FILEYEAR%	Year during which the file currently being processed by the action was last modified.	Everywhere
%FILEHOUR12%	Hour based on a 12-hour clock on which the file currently being processed by the action was last modified.	Everywhere
%FILEHOUR24%	Hour based on a 24-hour clock on which the file currently being processed by the action was last modified.	Everywhere
%FILEAMP%	Replaced with AM or PM depending on whether the file currently being processed by the action was last modified before noon or after noon.	Everywhere
%FILESIZE%	Size in bytes of the file currently being processed by the action (prior to any changes made by the action).	Everywhere
%FILESIZEKB%	Size in kilobytes of the file currently being processed by the action (prior to any changes made by the action).	Everywhere
%FILESIZEMB%	Size in megabytes of the file currently being processed by the action (prior to any changes made by the action).	Everywhere
%FILESIZEGB%	Size in gigabytes of the file currently being processed by the action (prior to any changes made by the action).	Everywhere
%SECTION%	The text of the section being searched through.	In the replacement text of the main action and extra processing, when using file sectioning.
%SECTIONLEFT% and %SECTIONRIGHT%	The part of the text the section being searched through to the left or right of the search match the main action found in that section.	In the replacement text of the main action and extra processing, when using file sectioning.
%SECTIONN% and %SECTIONNZ%	The number of the section being searched through. %SECTIONN% starts counting at one, and %SECTIONNZ% at zero. Example: Collect page numbers	In the search terms and replacement text of the main action and extra processing, when using file sectioning.
%SECTIONNA%	The sequential letter of the section being searched through. The first section is “a”, the second “b”, etc. Type the placeholder in upper or lowercase to determine the case of the letter.	In the search terms and replacement text of the main action and extra processing, when using file sectioning.

%LINE%	The line being searched through	In the replacement text of the main action and extra processing, when searching line by line.
%LINELEFT% and %LINERIGHT%	The part of line being searched through to the left or right of the search match the main action found in that line	In the replacement text of the main action and extra processing, when searching line by line.
%LINEN% and %LINENZ%	The number of the line being searched through. %LINEN% starts counting at one, and %LINENZ% at zero.	In the search terms and replacement text of the main action and extra processing, when searching line by line.
%LINENA%	The sequential letter of the line being searched through. The first line is “a”, the second “b”, etc. Type the placeholder in upper or lowercase to determine the case of the letter.	In the search terms and replacement text of the main action and extra processing, when searching line by line.
%MATCH%	The search match Examples: Padding replacements and Capitalize the first letter of each word	In the replacement text of the main action and extra processing.
%GROUP1%, %GROUP2%, etc.	%GROUP1% is a backreference to a capturing group, equivalent to «\1» in a regular expression or “\1” or “\$1” in the replacement text. The advantage of %GROUP1% is that you can use the padding and case conversion specifiers listed below. Example: Padding replacements	In any regular expression and any replacement text corresponding with a regular expression.
%MATCHSTART% and %MATCHSTARTZ%	The byte offset of the first character in the search match in the file, with the first byte in the file numbered one or zero, respectively.	In the replacement text of the main action and extra processing.
%MATCHSTOP% and %MATCHSTOP%	The byte offset of the first character after the search match in the file, with the first byte numbered one or zero, respectively.	In the replacement text of the main action and extra processing.
%GROUP1START%, %GROUP1STARTZ%, %GROUP2START%, %GROUP2STARTZ%, etc.	%GROUP1START% is the byte offset of the first character in the first capturing group in the regular expression, with the first byte in the file numbered one. %GROUP1STARTZ% starts numbering at zero. %GROUP2START% is the byte offset of the first character in the second capturing group, etc.	In any replacement text corresponding with a regular expression that has capturing groups.
%GROUP1STOP%, %GROUP1STARTZ%, %GROUP2STOP%, %GROUP2STARTZ%, etc.	%GROUP1STOP% is the byte offset of the first character after the first capturing group in the regular expression, with the first byte in the file numbered one. %GROUP1STARTZ% starts numbering at zero. %GROUP2STOP% is the	In any replacement text corresponding with a regular expression that has capturing groups.

	byte offset of the first character after the second capturing group, etc.	
%MATCHN% and %MATCHNZ%	The number of the search match. %MATCHN% starts counting at one, and %MATCHNZ% at zero. When used in the search term, the number indicates the number of the next match to be found, or one more than the number of matches already found when using %MATCHN%. When searching through more than one file, the numbering continues through the whole action. When searching for more than one search term, the matches for all search terms are numbered together.	In the search terms and replacement text of the main action and extra processing.
%MATCHNA%	The sequential letter of the search match. The first match is “a”, the second “b”, etc. Type the placeholder in upper or lowercase to determine the case of the letter. When used in the search term, the letter is the letter of the next match to be found. When searching through more than one file, the sequence continues through the whole action. When searching for more than one search term, the letters for all search terms form one sequence.	In the search terms and replacement text of the main action and extra processing.
%MATCHFILEN% and %MATCHFILENZ%	The number of the search match. %MATCHFILEN% starts counting at one, and %MATCHFILENZ% at zero. When used in the search term, the number indicates the number of the next match to be found, or one more than the number of matches already found. The numbering restarts at one (or zero) for each file searched through. When searching for more than one search term, the matches for all search terms are numbered together. Examples: Add line numbers and Collect a numbered list	In the search terms and replacement text of the main action and extra processing.
%MATCHFILENA%	The sequential letter of the search match. The first match is “a”, the second “b”, etc. Type the placeholder in upper or lowercase to determine the case of the letter. When used in the search term, the letter is the letter of the next match to be found. The sequence restarts with “a” for each file searched through. When searching for more than one search term, the letters for all search terms form one sequence.	In the search terms and replacement text of the main action and extra processing.
%MATCHSECTIONN% and %MATCHSECTIONNZ%	The number of the search match. %MATCHSECTIONN% starts counting at one, and %MATCHSECTIONNZ% at zero. When used in the search term, the number	In the search terms and replacement text of the main action and extra processing, but only when

	<p>indicates the number of the next match to be found, or one more than the number of matches already found. The numbering restarts at one (or zero) for each line or each section searched through. When searching for more than one search term, the matches for all search terms are numbered together.</p> <p>Example: Add line numbers</p>	using file sectioning.
%MATCHSECTIONNA%	The sequential letter of the search match. The first match is “a”, the second “b”, etc. Type the placeholder in upper or lowercase to determine the case of the letter. When used in the search term, the letter is the letter of the next match to be found. The sequence restarts with “a” for each line or each section searched through. When searching for more than one search term, the letters for all search terms form one sequence.	In the search terms and replacement text of the main action and extra processing, but only when using file sectioning.
%MATCHTERMN% and %MATCHTERMNZ%	The number of the search match. %MATCHTERMN% starts counting at one, and %MATCHTERMNZ% at zero. When used in the search term, the number indicates the number of the next match to be found, or one more than the number of matches already found when using %MATCHTERMN%. When searching through more than one file, the numbering continues through the whole action. When searching for more than one search term, each search term has its own numbering sequence independent of the other search terms.	In the search terms and replacement text of the main action and extra processing.
%MATCHTERMNA%	The sequential letter of the search match. The first match is “a”, the second “b”, etc. Type the placeholder in upper or lowercase to determine the case of the letter. When used in the search term, the letter is the letter of the next match to be found. When searching through more than one file, the sequence continues through the whole action. When searching for more than one search term, each search term has its own lettering sequence independent of the other search terms.	In the search terms and replacement text of the main action and extra processing.
%MATCHFILETERMN% and %MATCHFILETERM NZ%	The number of the search match. %MATCHFILETERMN% starts counting at one, and %MATCHFILETERMNZ% at zero. When used in the search term, the number indicates the number of the next match to be found, or one more than the number of matches already found. The numbering restarts at one (or zero) for each file searched through. When searching for more than one search term, each search term has its	In the search terms and replacement text of the main action and extra processing.

	own numbering sequence independent of the other search terms.	
%MATCHFILETERMNA%	The sequential letter of the search match. The first match is “a”, the second “b”, etc. Type the placeholder in upper or lowercase to determine the case of the letter. When used in the search term, the letter is the letter of the next match to be found. The sequence restarts with “a” for each file searched through. When searching for more than one search term, each search term has its own lettering sequence independent of the other search terms.	In the search terms and replacement text of the main action and extra processing.
%MATCHSECTIONTERM N% and %MATCHSECTIONTERM N Z%	The number of the search match. %MATCHSECTIONTERM N% starts counting at one, and %MATCHSECTIONTERM N Z% at zero. When used in the search term, the number indicates the number of the next match to be found, or one more than the number of matches already found. The numbering restarts at one (or zero) for each line or each section searched through. When searching for more than one search term, each search term has its own numbering sequence independent of the other search terms.	In the search terms and replacement text of the main action and extra processing, but only when using file sectioning.
%MATCHSECTIONTERM NA%	The sequential letter of the search match. The first match is “a”, the second “b”, etc. Type the placeholder in upper or lowercase to determine the case of the letter. When used in the search term, the letter is the letter of the next match to be found. The sequence restarts with “a” for each line or each section searched through. When searching for more than one search term, each search term has its own lettering sequence independent of the other search terms.	In the search terms and replacement text of the main action and extra processing, but only when using file sectioning.
%MATCHCOUNT%	The number of times a particular search match was found. This placeholder is substituted at the end of “collect data” actions that group unique matches. The placeholder is replaced with the number of items that the same piece of text was collected.	Only available in the text to be collected of “collect data” actions, and only when grouping identical matches.
%GUID%	Generates a new GUID in the form of {067F8296-9D1F-4CF2-87E0-70EFC4CE41BF} each time a replacement is made.	In the replacement text of the main action and extra processing.

Padding

You can add additional specifiers to all of the above placeholders. You can pad the placeholder's value to a certain length, and control the casing of any letters in its value. The specifiers must appear before the second % sign in the placeholder, separated from the placeholder's name with a colon. E.g. `%MATCH:6L%` inserts the match padded with spaces at the left to a length of 6 characters. You can add both padding and case placeholders. `%MATCH:U:6L%` inserts the padded match converted to upper case.

Padding specifiers start with a number indicating the length, followed by a letter indicating the padding style. The length is the number of characters the placeholder should insert into the regular expression or replacement text. If the length of the placeholder's value exceeds the requested length, it will be inserted unchanged. It won't be truncated to fit the length. If the value is shorter, it will be padded according to the padding style you specified.

The L or "left" padding style puts spaces before the placeholder's value. This style is useful for padding numbers or currency values to line them up in columns. The R or "right" padding style puts spaces after the placeholder's value. This style is useful for padding words or text to line them up in columns. The C or "center" padding style puts the same number of spaces before and after the placeholder's value. If an odd number of spaces is needed for padding, one more space will be placed before the value than after it.

The Z or "zero" padding style puts zeros before the placeholder's value. This style is useful for padding sequence numbers. The A or "alpha" padding style puts letters "a" before the placeholder's value. This style is useful for padding sequence letters like `%MATCHNA%`.

Padding style letters are case insensitive, except for the "alpha" style. `%matchna:6a%` uses lowercase letters, and `%MATCHNA:6A%` uses uppercase letters.

Examples: Padding replacements and Padding and unpadding CSV files

Case Conversion

Case conversion specifiers consist of a letter only. The specifier letters are case insensitive. Both "U" and "u" convert the placeholder's value to uppercase. L converts it to lowercase, I to initial caps (first letter in each word capitalized) and F to first cap only (first character in the value capitalized). E.g. `%MATCH:I%` inserts the match formatted as a title.

Instead of a case conversion specifier, you can use the case adaptation specifier. It consists of A followed by a digit (not to be confused with a number followed by A, which is a padding specifier). This specifier is only available in a replacement text corresponding with a regular expression. You can use the specifier to give the placeholder the same casing style as the regular expression match or the text matched by a capturing group. Specify zero for the whole regex, or the backreference number of a capturing group. E.g. `%MATCH:A2%` inserts the whole regular expression match, converted to the case used by the second capturing group. The case adaptation specifier detects and adapts to uppercase, lowercase, initial caps and first cap. If the referenced capturing group uses a mixed casing style, the placeholder's value is inserted unchanged.

Example: Capitalize the first letter of each word

Arithmetic

Arithmetic specifiers perform basic arithmetic on the placeholder's value. This works with any placeholder that, at least prior to padding, represents an integer number. If the placeholder does not represent a number, the arithmetic specifiers are ignored. For placeholders like `%MATCH%` that can sometimes be numeric and other times not, the arithmetic specifiers are used whenever the placeholder happens to represent an integer.

An arithmetic specifier consists of one or more operator and integer pairs. The operator can be `+`, `-`, `*`, or `/` to signify addition, subtraction, multiplication, or integer division. It must be followed by a positive integer. Multiple pairs of operators and integers are evaluated from left to right. E.g. when `%MATCHN%` evaluates to 2, `%MATCHN:+1*2%` evaluates to 6 while `%MATCHN:/3%` evaluates to 0. Integer division drops the fractional part of the division's result.

43. Path Placeholders

Path placeholders can be used in the replacement text on the Replace and Sequence pages, as well as in the text to be collected on the Collect page. For this to work, the option “Expand match placeholders and path placeholders” must have been enabled in the action & results preferences.

The placeholders allow you to use the full path or parts of the path to the file that PowerGREP is searching through in replacements and collections. When processing a file, PowerGREP will set %FILE% to the full path to the file, and compute the other placeholders from that.

If the file is inside an archive, PowerGREP will treat the path to the archive as the folder containing the file. E.g. when searching through a file zipped.txt in an archive c:\data\archive.zip, then %FILE% is set to c:\data\archive.zip\zipped.txt. If the archive contains a folder structure, and PowerGREP is searching through zipfolder\zipped.txt in the same archive, then %FILE% is set to c:\data\archive.zip\zipfolder\zipped.txt.

Placeholder	Meaning	Example
%FILE%	The entire path plus filename to the file	C:\data\files\web\log\foo.bar.txt
%FILENAME%	The file name without path	foo.bar.txt
%FILENAMENOEXT%	The file name without the extension	foo.bar
%FILENAMENODOT%	The file name cut off at the first dot	foo
%FILENAMENOGZ%	The file name without the .gz, .bz2, or .xz extension	foo.bar
%FILEEXT%	The extension of the file name without the dot	txt
%FILELONGEXT%	Everything in the file name after the first dot	bar.txt
%PATH%	The full path without trailing delimiter to the file	C:\data\files\web\log
%DRIVE%	The drive the file is on.	C: for DOS paths \\server for UNC paths blank for UNIX paths
%FOLDER%	The full path without the drive and without leading or trailing delimiters	data\files\web\log
%FOLDER1%	First folder in the path	data
%FOLDER2%	Second folder in the path	files
(...etc...)		
%FOLDER99%	99th folder in the path.	In this example, this tag will be replaced with nothingness, because there are less than 99 folders.

%FOLDER<1%	Last folder in the path	log
%FOLDER<2%	Second folder from the end in the path	web
(...etc...)		
%FOLDER<99%	99th folder from the end in the path.	In this example, this tag will be replaced with nothingness.
%PATH1%	First folder in the path	data
%PATH2%	First two folders in the path	data\files
(...etc...)		
%PATH99%	First 99 folders in the path	data\files\web\log
%PATH<1%	Last folder in the path	log
%PATH<2%	Last two folders in the path	web\log
(...etc...)		
%PATH<99%	Last 99 folders in the path	data\files\web\log
%PATH-1%	Path without the drive or the first folder	files\web\log
%PATH-2%	Path without the drive or the first two folders	web\log
(...etc...)		
%PATH-99%	Path without the drive or the first 99 folders.	In this example, this tag will be replaced with nothingness.
%PATH<-1%	Path without the drive or the last folder	data\files\web
%PATH<-2%	Path without the drive or the last two folders	data\files
(...etc...)		
%PATH<-99%	Path without the drive or the last 99 folders.	In this example, this tag will be replaced with nothingness.

Examples: Compile indices of files and Generate a PHP navigation bar

Combining Path Placeholders

You can string several path placeholders together to form a complete path. If you have a file `c:\data\test\file.txt` then `d:\%FOLDER2%\%FILENAME%` will be substituted with `d:\test\file.txt`. However, if the original file is `c:\more\file.txt` then the same path will be replaced with `d:\file.txt` because `%FOLDER2%` is empty. The result is an invalid path.

The solution is to use combined path placeholders, like this: `d:\%FOLDER2\FILENAME%`. The first example will be substituted with `c:\test\file.txt` just the same, and the second will be substituted with `d:\file.txt`, a valid path. You can combine any number of path placeholders into a single path placeholder,

separating them either with backslashes (\) or forward slashes (/). Place the entire combined placeholder between two percentage signs.

A slash between two placeholders inside the combined placeholder is only added if there is actually something to separate inside the placeholder. Slashes between two placeholders will never cause a slash to be put at the start or the end of the entire resulting path. In the above example, the backslash inside the placeholder is only included in the final path if %FOLDER2% is not empty.

A slash just after the first percentage sign makes sure that the resulting path starts with a slash. If the entire resulting path is empty, or if it already starts with a slash, then the slash is not added.

A slash just before the final percentage sign makes sure that the resulting path ends with a slash. If the entire resulting path is empty, or if it already starts with a slash, then the slash is not added.

Mixing backslashes and forward slashes is not permitted. Using a forward slash inside a combined placeholder, will convert all backslashes in the resulting path to forward slashes. This is useful when creating URLs based on file names, as URLs use forward slashes, but Windows file names use backslashes.

Example: If the original path is `c:\data\files\web\log\foo.bar.txt`

- %\FOLDER1\% => \data\
• %\FOLDER5\% => (nothing)
- %PATH-2\FILENAME% => web\log\foo.bar.txt
- %PATH-2/FILENAME% => web/log/foo.bar.txt
- %PATH-4\FILENAME% => foo.bar.txt
- %DRIVE\PATH-2\FILENAME% => c:\web\log\foo.bar.txt
- %DRIVE\PATH-4\FILENAME% => c:\foo.bar.txt
- %\FOLDER1\FOLDER4\% => \data\log\
• %\FOLDER1\FOLDER5\% => \data\

44. Command Line Parameters

PowerGREP can be fully controlled from the command line. This allows you to use PowerGREP from batch files or scripts and add PowerGREP as an external tool to other applications.

All parameters are optional. You can use as many or as few of them as you want. The order of the parameters on the command line is irrelevant, except that when you specify both files and options, the files should be specified before options. This to ensure that PowerGREP loads the file first, and then applies the options you specified. If you specify the options first, they'll be replaced by whatever was saved in the file.

If a parameter requires a value as a second parameter, the second parameter must follow right after the first one. Values must always be specified as a separate parameter (i.e. be separated from the parameter by a space). Values are indicated between sharp brackets in the list below. Remember that if a value contains spaces, you must put double quotes around it (eg: "search text") to make sure the value is interpreted as a single parameter. For some parameters, the number of values is variable. Make sure to specify the correct number of values. If you want to leave a required value blank, specify two double quotes. E.g. /replacertext "" blanks the replacement text.

Opening Files via The Command Line

You can specify any number of files of the command line, but only one file of each kind. You can specify one file selection file, one action file, one results file, one library file and one undo history file. The file will be loaded into the corresponding panel. In addition, you can specify one file of any other file. That file will be opened in the built-in file editor.

File selections are saved in file selection files, action files and results files. If you specify two or all three of these files on the command line, PowerGREP will use the file selection from the file selection file, or from the action file if you didn't specify a file selection file. Only when you don't specify either a file selection file or an action file, will PowerGREP read the file selection from the results file.

In similar vein, action definitions are saved in both action files and results files. If you specify both an action file and results file on the command line, PowerGREP will read the action definition from the action file.

If you specify options that affect the file selection or action, PowerGREP will load the file, and then use the options to modify the settings. In this situation, PowerGREP's caption bar will *not* indicate the name of the file selection file or action file.

File Selection, Action and Results Options

Use the command line parameters below to change basic settings in the file selection and action definition. Not all settings you can make in PowerGREP's user interface can be made via the command line. To control the additional settings, first save a file selection file and/or action file in the user interface. Then pass that file on the command line, *before* any of the options listed below.

1. /simple sets the action type to "simple search".
2. /search sets the action type to "search".

3. `/collectsets` the action type to “collect data”.
4. `/findsets` the action type to “list files”. This action type was known as “find files” in PowerGREP 3, hence the name of the command line parameter.
5. `/findnamesets` the action type to “search file names”.
6. `/renamesets` the action type to “rename files”.
7. `/replacesets` the action type to “search-and-replace”.
8. `/deletesets` the action type to “search-and-delete”.
9. `/mergesets` the action type to “merge files”.
10. `/splitsets` the action type to “split files”.
11. `/searchtext <text>`sets the search term(s) of the main part of the action to “text”. If you have more than one search term, use one `/searchtext` parameter in combination with `/delimitsearch`.
12. `/replacetext <text>`sets the replacement text or text to be collected to “text”.
13. `/searchbytes <bytes>`sets the search term of the main part of the action. The `<bytes>` value must be a string of hexadecimal bytes. Changes the search type to binary data.
14. `/replacebytes <bytes>`sets the replacement bytes or bytes to be collected. The `<bytes>` value must be a string of hexadecimal bytes. Changes the search type to binary data.
15. `/searchtextfile <file path> <charset>`loads the search term(s) for the main part of the action from a file. If the file contains more than one search term, use `/delimitsearch` to specify the delimiter. If the file also contains the replacement text, use `/delimitreplace` to specify the delimiter.

You can specify an additional value after the file name to indicate the character set or text encoding used by the file you’re reading the search terms from. You can use the same identifiers used by XML files and HTML files to specify character sets, such as `utf-8`, `utf-16le`, or `windows-1252`. You can omit this parameter if the file starts with a Unicode signature (BOM). The default is your computer’s default Windows code page.
16. `/searchbytesfile <file path>`loads the search term(s) for the main part of the action from a file. The file should contain the actual bytes you want to search for (unlike the `/searchbytes` parameter which expects the hexadecimal representation of the bytes). Changes the search type to binary data. If the file contains more than one search term, use `/delimitsearch` to specify the delimiter. If the file also contains the replacement bytes, use `/delimitreplace` to specify the delimiter.
17. `/regexsets` the search type to a regular expression.
18. `/literalsets` the search type to literal text.
19. `/delimitprefix <delimiter>`sets the search prefix label delimiter to “delimiter”. Also sets the search type to a delimited list.

- 20.** `/delimitsearch <delimiter>`sets the search item delimiter to “delimiter”. Also sets the search type to a delimited list.
- 21.** `/delimitreplace <delimiter>`sets the search pair delimiter to “delimiter”. Also sets the search type to a delimited list.
- 22.** `/optnonoverlap <0|1>`Sets the option “non-overlapping search”. Only has an effect when the search type is a delimited list.
- 23.** `/optdotal1 <0|1>`Sets the option “dot matches newlines”. Only has an effect when the search type is a regular expression.
- 24.** `/optwords <0|1>`Sets the option “whole words only”. Does not have any effect when the search type is a regular expression.
- 25.** `/optcase <0|1>`Sets the option “case sensitive”.
- 26.** `/optadaptive <0|1>`Sets the option “adaptive case”.
- 27.** `/optinvert <0|1>`Sets the option “invert results”. Only has an effect when the action type is “list files”, or when you load an action definition that sections files.
- 28.** `/context <none|section|line>`Sets the “context type” to “no context”, “use sections as context”, or “use lines as context”. Context is only used to display results on the Results panel in PowerGREP or when saving results using the `/save` parameter.
- 29.** `/contextextra <context|lines> <before> <after>`Tells PowerGREP how many blocks of context or how many lines of context to show before and after each match, in addition to the block of context that contains the match. E.g. `/contextextra lines 2 3` shows 2 lines before and 3 lines after. If you omit the “context” or “lines” parameter after `/contextextra`, then “context” is implied when using sections as context, and “lines” is implied when using lines as context. Both the before and after numbers are required. This parameter is ignored when using `/context none`.
- 30.** `/target <same|"copy modified"|"copy all"|none|single|move|delete|replacement|placeholders> <"single folder"|"folder tree"|archive|"numbered archive"|placeholders> <location>`Sets the target options on the Action panel. This parameter must be followed by one value with of the target types listed below. If the target type is something other than “same”, “none”, or “replacement”, then that value needs to be followed by two more values.

The first value indicates how files are copied. When copying files, the original file will remain untouched. The available values depends on the action type. Values with spaces need to be kept together with double quotes.
 same = Do not copy files but change the file searched through. Do not specify any destination type or location. (search; collect data; search-and-replace; search-and-delete)

“copy modified” = Copy files in which matches have been found. (search; collect data; list files; rename files; search-and-replace; search-and-delete)

“copy all” = Copy all files searched through. (search-and-replace; search-and-delete)

none = Do not save results. Do not specify any destination type or location. (list files; simple search; search; collect data; list files)

single = Save results to single file. (search; collect data; list files; merge files)

move = Move matching files (list files; rename files)

delete = Delete matching files. Do not specify any destination type or location. (list files)
 replacement = Use replacement text as target (merge files; split files)
 placeholders = Use path placeholders. Do not specify any destination type or location. The second value must also be “placeholders” and the third value must be the path using path placeholders. (search; collect data; merge files)

The second value indicates the destination type for copied files:

“single folder” = Place all target files into a single folder.
 “folder tree” = Place target files into a folder tree.
 archive = Place target files into an archive.
 “numbered archive” = Place target files into a numbered archive.
 placeholders = Use path placeholders

The third value indicates the actual location. It must specify the full path to a folder, a file, an archive or use path placeholders, depending on the destination type in the second parameter.

31. /backup <none|"single bak"|"single tilde"|"multi bak"|"multi name"|"same|placeholders|history> <"same folder"|subfolder|"single folder"|"folder tree"|archive|"numbered archive"> <location>Sets the backup options on the Action panel. This parameter must be followed by one value with of the backup types listed below. If the backup type is something other than “none” or “history”, then that value needs to be followed by two more values.

The first value indicates the type of backup to create. Values with spaces need to be kept together with double quotes.

none = Do not create backup files. Do not specify any destination type or location.
 “single bak” = Single backup appending .bak extension
 “single tilde” = Single backup with .* extension
 “multi bak” = Multi backup appending .bak, .bak2, ... extensions
 “multi name” = Multi backup prepending "Backup X of ..."
 same = Backup with same file name as original file (destination cannot be the same folder)
 placeholders = Use path placeholders. The second parameter must be specified but its value is ignored. The third parameter must specify the path using path placeholders.
 history = Hidden __history folder. Do not specify any destination type or location.

The second value indicates the destination type of the backup files:

“same folder” = Same folder as original. Do not specify a location.
 subfolder = Place all backup files into a specific subfolder of the folders holding the original files. Specify the name of a subfolder as the location.
 “single folder” = Place all backup files into a single folder. Specify the full path to a folder as the location.
 “folder tree” = Place backup files into a folder tree. Specify the full path to a folder as the location.
 archive = Place backup files into an archive. Specify the full path to an archive file as the location.
 “numbered archive” = Place backup files into a numbered archive. Specify the full path to an archive file as the location.

The third value indicates the actual location, either a folder, archive file or a path using path placeholders.

32. /optbinary <0|1>Sets the option “search through binary files”.

33. /optarchives <0|1>Sets the option “search through archives”.

- 34. /folder <folders>**The value must be a comma-delimited or semicolon-delimited list of full paths to folders. Includes those folders, but not their subfolders, in the next action.
- 35. /folderrecurse <folders>**The value must be a comma-delimited or semicolon-delimited list of full paths to folders. Includes those folders and their subfolders in the next action.
- 36. /folderexclude <folders>**The value must be a comma-delimited or semicolon-delimited list of full paths to folders. Excludes those folders from the next action. Subfolders are also excluded, unless they're explicitly included.
- 37. /file <files>**The value must be a comma-delimited or semicolon-delimited list of full paths to files. Includes those files in the next action.
- 38. /fileexclude <files>**The value must be a comma-delimited or semicolon-delimited list of full paths to files. Excludes those files from the next action.
- 39. /masks <include> <exclude> <0|1>**Sets the file masks. The first value is the inclusion mask. The second value is the exclusion mask. The third value indicates if the masks are traditional file masks or regular expressions.
- 40. /resultsoptions <none|file|"file target"|"match"|"match number"|"match context"|"match context number"|"align match context"|"align match context number"> <none|file|"file void"|"file match"|"file match void"|"match"|"match file"> <none|header|before|after|"before after"|"group"|"header group"|"before group"|"after group"|"before after group"> <none|reverse|"alpha inc"|"alpha dec"|"total inc"|"total dec"|"new old"|"old new"> <none|"alpha inc"|"alpha dec"|"total inc"|"total dec"> <match|replace|inline|separate>**Sets the options to be used on the Results panel for displaying the search results, or saving them into a text file with the /save parameter. You always need to specify 6 numbers, one for each of the 6 drop-down lists on the Results panel.

Display files and matches:

none = Do not show files or matches

file = File names only

“file target” = Matches without context

match = Matches with section numbers

“match number” = Matches with context numbers

“match context” = Matches with context

“match context number” = Matches with context and context numbers

“align match context” = Aligned matches with context

“align match context number” = Aligned matches with context and context numbers

Group search matches:

none = Do not group matches

file = Per file

“file void” = Per file, with or without matches

“file match” = Per file, then per unique match

“file match void” = Per file, per match, with or without matches

match = Per unique match

“match file” = Per unique match, listing files

Display totals:

none = Do not show totals

header = Show totals with the file header (or before the results if no header).

before = Show totals before the results

after = Show totals after the results

“before after” = Show totals before and after the results.

group = Show totals for grouped matches.

“header group” = Totals with the file header, and grouped matches.

“before group” = Totals before results, and grouped matches.

“after group” = Totals after results, and grouped matches.

“before after group” = Totals before and after the results, and grouped matches.

Sort files:

none = First searched to last searched

reverse = Last searched to first searched

“alpha inc” = Alphabetically, A..Z

“alpha dec” = Alphabetically, Z..A

“total inc” = By increasing totals

“total dec” = By decreasing totals

“new old” = Newest to oldest

“old new” = Oldest to newest

Sort matches:

none = Show in original order

“alpha inc” = Alphabetically, A..Z

“alpha dec” = Alphabetically, Z..A

“total inc” = By increasing totals

“total dec” = By decreasing totals

Display replacements:

match = Search match only

replace = Replacement only

inline = In-line match and replacement

separate = Separate match and replacement

41. /save <filename> <charset>If the extension is .pgr or .pgsr, the results created by the /preview, /execute or /quick parameter will be saved into a PowerGREP results file or a PowerGREP sequence results file. If you specify any other extension, PowerGREP will export the results as a plain text file or HTML that can be read by other software. The /save parameter will be ignored if you did not specify /preview, /execute or /quick. The file will be overwritten without warning if it already exists.

If the extension is not .pgr or .pgsr, you can specify an additional value after the file name to determine the character set or text encoding to be used for the text file. You can use the same identifiers used by XML files and HTML files to specify character sets, such as utf-8, utf-16le, or windows-1252. If you omit this parameter, your computer’s default Windows code page is used.

42. /pre viewPreview the action.

43. /exec uteExecute the action.

44. /qu ic kQuick execute the action.

45. /quitThis parameter causes PowerGREP to terminate after successfully executing an action. This parameter is only used if you specify `/preview` and `/save`, or `/execute`, or `/quick`. This parameter is implied if you specify `/silent`. This parameter is ignored if you specify `/reuse` and a running instance was actually reused.

46. /reuseTells PowerGREP to pass the command line parameters to a PowerGREP instance that is already running if that instance is not already executing an action. If there are multiple PowerGREP instances running that are not executing an action, one of those instances is chosen at random. If there are no idle PowerGREP instances then a new instance is started as if you had not specified `/reuse`. This parameter is ignored if you specify `/silent`.

If there are errors in your command line then the `/reuse` parameter only takes effect if it occurs before the error on the command line. Specify `/reuse` as the first parameter if you want to make sure that errors in the command line are displayed by a running instance rather than by a new instance.

47. /silentQuick execute the action without showing PowerGREP at all. No indication is given that PowerGREP is running.

If there are errors in your command line or in the files it references then a message box is displayed to indicate the error, even when using `/silent`. If the `/silent` parameter occurs on the command before the invalid parameter, then the message box is all that is displayed. If the parameter occurs on the command line after the invalid parameter, then the `/silent` parameter is ignored and a new PowerGREP instance is started. Specify `/silent` as the first parameter to make sure no new PowerGREP instance is started.

48. /noundoTells PowerGREP not to add the action to the undo history. This means you will not be able to undo the action or delete any backup files that it created. This parameter is ignored unless you use `/silent`. `/noundo` is implied for actions that do not create backup files (either because they don't save any files or because you chose not to create backups). You should only use `/noundo` when using PowerGREP as part of an automated process and your automated process already deals with the backup files that PowerGREP creates.

49. /noundomanagerTells PowerGREP to add the action directly to the undo history, without using the undo manager. This parameter is ignored unless you use `/silent`. You have to specify the path to a `.pgu` file to save the undo history when using `/noundomanager`. If the `.pgu` file does not exist it will be created. If it does exist the action is added to the `.pgu` file. The difference between using and not using the undo manager is that the undo manager allows multiple PowerGREP instances to share the same undo history. You should not use `/noundomanager` unless you can be 100% sure no other PowerGREP instance is using the same `.pgu` file in any way.

50. /nocachecan be used in combination with `/silent` to temporarily disable the conversion cache. Files in proprietary formats will be converted even if they were cached before. The conversions will not be added to the cache. The conversion manager will not be used at all. Use this option if the action you're executing silently searches through files that you normally don't search through. That way PowerGREP doesn't needlessly take up disk space to cache the plain text conversions of these files. It also prevents PowerGREP from flushing files that you do regularly work with from the cache when the cache reaches its maximum size.

45. XML Format of PowerGREP Files

When working with PowerGREP, you will save your data in a number of different files. File selections are saved in *.pgf files, action definitions are saved in *.pga files, libraries are saved in *.pgl files, results are saved in *.pgr files and the undo history is saved in a *.pgu file. All these files are XML files. You can easily open them in a text editor or XML editor to look at their contents.

The main benefit of the XML format is that you can use standard XML software to read files saved by PowerGREP, or even create your own. The ability to create file selection files and action definition files is particularly useful. While PowerGREP offers a wide range of command line parameters, not all aspects of the file selection and action definition can be controlled from the command line. The flat command line is simply too cumbersome to control PowerGREP's wide range of abilities. A structured XML file is much more useful. Instead of controlling individual settings via command line parameters, simply use your favorite XML software or XML programming library to generate a .pgf and/or .pga file, and pass those on the command line to PowerGREP.

Reading PowerGREP results files can be very handy if you want to apply some special processing to the results found by PowerGREP. The XML structure of a *.pgr file stores all the information that PowerGREP itself uses to display the results on the Results panel, without requiring access to the files that were searched through to produce the results.

PowerGREP XML Schema

All five file formats used by PowerGREP are based on a single XML Schema. You can download the schema definition file from <http://www.powergrep.com/powergrep40.xsd>. When creating file selection and action files by yourself, make sure to validate them against the schema. PowerGREP does not validate files against the schema, and makes little effort to display helpful error messages.

In order to avoid inconsistencies, there is no separate documentation for PowerGREP's file formats. The XML schema is annotated, and serves as both human-readable documentation and machine-readable specification.

The XML schema is laid out in a bottom-up fashion. The root element, which is always "powergrep", is defined as the last element in the schema. If you collapse all nodes under xsd:schema you will get an overview of all the types the schema defines.

Part 4

Regular Expression Tutorial

1. Regular Expression Tutorial

In this tutorial, I will teach you all you need to know to be able to craft powerful time-saving regular expressions. I will start with the most basic concepts, so that you can follow this tutorial even if you know nothing at all about regular expressions yet.

But I will not stop there. I will also explain how a regular expression engine works on the inside, and alert you at the consequences. This will help you to understand quickly why a particular regex does not do what you initially expected. It will save you lots of guesswork and head scratching when you need to write more complex regexes.

What Regular Expressions Are Exactly - Terminology

Basically, a regular expression is a pattern describing a certain amount of text. Their name comes from the mathematical theory on which they are based. But we will not dig into that. Since most people including myself are lazy to type, you will usually find the name abbreviated to regex or regexp. I prefer regex, because it is easy to pronounce the plural “regexes”. In this book, regular expressions are printed between guillemots: «regex». They clearly separate the pattern from the surrounding text and punctuation.

This first example is actually a perfectly valid regex. It is the most basic pattern, simply matching the literal text „regex”. A “match” is the piece of text, or sequence of bytes or characters that pattern was found to correspond to by the regex processing software. Matches are indicated by double quotation marks, with the left one at the base of the line.

«\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b» is a more complex pattern. It describes a series of letters, digits, dots, underscores, percentage signs and hyphens, followed by an at sign, followed by another series of letters, digits and hyphens, finally followed by a single dot and between two and four letters. In other words: this pattern describes an email address.

With the above regular expression pattern, you can search through a text file to find email addresses, or verify if a given string looks like an email address. In this tutorial, I will use the term “string” to indicate the text that I am applying the regular expression to. I will indicate strings using regular double quotes. The term “string” or “character string” is used by programmers to indicate a sequence of characters. In practice, you can use regular expressions with whatever data you can access using the application or programming language you are working with.

Different Regular Expression Engines

A regular expression “engine” is a piece of software that can process regular expressions, trying to match the pattern to the given string. Usually, the engine is part of a larger application and you do not access the engine directly. Rather, the application will invoke it for you when needed, making sure the right regular expression is applied to the right file or data.

As usual in the software world, different regular expression engines are not fully compatible with each other. It is not possible to describe every kind of engine and regular expression syntax (or “flavor”) in this tutorial. I will focus on the regex flavor used by Perl 5, for the simple reason that this regex flavor is the most popular

one, and deservedly so. Many more recent regex engines are very similar, but not identical, to the one of Perl 5. Examples are the open source PCRE engine (used in many tools and languages like PHP), the .NET regular expression library, and the regular expression package included with version 1.4 and later of the Java JDK. I will point out to you whenever differences in regex flavors are important, and which features are specific to the Perl-derivatives mentioned above.

Give Regexes a First Try

You can easily try the following yourself in a text editor that supports regular expressions, such as EditPad Pro. If you do not have such an editor, you can download the free evaluation version of EditPad Pro to try this out. EditPad Pro's regex engine is fully functional in the demo version. As a quick test, copy and paste the text of this page into EditPad Pro. Then select Search|Show Search Panel from the menu. In the search pane that appears near the bottom, type in «regex» in the box labeled "Search Text". Mark the "Regular expression" checkbox, and click the Find First button. This is the leftmost button on the search panel. See how EditPad Pro's regex engine finds the first match. Click the Find Next button, which sits next to the Find First button, to find further matches. When there are no further matches, the Find Next button's icon will flash briefly.

Now try to search using the regex «reg(ular expressions?|ex(p|es)?)». This regex will find all names, singular and plural, I have used on this page to say "regex". If we only had plain text search, we would have needed 5 searches. With regexes, we need just one search. Regexes save you time when using a tool like EditPad Pro. Select Count Matches in the Search menu to see how many times this regular expression can match the file you have open in EditPad Pro.

If you are a programmer, your software will run faster since even a simple regex engine applying the above regex once will outperform a state of the art plain text search algorithm searching through the data five times. Regular expressions also reduce development time. With a regex engine, it takes only one line (e.g. in Perl, PHP, Java or .NET) or a couple of lines (e.g. in C using PCRE) of code to, say, check if the user's input looks like a valid email address.

2. Regex Tutorial Table of Contents

This regular expression tutorial teaches you every aspect of regular expressions. Each topic assumes you have read and understood all previous topics. So if you are new to regular expressions, I recommend you read the topics in the order presented.

Introduction

The introduction indicates the scope of the tutorial and which regex flavors will be discussed. It also introduces basic terminology.

Literal Characters and Special Characters

The simplest regex consists of only literal characters. Certain characters have special meanings in a regex and have to be escaped. Escaping rules may get a bit complicated when using regexes in software source code. How to enter non-printable characters.

How a Regex Engine Works Internally

First look at the internals of the regular expression engine's internals. Later topics will build on this information. Knowing the engine's internals will greatly help you to craft regexes that match what you intended, and not match what you do not want.

Character Classes or Character Sets

A character class or character set matches a single character out of several possible characters, consisting of individual characters and/or ranges of characters. A negated character class matches a single character not in the character class. Shorthand character classes allow you to use common sets quickly.

The Dot

The dot matches any character, though usually not line break characters unless you change an option.

Start of String and End of String Anchors

Anchors are zero-width. They do not match any characters, but rather a position. The caret and the dollar sign match at the start and the end of the string. Depending on your regex flavor and its options, they can match at the start and the end of a line as well.

Word Boundaries

Word boundaries are like anchors, but match at the start of a word and the end of a word. However, most regex flavors define the concept of a "word" differently than your English teacher in grade school.

Alternation

By separating different sub-regexes with vertical bars, you can tell the regex engine to attempt them from left to right, and return success as soon as one of them can be matched.

Optional Items

Putting a question mark after an item tells the regex engine to match the item if possible, but continue anyway (rather than admit defeat) if it cannot be matched.

Repetition Using Various Quantifiers

Three styles of operators, the star, the plus, and curly braces, allow you to repeat an item zero or more times, once or more, or an arbitrary number of times. It is important to understand that these quantifiers are “greedy” by default, unless you explicitly make them “lazy”.

Grouping and Backreferences

By placing round brackets around part of the regex, you tell the engine to treat that part as a single item when applying operators such as quantifiers. With round brackets, you can also create backreferences that allow you to reuse the text matched by part of the regex inside the regular expression, or later in the replacement text of a search and replace operation. Backreferences are also very useful for extracting parts from a string in a programming language.

Unicode Characters and Properties

If your regular expression flavor supports Unicode, then you can use special Unicode regex tokens to match specific Unicode characters, or to match any character that has a certain Unicode property or is part of a particular Unicode script or block.

Mode Modifiers

Change matching modes such as “case insensitive” for specific parts of the regular expression.

Atomic Grouping and Possessive Quantifiers

Nested quantifiers can cause an exponentially increasing amount of backtracking that brings the regex engine to a grinding halt. Atomic grouping and possessive quantifiers provide a solution.

Lookaround with Zero-Width Assertions, part 1 and part 2

Lookahead and lookbehind (collectively lookahead) are zero-width. With positive lookahead, you can specify multiple requirements (sub-regexes) to be applied to the same part of the string. With negative lookahead, you can invert the result of a regex match (i.e. match something that does not match something else).

Continuing from The Previous Match Attempt

Forcing a regex match to start at the end of a previous match provides an efficient way to parse text data.

Combining Positive and Negative Lookaround with Conditionals

A conditional is a special construct that will first evaluate a lookaround, and then execute one sub-regex if the lookaround succeeds, and another sub-regex if the lookaround fails.

XML Character Classes

XML Schema regular expressions support four shorthand character classes to match XML names. They also introduce a handy feature called “character class subtraction”, which is now also available in the JGsoft and .NET regex engines.

POSIX Bracket Expressions

If you are using a POSIX-compliant regular expression engine, you can use POSIX bracket expressions to match locale-dependent characters.

Adding Comments

Some regex flavors allow you to add comments to make complex regular expressions easier to understand.

Free-Spacing Mode

Splitting a regular expression into multiple lines, adding comments and whitespace, makes it even more readable.

3. Literal Characters

The most basic regular expression consists of a single literal character, e.g.: «a». It will match the first occurrence of that character in the string. If the string is “Jack is a boy”, it will match the „a” after the “J”. The fact that this “a” is in the middle of the word does not matter to the regex engine. If it matters to you, you will need to tell that to the regex engine by using word boundaries. We will get to that later.

This regex can match the second „a” too. It will only do so when you tell the regex engine to start searching through the string after the first match. In a text editor, you can do so by using its “Find Next” or “Search Forward” function. In a programming language, there is usually a separate function that you can call to continue searching through the string after the previous match.

Similarly, the regex «cat» will match „cat” in “About cats and dogs”. This regular expression consists of a series of three literal characters. This is like saying to the regex engine: find a «c», immediately followed by an «a», immediately followed by a «t».

Note that regex engines are case sensitive by default. «cat» does not match “Cat”, unless you tell the regex engine to ignore differences in case.

Special Characters

Because we want to do more than simply search for literal pieces of text, we need to reserve certain characters for special use. In the regex flavors discussed in this tutorial, there are 11 characters with special meanings: the opening square bracket «[», the backslash «\», the caret «^», the dollar sign «\$», the period or dot «.», the vertical bar or pipe symbol «|», the question mark «?», the asterisk or star «*», the plus sign «+», the opening round bracket «(» and the closing round bracket «)». These special characters are often called “metacharacters”.

If you want to use any of these characters as a literal in a regex, you need to escape them with a backslash. If you want to match „1+1=2”, the correct regex is «1\+1=2». Otherwise, the plus sign will have a special meaning.

Note that «1+1=2», with the backslash omitted, is a valid regex. So you will not get an error message. But it will not match “1+1=2”. It would match „111=2” in “123+111=234”, due to the special meaning of the plus character.

If you forget to escape a special character where its use is not allowed, such as in «+1», then you will get an error message.

Most regular expression flavors treat the brace «{» as a literal character, unless it is part of a repetition operator like «{1,3}». So you generally do not need to escape it with a backslash, though you can do so if you want. An exception to this rule is the java.util.regex package: it requires all literal braces to be escaped.

All other characters should not be escaped with a backslash. That is because the backslash is also a special character. The backslash in combination with a literal character can create a regex token with a special meaning. E.g. «\d» will match a single digit from 0 to 9.

Escaping a single metacharacter with a backslash works in all regular expression flavors. Many flavors also support the `\Q . . \E` escape sequence. All the characters between the `\Q` and the `\E` are interpreted as literal characters. E.g. `«\Q*\d+*\E»` matches the literal text `„*\d+*”`. The `\E` may be omitted at the end of the regex, so `«\Q*\d+*»` is the same as `«\Q*\d+*\E»`. This syntax is supported by the JGsoft engine, Perl, PCRE and Java, both inside and outside character classes. However, in Java, this feature does not work correctly in JDK 1.4 and 1.5 when used in a character class or followed by a quantifier.

Special Characters and Programming Languages

If you are a programmer, you may be surprised that characters like the single quote and double quote are not special characters. That is correct. When using a regular expression or grep tool like PowerGREP or the search function of a text editor like EditPad Pro, you should not escape or repeat the quote characters like you do in a programming language.

In your source code, you have to keep in mind which characters get special treatment inside strings by your programming language. That is because those characters will be processed by the compiler, before the regex library sees the string. So the regex `«1\+1=2»` must be written as `"1\\+1=2"` in C++ code. The C++ compiler will turn the escaped backslash in the source code into a single backslash in the string that is passed on to the regex library. To match `„c:\temp”`, you need to use the regex `«c:\\temp»`. As a string in C++ source code, this regex becomes `"c:\\\\temp"`. Four backslashes to match a single one indeed.

See the tools and languages section in this book for more information on how to use regular expressions in various programming languages.

Non-Printable Characters

You can use special character sequences to put non-printable characters in your regular expression. Use `«\t»` to match a tab character (ASCII 0x09), `«\r»` for carriage return (0x0D) and `«\n»` for line feed (0x0A). More exotic non-printables are `«\a»` (bell, 0x07), `«\e»` (escape, 0x1B), `«\f»` (form feed, 0x0C) and `«\v»` (vertical tab, 0x0B). Remember that Windows text files use `“\r\n”` to terminate lines, while UNIX text files use `“\n”`.

You can include any character in your regular expression if you know its hexadecimal ASCII or ANSI code for the character set that you are working with. In the Latin-1 character set, the copyright symbol is character 0xA9. So to search for the copyright symbol, you can use `«\xA9»`. Another way to search for a tab is to use `«\x09»`. Note that the leading zero is required.

Most regex flavors also support the tokens `«\cA»` through `«\cZ»` to insert ASCII control characters. The letter after the backslash is always a lowercase c. The second letter is an uppercase letter A through Z, to indicate Control+A through Control+Z. These are equivalent to `«\x01»` through `«\x1A»` (26 decimal). E.g. `«\cM»` matches a carriage return, just like `«\r»` and `«\x0D»`. In XML Schema regular expressions, `«\c»` is a shorthand character class that matches any character allowed in an XML name.

If your regular expression engine supports Unicode, use `«\uFFFF»` rather than `«\xFF»` to insert a Unicode character. The euro currency sign occupies code point 0x20AC. If you cannot type it on your keyboard, you can insert it into a regular expression with `«\u20AC»`.

4. First Look at How a Regex Engine Works Internally

Knowing how the regex engines will enable you to craft better regexes more easily. It will help you understand quickly why a particular regex does not do what you initially expected. This will save you lots of guesswork and head scratching when you need to write more complex regexes.

There are two kinds of regular expression engines: text-directed engines, and regex-directed engines. Jeffrey Friedl calls them DFA and NFA engines, respectively. All the regex flavors treated in this tutorial are based on regex-directed engines. This is because certain very useful features, such as lazy quantifiers and backreferences, can only be implemented in regex-directed engines. No surprise that this kind of engine is more popular.

Notable tools that use text-directed engines are `awk`, `egrep`, `flex`, `lex`, `MySQL` and `Procmail`. For `awk` and `egrep`, there are a few versions of these tools that use a regex-directed engine.

You can easily find out whether the regex flavor you intend to use has a text-directed or regex-directed engine. If backreferences and/or lazy quantifiers are available, you can be certain the engine is regex-directed. You can do the test by applying the regex `«regex|regex not»` to the string `“regex not”`. If the resulting match is only `„regex”`, the engine is regex-directed. If the result is `„regex not”`, then it is text-directed. The reason behind this is that the regex-directed engine is “eager”.

In this tutorial, after introducing a new regex token, I will explain step by step how the regex engine actually processes that token. This inside look may seem a bit long-winded at certain times. But understanding how the regex engine works will enable you to use its full power and help you avoid common mistakes.

The Regex-Directed Engine Always Returns the Leftmost Match

This is a very important point to understand: a regex-directed engine will always return the leftmost match, even if a “better” match could be found later. When applying a regex to a string, the engine will start at the first character of the string. It will try all possible permutations of the regular expression at the first character. Only if all possibilities have been tried and found to fail, will the engine continue with the second character in the text. Again, it will try all possible permutations of the regex, in exactly the same order. The result is that the regex-directed engine will return the *leftmost* match.

When applying `«cat»` to `“He captured a catfish for his cat.”`, the engine will try to match the first token in the regex `«c»` to the first character in the match `“H”`. This fails. There are no other possible permutations of this regex, because it merely consists of a sequence of literal characters. So the regex engine tries to match the `«c»` with the `“e”`. This fails too, as does matching the `«c»` with the space. Arriving at the 4th character in the match, `«c»` matches `„c”`. The engine will then try to match the second token `«a»` to the 5th character, `„a”`. This succeeds too. But then, `«t»` fails to match `“p”`. At that point, the engine knows the regex cannot be matched starting at the 4th character in the match. So it will continue with the 5th: `“a”`. Again, `«c»` fails to match here and the engine carries on. At the 15th character in the match, `«c»` again matches `„c”`. The engine then proceeds to attempt to match the remainder of the regex at character 15 and finds that `«a»` matches `„a”` and `«t»` matches `„t”`.

The entire regular expression could be matched starting at character 15. The engine is “eager” to report a match. It will therefore report the first three letters of `catfish` as a valid match. The engine never proceeds beyond this point to see if there are any “better” matches. The first match is considered good enough.

In this first example of the engine's internals, our regex engine simply appears to work like a regular text search routine. A text-directed engine would have returned the same result too. However, it is important that you can follow the steps the engine takes in your mind. In following examples, the way the engine works will have a profound impact on the matches it will find. Some of the results may be surprising. But they are always logical and predetermined, once you know how the engine works.

5. Character Classes or Character Sets

With a "character class", also called "character set", you can tell the regex engine to match only one out of several characters. Simply place the characters you want to match between square brackets. If you want to match an a or an e, use `«[ae]»`. You could use this in `«gr[ae]y»` to match either „gray” or „grey”. Very useful if you do not know whether the document you are searching through is written in American or British English.

A character class matches only a single character. `«gr[ae]y»` will not match “graay”, “graey” or any such thing. The order of the characters inside a character class does not matter. The results are identical.

You can use a hyphen inside a character class to specify a range of characters. `«[0-9]»` matches a *single* digit between 0 and 9. You can use more than one range. `«[0-9a-fA-F]»` matches a single hexadecimal digit, case insensitively. You can combine ranges and single characters. `«[0-9a-fxA-FX]»` matches a hexadecimal digit or the letter X. Again, the order of the characters and the ranges does not matter.

Useful Applications

Find a word, even if it is misspelled, such as `«sep[ae]r[ae]te»` or `«li[cs]en[cs]e»`.

Find an identifier in a programming language with `«[A-Za-z_] [A-Za-z_0-9]*»`.

Find a C-style hexadecimal number with `«0[xX] [A-Fa-f0-9]+»`.

Negated Character Classes

Typing a caret after the opening square bracket will negate the character class. The result is that the character class will match any character that is *not* in the character class. Unlike the dot, negated character classes also match (invisible) line break characters.

It is important to remember that a negated character class still must match a character. `«q[^u]»` does *not* mean: “a q not followed by a u”. It means: “a q followed by a character that is not a u”. It will not match the q in the string “Iraq”. It will match the q and the space after the q in “Iraq is a country”. Indeed: the space will be part of the overall match, because it is the “character that is not a u” that is matched by the negated character class in the above regexp. If you want the regex to match the q, and only the q, in both strings, you need to use negative lookahead: `«q(?!u)»`. But we will get to that later.

Metacharacters Inside Character Classes

Note that the only special characters or metacharacters inside a character class are the closing bracket (]), the backslash (\), the caret (^) and the hyphen (-). The usual metacharacters are normal characters inside a character class, and do not need to be escaped by a backslash. To search for a star or plus, use `«[+*]»`. Your regex will work fine if you escape the regular metacharacters inside a character class, but doing so significantly reduces readability.

To include a backslash as a character without any special meaning inside a character class, you have to escape it with another backslash. «`[\x]`» matches a backslash or an x. The closing bracket (`]`), the caret (`^`) and the hyphen (`-`) can be included by escaping them with a backslash, or by placing them in a position where they do not take on their special meaning. I recommend the latter method, since it improves readability. To include a caret, place it anywhere except right after the opening bracket. «`[x^]`» matches an x or a caret. You can put the closing bracket right after the opening bracket, or the negating caret. «`[]x]`» matches a closing bracket or an x. «`[^]x]`» matches any character that is not a closing bracket or an x. The hyphen can be included right after the opening bracket, or right before the closing bracket, or right after the negating caret. Both «`[-x]`» and «`[x-]`» match an x or a hyphen.

You can use all non-printable characters in character classes just like you can use them outside of character classes. E.g. «`[$\u20AC]`» matches a dollar or euro sign, assuming your regex flavor supports Unicode.

The JGsoft engine, Perl and PCRE also support the `\Q..\E` sequence inside character classes to escape a string of characters. E.g. «`[\Q[-]\E]`» matches „`[`”, „`]`”, „`-`” or „`]`”.

POSIX regular expressions treat the backslash as a literal character inside character classes. This means you can't use backslashes to escape the closing bracket (`]`), the caret (`^`) and the hyphen (`-`). To use these characters, position them as explained above in this section. This also means that special tokens like shorthands are not available in POSIX regular expressions. See the tutorial topic on POSIX bracket expressions for more information.

Shorthand Character Classes

Since certain character classes are used often, a series of shorthand character classes are available. «`\d`» is short for «`[0-9]`».

«`\w`» stands for “word character”, usually «`[A-Za-z0-9_]`». Notice the inclusion of the underscore and digits.

«`\s`» stands for “whitespace character”. Again, which characters this actually includes, depends on the regex flavor. In all flavors discussed in this tutorial, it includes «`[\t\r\n]`». That is: «`\s`» will match a space, a tab or a line break. Some flavors include additional, rarely used non-printable characters such as vertical tab and form feed.

The flavor comparison shows “ascii only” for flavors that match only the ASCII characters listed in the previous paragraphs. With flavors marked as “YES”, letters, digits and space characters from other languages or Unicode are also included in the shorthand classes. In the screen shot, you can see the characters matched by «`\w`» in RegxBuddy using various scripts. Notice that JavaScript uses ASCII for «`\d`» and «`\w`», but Unicode for «`\s`». XML does it the other way around. Python offers flags to control what the shorthands should match.



Shorthand character classes can be used both inside and outside the square brackets. «\s\d» matches a whitespace character followed by a digit. «[\s\d]» matches a single character that is either whitespace or a digit. When applied to “1 + 2 = 3”, the former regex will match „ 2” (space two), while the latter matches „1” (one). «[\da-fA-F]» matches a hexadecimal digit, and is equivalent to «[0-9a-fA-F]».

Negated Shorthand Character Classes

The above three shorthands also have negated versions. «\D» is the same as «[^\d]», «\W» is short for «[^\w]» and «\S» is the equivalent of «[^\s]».

Be careful when using the negated shorthands inside square brackets. «[\D\S]» is *not* the same as «[^\d\s]». The latter will match any character that is not a digit or whitespace. So it will match „x”, but not “8”. The former, however, will match any character that is either not a digit, or is not whitespace. Because a digit is not whitespace, and whitespace is not a digit, «[\D\S]» will match any character, digit, whitespace or otherwise.

Repeating Character Classes

If you repeat a character class by using the «?», «*» or «+» operators, you will repeat the entire character class, and not just the character that it matched. The regex «[0-9]+» can match „837” as well as „222”.

If you want to repeat the matched character, rather than the class, you will need to use backreferences. «([0-9])\1+» will match „222” but not “837”. When applied to the string “833337”, it will match „3333” in the middle of this string. If you do not want that, you need to use lookahead and lookbehind.

But I digress. I did not yet explain how character classes work inside the regex engine. Let us take a look at that first.

Looking Inside The Regex Engine

As I already said: the order of the characters inside a character class does not matter. «gr[ae]y» will match „grey” in “Is his hair grey or gray?”, because that is the *leftmost match*. We already saw how the engine applies a regex consisting only of literal characters. Below, I will explain how it applies a regex that has more than one permutation. That is: «gr[ae]y» can match both „gray” and „grey”.

Nothing noteworthy happens for the first twelve characters in the string. The engine will fail to match «g» at every step, and continue with the next character in the string. When the engine arrives at the 13th character, „g” is matched. The engine will then try to match the remainder of the regex with the text. The next token in the regex is the literal «r», which matches the next character in the text. So the third token, «[ae]» is attempted at the next character in the text (“e”). The character class gives the engine two options: match «a» or match «e». It will first attempt to match «a», and fail.

But because we are using a regex-directed engine, it must continue trying to match all the other permutations of the regex pattern before deciding that the regex cannot be matched with the text starting at character 13. So it will continue with the other option, and find that «e» matches „e”. The last regex token is «y», which can be matched with the following character as well. The engine has found a complete match with the text starting at character 13. It will return „grey” as the match result, and look no further. Again, the *leftmost match* was returned, even though we put the «a» first in the character class, and „gray” could have been matched in the string. But the engine simply did not get that far, because another equally valid match was found to the left of it.

6. The Dot Matches (Almost) Any Character

In regular expressions, the dot or period is one of the most commonly used metacharacters. Unfortunately, it is also the most commonly misused metacharacter.

The dot matches a single character, without caring what that character is. The only exception are newline characters. In all regex flavors discussed in this tutorial, the dot will *not* match a newline character by default. So by default, the dot is short for the negated character class «`[\n]`» (UNIX regex flavors) or «`[\r\n]`» (Windows regex flavors).

This exception exists mostly because of historic reasons. The first tools that used regular expressions were line-based. They would read a file line by line, and apply the regular expression separately to each line. The effect is that with these tools, the string could never contain newlines, so the dot could never match them.

Modern tools and languages can apply regular expressions to very large strings or even entire files. All regex flavors discussed here have an option to make the dot match all characters, including newlines. In RegxBuddy, EditPad Pro or PowerGREP, you simply tick the checkbox labeled “dot matches newline”.

In Perl, the mode where the dot also matches newlines is called “single-line mode”. This is a bit unfortunate, because it is easy to mix up this term with “multi-line mode”. Multi-line mode only affects anchors, and single-line mode only affects the dot. You can activate single-line mode by adding an `s` after the regex code, like this: `m/^\regex$/s;`

Other languages and regex libraries have adopted Perl’s terminology. When using the regex classes of the .NET framework, you activate this mode by specifying `RegexOptions.Singleline`, such as in `Regex.Match("string", "regex", RegexOptions.Singleline)`.

In all programming languages and regex libraries I know, activating single-line mode has no effect other than making the dot match newlines. So if you expose this option to your users, please give it a clearer label like was done in RegxBuddy, EditPad Pro and PowerGREP.

JavaScript and VBScript do not have an option to make the dot match line break characters. In those languages, you can use a character class such as «`[\s\S]`» to match any character. This character matches a character that is either a whitespace character (including line break characters), or a character that is not a whitespace character. Since all characters are either whitespace or non-whitespace, this character class matches any character.

Use The Dot Sparingly

The dot is a very powerful regex metacharacter. It allows you to be lazy. Put in a dot, and everything will match just fine when you test the regex on valid data. The problem is that the regex will also match in cases where it should not match. If you are new to regular expressions, some of these cases may not be so obvious at first.

I will illustrate this with a simple example. Let’s say we want to match a date in `mm/dd/yy` format, but we want to leave the user the choice of date separators. The quick solution is «`\d\d.\d\d.\d\d`». Seems fine at first. It will match a date like `„02/12/03”` just fine. Trouble is: `„02512703”` is also considered a valid date by

this regular expression. In this match, the first dot matched „5”, and the second matched „7”. Obviously not what we intended.

«\d\d[- /.]\d\d[- /.]\d\d» is a better solution. This regex allows a dash, space, dot and forward slash as date separators. Remember that the dot is not a metacharacter inside a character class, so we do not need to escape it with a backslash.

This regex is still far from perfect. It matches „99/99/99” as a valid date. «[0-1]\d[- /.][0-3]\d[- /.]\d\d» is a step ahead, though it will still match „19/39/99”. How perfect you want your regex to be depends on what you want to do with it. If you are validating user input, it has to be perfect. If you are parsing data files from a known source that generates its files in the same way every time, our last attempt is probably more than sufficient to parse the data without errors. You can find a better regex to match dates in the example section.

Use Negated Character Sets Instead of the Dot

I will explain this in depth when I present you the repeat operators star and plus, but the warning is important enough to mention it here as well. I will illustrate with an example.

Suppose you want to match a double-quoted string. Sounds easy. We can have any number of any character between the double quotes, so «".*» seems to do the trick just fine. The dot matches any character, and the star allows the dot to be repeated any number of times, including zero. If you test this regex on “Put a "string" between double quotes”, it will match „"string” just fine. Now go ahead and test it on “Houston, we have a problem with "string one" and "string two". Please respond.”

Ouch. The regex matches „"string one" and "string two””. Definitely not what we intended. The reason for this is that the star is *greedy*.

In the date-matching example, we improved our regex by replacing the dot with a character class. Here, we will do the same. Our original definition of a double-quoted string was faulty. We do not want any number of *any character* between the quotes. We want any number of characters that are not double quotes or newlines between the quotes. So the proper regex is «"[^"r\n]*».

7. Start of String and End of String Anchors

Thus far, I have explained literal characters and character classes. In both cases, putting one in a regex will cause the regex engine to try to match a single character.

Anchors are a different breed. They do not match any character at all. Instead, they match a position before, after or between characters. They can be used to “anchor” the regex match at a certain position. The caret «`^`» matches the position before the first character in the string. Applying «`^a`» to “abc” matches „a”. «`^b`» will not match “abc” at all, because the «`b`» cannot be matched right after the start of the string, matched by «`^`». See below for the inside view of the regex engine.

Similarly, «`$`» matches right after the last character in the string. «`c$`» matches „c” in “abc”, while «`a$`» does not match at all.

Useful Applications

When using regular expressions in a programming language to validate user input, using anchors is very important. If you use the code `if ($input =~ m/\d+/)` in a Perl script to see if the user entered an integer number, it will accept the input even if the user entered “qsd4fghjk”, because «`\d+`» matches the 4. The correct regex to use is «`^\d+$`». Because “start of string” must be matched before the match of «`\d+`», and “end of string” must be matched right after it, the entire string must consist of digits for «`^\d+$`» to be able to match.

It is easy for the user to accidentally type in a space. When Perl reads from a line from a text file, the line break will also be stored in the variable. So before validating input, it is good practice to trim leading and trailing whitespace. «`^\s+`» matches leading whitespace and «`\s+$`» matches trailing whitespace. In Perl, you could use `$input =~ s/^\s+|\s+$//g`. Handy use of alternation and `/g` allows us to do this in a single line of code.

Using `^` and `$` as Start of Line and End of Line Anchors

If you have a string consisting of multiple lines, like “first line\nsecond line” (where `\n` indicates a line break), it is often desirable to work with lines, rather than the entire string. Therefore, all the regex engines discussed in this tutorial have the option to expand the meaning of both anchors. «`^`» can then match at the start of the string (before the “f” in the above string), as well as after each line break (between “\n” and “s”). Likewise, «`$`» will still match at the end of the string (after the last “e”), and also before every line break (between “e” and “\n”).

In text editors like EditPad Pro or GNU Emacs, and regex tools like PowerGREP, the caret and dollar always match at the start and end of each line. This makes sense because those applications are designed to work with entire files, rather than short strings.

In all programming languages and libraries discussed in this book, except Ruby, you have to explicitly activate this extended functionality. It is traditionally called “multi-line mode”. In Perl, you do this by adding an `m` after the regex code, like this: `m/^\d+$/m`; In .NET, the anchors match before and after newlines when you specify `RegexOptions.Multiline`, such as in `Regex.Match("string", "regex", RegexOptions.Multiline)`.

Permanent Start of String and End of String Anchors

«\A» only ever matches at the start of the string. Likewise, «\Z» only ever matches at the end of the string. These two tokens never match at line breaks. This is true in all regex flavors discussed in this tutorial, even when you turn on “multiline mode”. In EditPad Pro and PowerGREP, where the caret and dollar always match at the start and end of lines, «\A» and «\Z» only match at the start and the end of the entire file.

JavaScript, POSIX and XML do not support «\A» and «\Z». You’re stuck with using the caret and dollar for this purpose.

The GNU extensions to POSIX regular expressions use «\`» (backtick) to match the start of the string, and «\’» (single quote) to match the end of the string.

Zero-Length Matches

We saw that the anchors match at a position, rather than matching a character. This means that when a regex only consists of one or more anchors, it can result in a zero-length match. Depending on the situation, this can be very useful or undesirable. Using «^\d*\$» to test if the user entered a number (notice the use of the star instead of the plus), would cause the script to accept an empty string as a valid input. See below.

However, matching only a position can be very useful. In email, for example, it is common to prepend a “greater than” symbol and a space to each line of the quoted message. In VB.NET, we can easily do this with `Dim Quoted as String = Regex.Replace(Original, "^", "> ", RegexOptions.Multiline)`. We are using multi-line mode, so the regex «^» matches at the start of the quoted message, and after each newline. The `Regex.Replace` method will remove the regex match from the string, and insert the replacement string (greater than symbol and a space). Since the match does not include any characters, nothing is deleted. However, the match does include a starting position, and the replacement string is inserted there, just like we want it.

Strings Ending with a Line Break

Even though «\Z» and «\$» only match at the end of the string (when the option for the caret and dollar to match at embedded line breaks is off), there is one exception. If the string ends with a line break, then «\Z» and «\$» will match at the position before that line break, rather than at the very end of the string. This “enhancement” was introduced by Perl, and is copied by many regex flavors, including Java, .NET and PCRE. In Perl, when reading a line from a file, the resulting string will end with a line break. Reading a line from a file with the text “joe” results in the string “joe\n”. When applied to this string, both «^[a-z]+\$» and «\A[a-z]+\Z» will match „joe”.

If you only want a match at the absolute very end of the string, use «\z» (lower case z instead of upper case Z). «\A[a-z]+\z» does not match “joe\n”. «\z» matches after the line break, which is not matched by the character class.

Looking Inside the Regex Engine

Let's see what happens when we try to match `«^4$»` to `"749\n486\n4"` (where `\n` represents a newline character) in multi-line mode. As usual, the regex engine starts at the first character: `"7"`. The first token in the regular expression is `«^»`. Since this token is a zero-width token, the engine does not try to match it with the character, but rather with the position before the character that the regex engine has reached so far. `«^»` indeed matches the position before `"7"`. The engine then advances to the next regex token: `«4»`. Since the previous token was zero-width, the regex engine does *not* advance to the next character in the string. It remains at `"7"`. `«4»` is a literal character, which does not match `"7"`. There are no other permutations of the regex, so the engine starts again with the first regex token, at the next character: `"4"`. This time, `«^»` cannot match at the position before the 4. This position is preceded by a character, and that character is not a newline. The engine continues at `"9"`, and fails again. The next attempt, at `"\n"`, also fails. Again, the position before `"\n"` is preceded by a character, `"9"`, and that character is not a newline.

Then, the regex engine arrives at the second `"4"` in the string. The `«^»` can match at the position before the `"4"`, because it is preceded by a newline character. Again, the regex engine advances to the next regex token, `«4»`, but does not advance the character position in the string. `«4»` matches `„4"`, and the engine advances both the regex token and the string character. Now the engine attempts to match `«$»` at the position before (indeed: before) the `"8"`. The dollar cannot match here, because this position is followed by a character, and that character is not a newline.

Yet again, the engine must try to match the first token again. Previously, it was successfully matched at the second `"4"`, so the engine continues at the next character, `"8"`, where the caret does not match. Same at the six and the newline.

Finally, the regex engine tries to match the first token at the third `"4"` in the string. With success. After that, the engine successfully matches `«4»` with `„4"`. The current regex token is advanced to `«$»`, and the current character is advanced to the very last position in the string: the void after the string. No regex token that needs a character to match can match here. Not even a negated character class. However, we are trying to match a dollar sign, and the mighty dollar is a strange beast. It is zero-width, so it will try to match the position before the current character. It does not matter that this "character" is the void after the string. In fact, the dollar will check the current character. It must be either a newline, or the void after the string, for `«$»` to match the position before the current character. Since that is the case after the example, the dollar matches successfully.

Since `«$»` was the last token in the regex, the engine has found a successful match: the last `„4"` in the string.

Another Inside Look

Earlier I mentioned that `«^\d*$»` would successfully match an empty string. Let's see why.

There is only one "character" position in an empty string: the void after the string. The first token in the regex is `«^»`. It matches the position before the void after the string, because it is preceded by the void before the string. The next token is `«\d*»`. As we will see later, one of the star's effects is that it makes the `«\d»`, in this case, optional. The engine will try to match `«\d»` with the void after the string. That fails, but the star turns the failure of the `«\d»` into a zero-width success. The engine will proceed with the next regex token, without advancing the position in the string. So the engine arrives at `«$»`, and the void after the string. We already saw that those match. At this point, the entire regex has matched the empty string, and the engine reports success.

Caution for Programmers

A regular expression such as «\$» all by itself can indeed match after the string. If you would query the engine for the character position, it would return the length of the string if string indices are zero-based, or the length+1 if string indices are one-based in your programming language. If you would query the engine for the length of the match, it would return zero.

What you have to watch out for is that `String[Regex.MatchPosition]` may cause an access violation or segmentation fault, because `MatchPosition` can point to the void after the string. This can also happen with «^» and «^\$» if the last character in the string is a newline.

8. Word Boundaries

The metacharacter `\b` is an anchor like the caret and the dollar sign. It matches at a position that is called a “word boundary”. This match is zero-length.

There are three different positions that qualify as word boundaries:

- Before the first character in the string, if the first character is a word character.
- After the last character in the string, if the last character is a word character.
- Between two characters in the string, where one is a word character and the other is not a word character.

Simply put: `\b` allows you to perform a “whole words only” search using a regular expression in the form of `\bword\b`. A “word character” is a character that can be used to form words. All characters that are not “word characters” are “non-word characters”.

In all flavors, the characters `[a-zA-Z0-9_]` are word characters. These are also matched by the short-hand character class `\w`. Flavors showing “ascii” for word boundaries in the flavor comparison recognize only these as word characters. Flavors showing “YES” also recognize letters and digits from other languages or all of Unicode as word characters. Notice that Java supports Unicode for `\b` but not for `\w`. Python offers flags to control which characters are word characters (affecting both `\b` and `\w`).

In Perl and the other regex flavors discussed in this tutorial, there is only one metacharacter that matches both before a word and after a word. This is because any position between characters can never be both at the start and at the end of a word. Using only one operator makes things easier for you.

Since digits are considered to be word characters, `\b4\b` can be used to match a 4 that is not part of a larger number. This regex will not match “44 sheets of a4”. So saying “`\b` matches before and after an alphanumeric sequence” is more exact than saying “before and after a word”.

Negated Word Boundary

`\B` is the negated version of `\b`. `\B` matches at every position where `\b` does not. Effectively, `\B` matches at any position between two word characters as well as at any position between two non-word characters.

Looking Inside the Regex Engine

Let’s see what happens when we apply the regex `\bis\b` to the string “This island is beautiful”. The engine starts with the first token `\b` at the first character “T”. Since this token is zero-length, the position before the character is inspected. `\b` matches here, because the T is a word character and the character before it is the void before the start of the string. The engine continues with the next token: the literal `i`. The engine does not advance to the next character in the string, because the previous regex token was zero-width. `i` does not match “T”, so the engine retries the first token at the next character position.

«\b» cannot match at the position between the “T” and the “h”. It cannot match between the “h” and the “i” either, and neither between the “i” and the “s”.

The next character in the string is a space. «\b» matches here because the space is not a word character, and the preceding character is. Again, the engine continues with the «i» which does not match with the space.

Advancing a character and restarting with the first regex token, «\b» matches between the space and the second “i” in the string. Continuing, the regex engine finds that «i» matches „i” and «s» matches „s”. Now, the engine tries to match the second «\b» at the position before the “l”. This fails because this position is between two word characters. The engine reverts to the start of the regex and advances one character to the “s” in “island”. Again, the «\b» fails to match and continues to do so until the second space is reached. It matches there, but matching the «i» fails.

But «\b» matches at the position before the third “i” in the string. The engine continues, and finds that «i» matches „i” and «s» matches „s”. The last token in the regex, «\b», also matches at the position before the third space in the string because the space is not a word character, and the character before it is.

The engine has successfully matched the word „is” in our string, skipping the two earlier occurrences of the characters i and s. If we had used the regular expression «i s», it would have matched the „is” in “This”.

Tcl Word Boundaries

Word boundaries, as described above, are supported by most regular expression flavors. Notable exceptions are the POSIX and XML Schema flavors, which don’t support word boundaries at all. Tcl uses a different syntax.

In Tcl, «\b» matches a backspace character, just like «\x08» in most regex flavors (including Tcl’s). «\B» matches a single backslash character in Tcl, just like «\» in all other regex flavors (and Tcl too).

Tcl uses the letter “y” instead of the letter “b” to match word boundaries. «\y» matches at any word boundary position, while «\Y» matches at any position that is not a word boundary. These Tcl regex tokens match exactly the same as «\b» and «\B» in Perl-style regex flavors. They don’t discriminate between the start and the end of a word.

Tcl has two more word boundary tokens that do discriminate between the start and end of a word. «\m» matches only at the start of a word. That is, it matches at any position that has a non-word character to the left of it, and a word character to the right of it. It also matches at the start of the string if the first character in the string is a word character. «\M» matches only at the end of a word. It matches at any position that has a word character to the left of it, and a non-word character to the right of it. It also matches at the end of the string if the last character in the string is a word character.

The only regex engine that supports Tcl-style word boundaries (besides Tcl itself) is the JGsoft engine. In PowerGREP and EditPad Pro, «\b» and «\B» are Perl-style word boundaries, and «\y», «\Y», «\m» and «\M» are Tcl-style word boundaries.

In most situations, the lack of «\m» and «\M» tokens is not a problem. «\yword\y» finds “whole words only” occurrences of “word” just like «\mword\M» would. «\Mword\m» could never match anywhere, since «\M» never matches at a position followed by a word character, and «\m» never at a position preceded by one. If your regular expression needs to match characters before or after «\y», you can easily specify in the regex

whether these characters should be word characters or non-word characters. E.g. if you want to match any word, «\y\w+\y» will give the same result as «\m.+ \M». Using «\w» instead of the dot automatically restricts the first «\y» to the start of a word, and the second «\y» to the end of a word. Note that «\y.+ \y» would not work. This regex matches each word, and also each sequence of non-word characters between the words in your subject string. That said, if your flavor supports «\m» and «\M», the regex engine could apply «\m\w+\M» slightly faster than «\y\w+\y», depending on its internal optimizations.

If your regex flavor supports lookahead and lookbehind, you can use «(?<!\w) (?=\w)» to emulate Tcl's «\m» and «(?<=\w) (?!\w)» to emulate «\M». Though quite a bit more verbose, these lookahead constructs match exactly the same as Tcl's word boundaries.

If your flavor has lookahead but not lookbehind, and also has Perl-style word boundaries, you can use «\b(?=\w)» to emulate Tcl's «\m» and «\b(?!\w)» to emulate «\M». «\b» matches at the start or end of a word, and the lookahead checks if the next character is part of a word or not. If it is we're at the start of a word. Otherwise, we're at the end of a word.

GNU Word Boundaries

The GNU extensions to POSIX regular expressions add support for the «\b» and «\B» word boundaries, as described above. GNU also uses its own syntax for start-of-word and end-of-word boundaries. «\<» matches at the start of a word, like Tcl's «\m». «\>» matches at the end of a word, like Tcl's «\M».

9. Alternation with The Vertical Bar or Pipe Symbol

I already explained how you can use character classes to match a single character out of several possible characters. Alternation is similar. You can use alternation to match a single regular expression out of several possible regular expressions.

If you want to search for the literal text «cat» or «dog», separate both options with a vertical bar or pipe symbol: «cat|dog». If you want more options, simply expand the list: «cat|dog|mouse|fish».

The alternation operator has the lowest precedence of all regex operators. That is, it tells the regex engine to match either everything to the left of the vertical bar, or everything to the right of the vertical bar. If you want to limit the reach of the alternation, you will need to use round brackets for grouping. If we want to improve the first example to match whole words only, we would need to use «\b(cat|dog)\b». This tells the regex engine to find a word boundary, then either “cat” or “dog”, and then another word boundary. If we had omitted the round brackets, the regex engine would have searched for “a word boundary followed by cat”, or, “dog” followed by a word boundary.

Remember That The Regex Engine Is Eager

I already explained that the regex engine is eager. It will stop searching as soon as it finds a valid match. The consequence is that in certain situations, the order of the alternatives matters. Suppose you want to use a regex to match a list of function names in a programming language: Get, GetValue, Set or SetValue. The obvious solution is «Get|GetValue|Set|SetValue». Let’s see how this works out when the string is “SetValue”.

The regex engine starts at the first token in the regex, «G», and at the first character in the string, “S”. The match fails. However, the regex engine studied the entire regular expression before starting. So it knows that this regular expression uses alternation, and that the entire regex has not failed yet. So it continues with the second option, being the second «G» in the regex. The match fails again. The next token is the first «S» in the regex. The match succeeds, and the engine continues with the next character in the string, as well as the next token in the regex. The next token in the regex is the «e» after the «S» that just successfully matched. «e» matches „e”. The next token, «t» matches „t”.

At this point, the third option in the alternation has been successfully matched. Because the regex engine is eager, it considers the entire alternation to have been successfully matched as soon as one of the options has. In this example, there are no other tokens in the regex outside the alternation, so the entire regex has successfully matched „Set” in “SetValue”.

Contrary to what we intended, the regex did not match the entire string. There are several solutions. One option is to take into account that the regex engine is eager, and change the order of the options. If we use «GetValue|Get|SetValue|Set», «SetValue» will be attempted before «Set», and the engine will match the entire string. We could also combine the four options into two and use the question mark to make part of them optional: «Get(Value)?|Set(Value)?». Because the question mark is greedy, «SetValue» will be attempted before «Set».

The best option is probably to express the fact that we only want to match complete words. We do not want to match Set or SetValue if the string is “SetValueFunction”. So the solution is

«\b(Get|GetValue|Set|SetValue)\b» or «\b(Get(Value)?|Set(Value))?\b». Since all options have the same end, we can optimize this further to «\b(Get|Set)(Value)?\b» .

All regex flavors discussed in this book work this way, except one: the POSIX standard mandates that the longest match be returned, regardless if the regex engine is implemented using an NFA or DFA algorithm.

10. Optional Items

The question mark makes the preceding token in the regular expression optional. E.g.: «colou?r» matches both „colour” and „color”.

You can make several tokens optional by grouping them together using round brackets, and placing the question mark after the closing bracket. E.g.: «Nov(ember)?» will match „Nov” and „November”.

You can write a regular expression that matches many alternatives by including more than one question mark. «Feb(ruary)? 23(rd)?» matches „February 23rd”, „February 23”, „Feb 23rd” and „Feb 23”.

Important Regex Concept: Greediness

With the question mark, I have introduced the first metacharacter that is *greedy*. The question mark gives the regex engine two choices: try to match the part the question mark applies to, or do not try to match it. The engine will always try to match that part. Only if this causes the entire regular expression to fail, will the engine try ignoring the part the question mark applies to.

The effect is that if you apply the regex «Feb 23(rd)?» to the string “Today is Feb 23rd, 2003”, the match will always be „Feb 23rd” and not „Feb 23”. You can make the question mark *lazy* (i.e. turn off the greediness) by putting a second question mark after the first.

I will say a lot more about greediness when discussing the other repetition operators.

Looking Inside The Regex Engine

Let’s apply the regular expression «colou?r» to the string “The colonel likes the color green”.

The first token in the regex is the literal «c». The first position where it matches successfully is the „c” in “colone1”. The engine continues, and finds that «o» matches „o”, «l» matches „l” and another «o» matches „o”. Then the engine checks whether «u» matches “n”. This fails. However, the question mark tells the regex engine that failing to match «u» is acceptable. Therefore, the engine will skip ahead to the next regex token: «r». But this fails to match “n” as well. Now, the engine can only conclude that the entire regular expression cannot be matched starting at the „c” in “colone1”. Therefore, the engine starts again trying to match «c» to the first o in “colone1”.

After a series of failures, «c» will match with the „c” in “color”, and «o», «l» and «o» match the following characters. Now the engine checks whether «u» matches “r”. This fails. Again: no problem. The question mark allows the engine to continue with «r». This matches „r” and the engine reports that the regex successfully matched „color” in our string.

11. Repetition with Star and Plus

I already introduced one repetition operator or quantifier: the question mark. It tells the engine to attempt to match the preceding token zero times or once, in effect making it optional.

The asterisk or star tells the engine to attempt to match the preceding token zero or more times. The plus tells the engine to attempt to match the preceding token once or more. `<<[A-Za-z][A-Za-z0-9]*>>` matches an HTML tag without any attributes. The sharp brackets are literals. The first character class matches a letter. The second character class matches a letter or digit. The star repeats the second character class. Because we used the star, it's OK if the second character class matches nothing. So our regex will match a tag like ``. When matching `<HTML>`, the first character class will match `H`. The star will cause the second character class to be repeated three times, matching `T`, `M` and `L` with each step.

I could also have used `<<[A-Za-z0-9]+>>`. I did not, because this regex would match `<1>`, which is not a valid HTML tag. But this regex may be sufficient if you know the string you are searching through does not contain any such invalid tags.

Limiting Repetition

Modern regex flavors, like those discussed in this tutorial, have an additional repetition operator that allows you to specify how many times a token can be repeated. The syntax is `{min,max}`, where *min* is a positive integer number indicating the minimum number of matches, and *max* is an integer equal to or greater than *min* indicating the maximum number of matches. If the comma is present but *max* is omitted, the maximum number of matches is infinite. So `{0,}` is the same as `*`, and `{1,}` is the same as `+`. Omitting both the comma and *max* tells the engine to repeat the token exactly *min* times.

You could use `<<\b[1-9][0-9]{3}\b>>` to match a number between 1000 and 9999. `<<\b[1-9][0-9]{2,4}\b>>` matches a number between 100 and 99999. Notice the use of the word boundaries.

Watch Out for The Greediness!

Suppose you want to use a regex to match an HTML tag. You know that the input will be a valid HTML file, so the regular expression does not need to exclude any invalid use of sharp brackets. If it sits between sharp brackets, it is an HTML tag.

Most people new to regular expressions will attempt to use `<<.+>>`. They will be surprised when they test it on a string like `"This is a first test"`. You might expect the regex to match `` and when continuing after that match, ``.

But it does not. The regex will match `first`. Obviously not what we wanted. The reason is that the plus is *greedy*. That is, the plus causes the regex engine to repeat the preceding token as often as possible. Only if that causes the entire regex to fail, will the regex engine *backtrack*. That is, it will go back to the plus, make it give up the last iteration, and proceed with the remainder of the regex. Let's take a look inside the regex engine to see in detail how this works and why this causes our regex to fail. After that, I will present you with two possible solutions.

Like the plus, the star and the repetition using curly braces are greedy.

Looking Inside The Regex Engine

The first token in the regex is «<». This is a literal. As we already know, the first place where it will match is the first „<” in the string. The next token is the dot, which matches any character except newlines. The dot is repeated by the plus. The plus is *greedy*. Therefore, the engine will repeat the dot as many times as it can. The dot matches „E”, so the regex continues to try to match the dot with the next character. „M” is matched, and the dot is repeated once more. The next character is the “>”. You should see the problem by now. The dot matches the „>”, and the engine continues repeating the dot. The dot will match all remaining characters in the string. The dot fails when the engine has reached the void after the end of the string. Only at this point does the regex engine continue with the next token: «>».

So far, «<. +» has matched „first test” and the engine has arrived at the end of the string. «>» cannot match here. The engine remembers that the plus has repeated the dot more often than is required. (Remember that the plus *requires* the dot to match only once.) Rather than admitting failure, the engine will *backtrack*. It will reduce the repetition of the plus by one, and then continue trying the remainder of the regex.

So the match of «<. +» is reduced to „first tes”. The next token in the regex is still «>». But now the next character in the string is the last “t”. Again, these cannot match, causing the engine to backtrack further. The total match so far is reduced to „first te”. But «>» still cannot match. So the engine continues backtracking until the match of «<. +» is reduced to „first”. Now, «>» can match the next character in the string. The last token in the regex has been matched. The engine reports that „first” has been successfully matched.

Remember that the regex engine is *eager* to return a match. It will not continue backtracking further to see if there is another possible match. It will report the first valid match it finds. Because of greediness, this is the leftmost longest match.

Laziness Instead of Greediness

The quick fix to this problem is to make the plus *lazy* instead of greedy. Lazy quantifiers are sometimes also called “ungreedy” or “reluctant”. You can do that by putting a question mark behind the plus in the regex. You can do the same with the star, the curly braces and the question mark itself. So our example becomes «<. +?>». Let’s have another look inside the regex engine.

Again, «<» matches the first „<” in the string. The next token is the dot, this time repeated by a lazy plus. This tells the regex engine to repeat the dot as few times as possible. The minimum is one. So the engine matches the dot with „E”. The requirement has been met, and the engine continues with «>» and “M”. This fails. Again, the engine will *backtrack*. But this time, the backtracking will force the lazy plus to expand rather than reduce its reach. So the match of «<. +» is expanded to „”, and the engine tries again to continue with «>». Now, „>” is matched successfully. The last token in the regex has been matched. The engine reports that „” has been successfully matched. That’s more like it.

An Alternative to Laziness

In this case, there is a better option than making the plus lazy. We can use a greedy plus and a negated character class: «<[^>]+>». The reason why this is better is because of the backtracking. When using the lazy plus, the engine has to backtrack for each character in the HTML tag that it is trying to match. When using

the negated character class, no backtracking occurs at all when the string contains valid HTML code. Backtracking slows down the regex engine. You will not notice the difference when doing a single search in a text editor. But you will save plenty of CPU cycles when using such a regex repeatedly in a tight loop in a script that you are writing, or perhaps in a custom syntax coloring scheme for EditPad Pro.

Finally, remember that this tutorial only talks about regex-directed engines. Text-directed engines do not backtrack. They do not get the speed penalty, but they also do not support lazy repetition operators.

Repeating `\Q...\E` Escape Sequences

The `\Q...\E` sequence escapes a string of characters, matching them as literal characters. The escaped characters are treated as individual characters. If you place a quantifier after the `\E`, it will only be applied to the last character. E.g. if you apply `«\Q*\d+\E+»` to `“*\d+**\d+*”`, the match will be `„*\d+**”`. Only the asterisk is repeated. Java 4 and 5 have a bug that causes the whole `\Q...\E` sequence to be repeated, yielding the whole subject string as the match. This was fixed in Java 6.

12. Use Round Brackets for Grouping

By placing part of a regular expression inside round brackets or parentheses, you can group that part of the regular expression together. This allows you to apply a regex operator, e.g. a repetition operator, to the entire group. I have already used round brackets for this purpose in previous topics throughout this tutorial.

Note that only round brackets can be used for grouping. Square brackets define a character class, and curly braces are used by a special repetition operator.

Round Brackets Create a Backreference

Besides grouping part of a regular expression together, round brackets also create a “backreference”. A backreference stores the part of the string matched by the part of the regular expression inside the parentheses.

That is, unless you use non-capturing parentheses. Remembering part of the regex match in a backreference, slows down the regex engine because it has more work to do. If you do not use the backreference, you can speed things up by using non-capturing parentheses, at the expense of making your regular expression slightly harder to read.

The regex `«Set(Value)?»` matches „Set” or „SetValue”. In the first case, the first backreference will be empty, because it did not match anything. In the second case, the first backreference will contain „Value”.

If you do not use the backreference, you can optimize this regular expression into `«Set(?:Value)?»`. The question mark and the colon after the opening round bracket are the special syntax that you can use to tell the regex engine that this pair of brackets should not create a backreference. Note the question mark after the opening bracket is unrelated to the question mark at the end of the regex. That question mark is the regex operator that makes the previous token optional. This operator cannot appear after an opening round bracket, because an opening bracket by itself is not a valid regex token. Therefore, there is no confusion between the question mark as an operator to make a token optional, and the question mark as a character to change the properties of a pair of round brackets. The colon indicates that the change we want to make is to turn off capturing the backreference.

How to Use Backreferences

Backreferences allow you to reuse part of the regex match. You can reuse it inside the regular expression (see below), or afterwards. What you can do with it afterwards, depends on the tool or programming language you are using. The most common usage is in search-and-replace operations. The replacement text will use a special syntax to allow text matched by capturing groups to be reinserted. This syntax differs greatly between various tools and languages, far more than the regex syntax does. Please check the replacement text reference for details.

Using Backreferences in The Regular Expression

Backreferences can not only be used after a match has been found, but also during the match. Suppose you want to match a pair of opening and closing HTML tags, and the text in between. By putting the opening tag into a backreference, we can reuse the name of the tag for the closing tag. Here's how: «<([A-Z][A-Z0-9]*)\b[^\>]*>.*?</\1>». This regex contains only one pair of parentheses, which capture the string matched by «[A-Z][A-Z0-9]*» into the first backreference. This backreference is reused with «\1» (backslash one). The «/» before it is simply the forward slash in the closing HTML tag that we are trying to match.

To figure out the number of a particular backreference, scan the regular expression from left to right and count the opening round brackets. The first bracket starts backreference number one, the second number two, etc. Non-capturing parentheses are not counted. This fact means that non-capturing parentheses have another benefit: you can insert them into a regular expression without changing the numbers assigned to the backreferences. This can be very useful when modifying a complex regular expression.

You can reuse the same backreference more than once. «([a-c])x\1x\1» will match „axaxa”, „bxbxb” and „cxcxc”.

Looking Inside The Regex Engine

Let's see how the regex engine applies the above regex to the string “Testing <I>bold italic</I> text”. The first token in the regex is the literal «<». The regex engine will traverse the string until it can match at the first „<” in the string. The next token is «[A-Z]». The regex engine also takes note that it is now inside the first pair of capturing parentheses. «[A-Z]» matches „B”. The engine advances to «[A-Z0-9]» and “>”. This match fails. However, because of the star, that's perfectly fine. The position in the string remains at “>”. The position in the regex is advanced to «[^\>]».

This step crosses the closing bracket of the first pair of capturing parentheses. This prompts the regex engine to store what was matched inside them into the first backreference. In this case, „B” is stored.

After storing the backreference, the engine proceeds with the match attempt. «[^\>]» does not match „>”. Again, because of another star, this is not a problem. The position in the string remains at “>”, and position in the regex is advanced to «>». These obviously match. The next token is a dot, repeated by a lazy star. Because of the laziness, the regex engine will initially skip this token, taking note that it should backtrack in case the remainder of the regex fails.

The engine has now arrived at the second «<» in the regex, and the second “<” in the string. These match. The next token is «/». This does not match “I”, and the engine is forced to backtrack to the dot. The dot matches the second „<” in the string. The star is still lazy, so the engine again takes note of the available backtracking position and advances to «<» and “I”. These do not match, so the engine again backtracks.

The backtracking continues until the dot has consumed „<I>bold italic”. At this point, «<» matches the third „<” in the string, and the next token is «/» which matches “/”. The next token is «\1». Note that the token is the backreference, and not «B». The engine does not substitute the backreference in the regular expression. Every time the engine arrives at the backreference, it will read the value that was stored. This means that if the engine had backtracked beyond the first pair of capturing parentheses before arriving the second time at «\1», the new value stored in the first backreference would be used. But this did not happen

here, so „B” it is. This fails to match at “I”, so the engine backtracks again, and the dot consumes the third “<” in the string.

Backtracking continues again until the dot has consumed „<I>bold italic</I>”. At this point, «<» matches „<” and «/» matches „/”. The engine arrives again at «\1». The backreference still holds „B”. «B» matches „B”. The last token in the regex, «>» matches „>”. A complete match has been found: „<I>bold italic</I>”.

Backtracking Into Capturing Groups

You may have wondered about the word boundary «\b» in the «<([A-Z][A-Z0-9]*)\b[^>]*>.*?</\1>» mentioned above. This is to make sure the regex won’t match incorrectly paired tags such as “<boo>bold”. You may think that cannot happen because the capturing group matches „boo” which causes «\1» to try to match the same, and fail. That is indeed what happens. But then the regex engine backtracks.

Let’s take the regex «<([A-Z][A-Z0-9]*)\b[^>]*>.*?</\1>» without the word boundary and look inside the regex engine at the point where «\1» fails the first time. First, «.*?» continues to expand until it has reached the end of the string, and «</\1>» has failed to match each time «.*?» matched one more character.

Then the regex engine backtracks into the capturing group. «[A-Z0-9]*» has matched „oo”, but would just as happily match „o” or nothing at all. When backtracking, «[A-Z0-9]*» is forced to give up one character. The regex engine continues, exiting the capturing group a second time. Since [A-Z][A-Z0-9]* has now matched „bo”, that is what is stored into the capturing group, overwriting „boo” that was stored before. «[^>]*» matches the second „o” in the opening tag. «>.*?</» matches „>bold<”. «\1» fails again.

The regex engine does all the same backtracking once more, until «[A-Z0-9]*» is forced to give up another character, causing it to match nothing, which the star allows. The capturing group now stores just „b”. «[^>]*» now matches „oo”. «>.*?</» once again matches „>bold<”. «\1» now succeeds, as does «>» and an overall match is found. But not the one we wanted.

There are several solutions to this. One is to use the word boundary. When «[A-Z0-9]*» backtracks the first time, reducing the capturing group to „bo”, «\b» fails to match between “o” and “o”. This forces «[A-Z0-9]*» to backtrack again immediately. The capturing group is reduced to „b” and the word boundary fails between “b” and “o”. There are no further backtracking positions, so the whole match attempt fails.

The reason we need the word boundary is that we’re using «[^>]*» to skip over any attributes in the tag. If your paired tags never have any attributes, you can leave that out, and use «<([A-Z][A-Z0-9]*)>.*?</\1>». Each time «[A-Z0-9]*» backtracks, the «>» that follows it will fail to match, quickly ending the match attempt.

If you didn’t expect the regex engine to backtrack into capturing groups, you can use an atomic group. The regex engine always backtracks into capturing groups, and never captures atomic groups. You can put the capturing group inside an atomic group to get an atomic capturing group: «(>(atomic capture))». In this case, we can put the whole opening tag into the atomic group: «(><([A-Z][A-Z0-9]*)\b[^>]*>).*?</\1>». The tutorial section on atomic grouping has all the details.

Backreferences to Failed Groups

The previous section applies to all regex flavors, except those few that don't support capturing groups at all. Flavors behave differently when you start doing things that don't fit the "match the text matched by a previous capturing group" job description.

There is a difference between a backreference to a capturing group that matched nothing, and one to a capturing group that did not participate in the match at all. The regex `«(q?)b\1»` will match „b”. `«q?»` is optional and matches nothing, causing `«(q?)»` to successfully match and capture nothing. `«b»` matches „b” and `«\1»` successfully matches the nothing captured by the group.

The regex `«(q)?b\1»` however will fail to match “b”. `«(q)»` fails to match at all, so the group never gets to capture anything at all. Because the whole group is optional, the engine does proceed to match `«b»`. However, the engine now arrives at `«\1»` which references a group that did not participate in the match attempt at all. This causes the backreference to fail to match at all, mimicking the result of the group. Since there's no `«?»` making `«\1»` optional, the overall match attempt fails.

The only exception is JavaScript. According to the official ECMA standard, a backreference to a non-participating capturing group must successfully match nothing just like a backreference to a participating group that captured nothing does. In other words, in JavaScript, `«(q?)b\1»` and `«(q)?b\1»` both match „b”.

Forward References and Invalid References

Modern flavors, notably JGsoft, .NET, Java, Perl, PCRE and Ruby allow forward references. That is: you can use a backreference to a group that appears later in the regex. Forward references are obviously only useful if they're inside a repeated group. Then there can be situations in which the regex engine evaluates the backreference after the group has already matched. Before the group is attempted, the backreference will fail like a backreference to a failed group does.

If forward references are supported, the regex `«(\2two|(one))+»` will match „oneonetwo”. At the start of the string, `«\2»` fails. Trying the other alternative, „one” is matched by the second capturing group, and subsequently by the first group. The first group is then repeated. This time, `«\2»` matches „one” as captured by the second group. `«two»` then matches „two”. With two repetitions of the first group, the regex has matched the whole subject string.

A nested reference is a backreference inside the capturing group that it references, e.g. `«(\1two|(one))+»`. This regex will give exactly the same behavior with flavors that support forward references. Some flavors that don't support forward references do support nested references. This includes JavaScript.

With all other flavors, using a backreference before its group in the regular expression is the same as using a backreference to a group that doesn't exist at all. All flavors discussed in this tutorial, except JavaScript and Ruby, treat backreferences to undefined groups as an error. In JavaScript and Ruby, they always result in a zero-width match. For Ruby this is a potential pitfall. In Ruby, `«(a)(b)?\2»` will fail to match “a”, because `«\2»` references a non-participating group. But `«(a)(b)?\7»` will match „a”. For JavaScript this is logical, as backreferences to non-participating groups do the same. Both regexes will match „a”.

Repetition and Backreferences

As I mentioned in the above inside look, the regex engine does not permanently substitute backreferences in the regular expression. It will use the last match saved into the backreference each time it needs to be used. If a new match is found by capturing parentheses, the previously saved match is overwritten. There is a clear difference between `«([abc]+)»` and `«([abc])+»`. Though both successfully match „cab”, the first regex will put „cab” into the first backreference, while the second regex will only store „b”. That is because in the second regex, the plus caused the pair of parentheses to repeat three times. The first time, „c” was stored. The second time „a” and the third time „b”. Each time, the previous value was overwritten, so „b” remains.

This also means that `«([abc]+)=\1»` will match „cab=cab”, and that `«([abc])+=\1»` will not. The reason is that when the engine arrives at `«\1»`, it holds «b» which fails to match “c”. Obvious when you look at a simple example like this one, but a common cause of difficulty with regular expressions nonetheless. When using backreferences, always double check that you are really capturing what you want.

Useful Example: Checking for Doubled Words

When editing text, doubled words such as “the the” easily creep in. Using the regex `«\b(\w+)\s+\1\b»` in your text editor, you can easily find them. To delete the second word, simply type in “\1” as the replacement text and click the Replace button.

Parentheses and Backreferences Cannot Be Used Inside Character Classes

Round brackets cannot be used inside character classes, at least not as metacharacters. When you put a round bracket in a character class, it is treated as a literal character. So the regex `«[(a)b]»` matches „a”, „b”, „(” and „)”.

Backreferences also cannot be used inside a character class. The `\1` in regex like `«(a)[\1b]»` will be interpreted as an octal escape in most regex flavors. So this regex will match an „a” followed by either `«\x01»` or a «b».

13. Named Capturing Groups

All modern regular expression engines support capturing groups, which are numbered from left to right, starting with one. The numbers can then be used in backreferences to match the same text again in the regular expression, or to use part of the regex match for further processing. In a complex regular expression with many capturing groups, the numbering can get a little confusing.

Named Capture with Python, PCRE and PHP

Python's `regex` module was the first to offer a solution: named capture. By assigning a name to a capturing group, you can easily reference it by name. `«(?P<name>group)»` captures the match of `«group»` into the backreference "name". You can reference the contents of the group with the numbered backreference `«\1»` or the named backreference `«(?P=name)»`.

The open source PCRE library has followed Python's example, and offers named capture using the same syntax. The PHP `preg` functions offer the same functionality, since they are based on PCRE.

Python's `sub()` function allows you to reference a named group as `«\1»` or `«\g<name>»`. This does *not* work in PHP. In PHP, you can use double-quoted string interpolation with the `$regs` parameter you passed to `pcre_match()`: `«$regs['name']»`.

Named Capture with .NET's System.Text.RegularExpressions

The regular expression classes of the .NET framework also support named capture. Unfortunately, the Microsoft developers decided to invent their own syntax, rather than follow the one pioneered by Python. Currently, no other regex flavor supports Microsoft's version of named capture.

Here is an example with two capturing groups in .NET style: `«(?<first>group) (? 'second' group)»`. As you can see, .NET offers two syntaxes to create a capturing group: one using sharp brackets, and the other using single quotes. The first syntax is preferable in strings, where single quotes may need to be escaped. The second syntax is preferable in ASP code, where the sharp brackets are used for HTML tags. You can use the pointy bracket flavor and the quoted flavors interchangeably.

To reference a capturing group inside the regex, use `«\k<name>»` or `«\k 'name' »`. Again, you can use the two syntactic variations interchangeably.

When doing a search-and-replace, you can reference the named group with the familiar dollar sign syntax: `«${name}»`. Simply use a name instead of a number between the curly braces.

Multiple Groups with The Same Name

The .NET framework allows multiple groups in the regular expression to have the same name. If you do so, both groups will store their matches in the same `Group` object. You won't be able to distinguish which group captured the text. This can be useful in regular expressions with multiple alternatives to match the same thing. E.g. if you want to match "a" followed by a digit 0..5, or "b" followed by a digit 4..7, and you only care about

the digit, you could use the regex `«a(?'digit'[0-5])|b(?'digit'[4-7])»`. The group named “digit” will then give you the digit 0..7 that was matched, regardless of the letter.

Python and PCRE do not allow multiple groups to use the same name. Doing so will give a regex compilation error.

Names and Numbers for Capturing Groups

Here is where things get a bit ugly. Python and PCRE treat named capturing groups just like unnamed capturing groups, and number both kinds from left to right, starting with one. The regex `«(a)(?P<x>b)(c)(?P<y>d)»` matches „abcd” as expected. If you do a search-and-replace with this regex and the replacement `“\1\2\3\4”`, you will get “abcd”. All four groups were numbered from left to right, from one till four. Easy and logical.

Things are quite a bit more complicated with the .NET framework. The regex `«(a)(?<x>b)(c)(?<y>d)»` again matches „abcd”. However, if you do a search-and-replace with `“$1$2$3$4”` as the replacement, you will get “acbd”. Probably not what you expected.

The .NET framework *does* number named capturing groups from left to right, but numbers them *after* all the unnamed groups have been numbered. So the unnamed groups `«(a)»` and `«(c)»` get numbered first, from left to right, starting at one. Then the named groups `«(?<x>b)»` and `«(?<y>d)»` get their numbers, continuing from the unnamed groups, in this case: three.

To make things simple, when using .NET’s regex support, just assume that named groups do not get numbered at all, and reference them by name exclusively. To keep things compatible across regex flavors, I strongly recommend that you do not mix named and unnamed capturing groups at all. Either give a group a name, or make it non-capturing as in `«(?:nocapture)»`. Non-capturing groups are more efficient, since the regex engine does not need to keep track of their matches.

Best of Both Worlds

The JGsoft regex engine supports both .NET-style and Python-style named capture. Python-style named groups are numbered along unnamed ones, like Python does. .NET-style named groups are numbered afterwards, like .NET does. You can mix both styles in the same regex. The JGsoft engine allows multiple groups to use the same name, regardless of the syntax used.

In PowerGREP, named capturing groups play a special roles. Groups with the same name are shared between all regular expressions and replacement texts in the same PowerGREP action. This allows captured by a named capturing group in one part of the action to be referenced in a later part of the action. Because of this, PowerGREP does not allow numbered references to named capturing groups at all. When mixing named and numbered groups in a regex, the numbered groups are still numbered following the Python and .NET rules, like the JGsoft flavor always does.

14. Unicode Regular Expressions

Unicode is a character set that aims to define all characters and glyphs from all human languages, living and dead. With more and more software being required to support multiple languages, or even just *any* language, Unicode has been strongly gaining popularity in recent years. Using different character sets for different languages is simply too cumbersome for programmers and users.

Unfortunately, Unicode brings its own requirements and pitfalls when it comes to regular expressions. Of the regex flavors discussed in this tutorial, Java, XML and the .NET framework use Unicode-based regex engines. Perl supports Unicode starting with version 5.6. PCRE can optionally be compiled with Unicode support. Note that PCRE is far less flexible in what it allows for the `\p` tokens, despite its name “Perl-compatible”. The PHP `preg` functions, which are based on PCRE, support Unicode when the `/u` option is appended to the regular expression.

RegexBuddy’s regex engine is fully Unicode-based starting with version 2.0.0. RegexBuddy 1.x.x did not support Unicode at all. PowerGREP uses the same Unicode regex engine starting with version 3.0.0. Earlier versions would convert Unicode files to ANSI prior to grepping with an 8-bit (i.e. non-Unicode) regex engine. EditPad Pro supports Unicode starting with version 6.0.0.

Characters, Code Points and Graphemes or How Unicode Makes a Mess of Things

Most people would consider “à” a single character. Unfortunately, it need not be depending on the meaning of the word “character”.

All Unicode regex engines discussed in this tutorial treat any single Unicode *code point* as a single character. When this tutorial tells you that the dot matches any single character, this translates into Unicode parlance as “the dot matches any single Unicode code point”. In Unicode, “à” can be encoded as two code points: U+0061 (a) followed by U+0300 (grave accent). In this situation, `«.»` applied to “à” will match „a” without the accent. `«^.»` will fail to match, since the string consists of two code points. `«^.»` matches „à”.

The Unicode code point U+0300 (grave accent) is a *combining mark*. Any code point that is not a combining mark can be followed by any number of combining marks. This sequence, like U+0061 U+0300 above, is displayed as a single *grapheme* on the screen.

Unfortunately, “à” can also be encoded with the single Unicode code point U+00E0 (a with grave accent). The reason for this duality is that many historical character sets encode “a with grave accent” as a single character. Unicode’s designers thought it would be useful to have a one-on-one mapping with popular legacy character sets, in addition to the Unicode way of separating marks and base letters (which makes arbitrary combinations not supported by legacy character sets possible).

How to Match a Single Unicode Grapheme

Matching a single grapheme, whether it’s encoded as a single code point, or as multiple code points using combining marks, is easy in Perl, RegexBuddy and PowerGREP: simply use `«\X»`. You can consider `«\X»` the Unicode version of the dot in regex engines that use plain ASCII. There is one difference, though: `«\X»`

always matches line break characters, whereas the dot does not match line break characters unless you enable the dot matches newline matching mode.

Java and .NET unfortunately do not support `\X` (yet). Use `\P{M}\p{M}*` or `(?>\P{M}\p{M}*)` as a reasonably close substitute. To match any number of graphemes, use `(?>\P{M}\p{M}*)+` as a substitute for `\X+`.

Matching a Specific Code Point

To match a specific Unicode code point, use `\uFFFF` where FFFF is the hexadecimal number of the code point you want to match. You must always specify 4 hexadecimal digits. E.g. `\u00E0` matches „à”, but only when encoded as a single code point U+00E0.

Perl and PCRE do not support the `\uFFFF` syntax. They use `\x{FFFF}` instead. You can omit leading zeros in the hexadecimal number between the curly braces. Since `\x` by itself is not a valid regex token, `\x{1234}` can never be confused to match `\x` 1234 times. It always matches the Unicode code point U+1234. `\x{1234}{5678}` will try to match code point U+1234 exactly 5678 times.

In Java, the regex token `\uFFFF` only matches the specified code point, even when you turned on canonical equivalence. However, the same syntax `\uFFFF` is also used to insert Unicode characters into literal strings in the Java source code. `Pattern.compile("\u00E0")` will match both the single-code-point and double-code-point encodings of „à”, while `Pattern.compile("\u00E0")` matches only the single-code-point version. Remember that when writing a regex as a Java string literal, backslashes must be escaped. The former Java code compiles the regex `à`, while the latter compiles `\u00E0`. Depending on what you’re doing, the difference may be significant.

JavaScript, which does not offer any Unicode support through its RegExp class, does support `\uFFFF` for matching a single Unicode code point as part of its string syntax.

XML Schema does not have a regex token for matching Unicode code points. However, you can easily use XML entities like `&#xXXXX`; to insert literal code points into your regular expression.

Unicode Character Properties

In addition to complications, Unicode also brings new possibilities. One is that each Unicode character belongs to a certain category. You can match a single character belonging to a particular category with `\p{}`. You can match a single character *not* belonging to a particular category with `\P{}`.

Again, “character” really means “Unicode code point”. `\p{L}` matches a single code point in the category “letter”. If your input string is “à” encoded as U+0061 U+0300, it matches „a” without the accent. If the input is “à” encoded as U+00E0, it matches „à” with the accent. The reason is that both the code points U+0061 (a) and U+00E0 (à) are in the category “letter”, while U+0300 is in the category “mark”.

You should now understand why `\P{M}\p{M}*` is the equivalent of `\X`. `\P{M}` matches a code point that is not a combining mark, while `\p{M}*` matches zero or more code points that are combining marks. To match a letter including any diacritics, use `\p{L}\p{M}*`. This last regex will always match „à”, regardless of how it is encoded.

The .NET Regex class and PCRE are case sensitive when it checks the part between curly braces of a `\p` token. `«\p{Zs}»` will match any kind of space character, while `«\p{zS}»` will throw an error. All other regex engines described in this tutorial will match the space in both cases, ignoring the case of the property between the curly braces. Still, I recommend you make a habit of using the same uppercase and lowercase combination as I did in the list of properties below. This will make your regular expressions work with all Unicode regex engines.

In addition to the standard notation, `«\p{L}»`, Java, Perl, PCRE and the JGsoft engine allow you to use the shorthand `«\pL»`. The shorthand only works with single-letter Unicode properties. `«\pLl»` is *not* the equivalent of `«\p{Ll}»`. It is the equivalent of `«\p{L}l»` which matches „A1” or „à1” or any Unicode letter followed by a literal „1”.

Perl and the JGsoft engine also support the longhand `«\p{Letter}»`. You can find a complete list of all Unicode properties below. You may omit the underscores or use hyphens or spaces instead.

- `«\p{L}»` or `«\p{Letter}»`: any kind of letter from any language.
 - `«\p{Ll}»` or `«\p{Lowercase_Letter}»`: a lowercase letter that has an uppercase variant.
 - `«\p{Lu}»` or `«\p{Uppercase_Letter}»`: an uppercase letter that has a lowercase variant.
 - `«\p{Lt}»` or `«\p{Titlecase_Letter}»`: a letter that appears at the start of a word when only the first letter of the word is capitalized.
 - `«\p{L&}»` or `«\p{Letter&}»`: a letter that exists in lowercase and uppercase variants (combination of Ll, Lu and Lt).
 - `«\p{Lm}»` or `«\p{Modifier_Letter}»`: a special character that is used like a letter.
 - `«\p{Lo}»` or `«\p{Other_Letter}»`: a letter or ideograph that does not have lowercase and uppercase variants.
- `«\p{M}»` or `«\p{Mark}»`: a character intended to be combined with another character (e.g. accents, umlauts, enclosing boxes, etc.).
 - `«\p{Mn}»` or `«\p{Non_Spacing_Mark}»`: a character intended to be combined with another character without taking up extra space (e.g. accents, umlauts, etc.).
 - `«\p{Mc}»` or `«\p{Spacing_Combining_Mark}»`: a character intended to be combined with another character that takes up extra space (vowel signs in many Eastern languages).
 - `«\p{Me}»` or `«\p{Enclosing_Mark}»`: a character that encloses the character is is combined with (circle, square, keycap, etc.).
- `«\p{Z}»` or `«\p{Separator}»`: any kind of whitespace or invisible separator.
 - `«\p{Zs}»` or `«\p{Space_Separator}»`: a whitespace character that is invisible, but does take up space.
 - `«\p{Zl}»` or `«\p{Line_Separator}»`: line separator character U+2028.
 - `«\p{Zp}»` or `«\p{Paragraph_Separator}»`: paragraph separator character U+2029.
- `«\p{S}»` or `«\p{Symbol}»`: math symbols, currency signs, dingbats, box-drawing characters, etc..
 - `«\p{Sm}»` or `«\p{Math_Symbol}»`: any mathematical symbol.
 - `«\p{Sc}»` or `«\p{Currency_Symbol}»`: any currency sign.
 - `«\p{Sk}»` or `«\p{Modifier_Symbol}»`: a combining character (mark) as a full character on its own.
 - `«\p{So}»` or `«\p{Other_Symbol}»`: various symbols that are not math symbols, currency signs, or combining characters.
- `«\p{N}»` or `«\p{Number}»`: any kind of numeric character in any script.
 - `«\p{Nd}»` or `«\p{Decimal_Digit_Number}»`: a digit zero through nine in any script except ideographic scripts.
 - `«\p{Nl}»` or `«\p{Letter_Number}»`: a number that looks like a letter, such as a Roman numeral.

- `«\p{No}»` or `«\p{Other_Number}»`: a superscript or subscript digit, or a number that is not a digit 0..9 (excluding numbers from ideographic scripts).
- `«\p{P}»` or `«\p{Punctuation}»`: any kind of punctuation character.
 - `«\p{Pd}»` or `«\p{Dash_Punctuation}»`: any kind of hyphen or dash.
 - `«\p{Ps}»` or `«\p{Open_Punctuation}»`: any kind of opening bracket.
 - `«\p{Pe}»` or `«\p{Close_Punctuation}»`: any kind of closing bracket.
 - `«\p{Pi}»` or `«\p{Initial_Punctuation}»`: any kind of opening quote.
 - `«\p{Pf}»` or `«\p{Final_Punctuation}»`: any kind of closing quote.
 - `«\p{Pc}»` or `«\p{Connector_Punctuation}»`: a punctuation character such as an underscore that connects words.
 - `«\p{Po}»` or `«\p{Other_Punctuation}»`: any kind of punctuation character that is not a dash, bracket, quote or connector.
- `«\p{C}»` or `«\p{Other}»`: invisible control characters and unused code points.
 - `«\p{Cc}»` or `«\p{Control}»`: an ASCII 0x00..0x1F or Latin-1 0x80..0x9F control character.
 - `«\p{Cf}»` or `«\p{Format}»`: invisible formatting indicator.
 - `«\p{Co}»` or `«\p{Private_Use}»`: any code point reserved for private use.
 - `«\p{Cs}»` or `«\p{Surrogate}»`: one half of a surrogate pair in UTF-16 encoding.
 - `«\p{Cn}»` or `«\p{Unassigned}»`: any code point to which no character has been assigned.

Unicode Scripts

The Unicode standard places each assigned code point (character) into one script. A script is a group of code points used by a particular human writing system. Some scripts like Thai correspond with a single human language. Other scripts like Latin span multiple languages.

Some languages are composed of multiple scripts. There is no Japanese Unicode script. Instead, Unicode offers the Hiragana, Katakana, Han and Latin scripts that Japanese documents are usually composed of.

A special script is the Common script. This script contains all sorts of characters that are common to a wide range of scripts. It includes all sorts of punctuation, whitespace and miscellaneous symbols.

All assigned Unicode code points (those matched by `«\P{Cn}»`) are part of exactly one Unicode script. All unassigned Unicode code points (those matched by `«\p{Cn}»`) are not part of any Unicode script at all.

Very few regular expression engines support Unicode scripts today. Of all the flavors discussed in this tutorial, only the JGsoft engine, Perl and PCRE can match Unicode scripts. Here's a complete list of all Unicode scripts:

1. `«\p{Common}»`
2. `«\p{Arabic}»`
3. `«\p{Armenian}»`
4. `«\p{Bengali}»`
5. `«\p{Bopomofo}»`
6. `«\p{Braille}»`
7. `«\p{Buhid}»`
8. `«\p{CanadianAboriginal}»`
9. `«\p{Cherokee}»`
10. `«\p{Cyrillic}»`
11. `«\p{Devanagari}»`

12. `«\p{Ethiopic}»`
13. `«\p{Georgian}»`
14. `«\p{Greek}»`
15. `«\p{Gujarati}»`
16. `«\p{Gurmukhi}»`
17. `«\p{Han}»`
18. `«\p{Hangul}»`
19. `«\p{Hanunoo}»`
20. `«\p{Hebrew}»`
21. `«\p{Hiragana}»`
22. `«\p{Inherited}»`
23. `«\p{Kannada}»`
24. `«\p{Katakana}»`
25. `«\p{Khmer}»`
26. `«\p{Lao}»`
27. `«\p{Latin}»`
28. `«\p{Limbu}»`
29. `«\p{Malayalam}»`
30. `«\p{Mongolian}»`
31. `«\p{Myanmar}»`
32. `«\p{Ogham}»`
33. `«\p{Oriya}»`
34. `«\p{Runic}»`
35. `«\p{Sinhala}»`
36. `«\p{Syriac}»`
37. `«\p{Tagalog}»`
38. `«\p{Tagbanwa}»`
39. `«\p{TaiLe}»`
40. `«\p{Tamil}»`
41. `«\p{Telugu}»`
42. `«\p{Thaana}»`
43. `«\p{Thai}»`
44. `«\p{Tibetan}»`
45. `«\p{Yi}»`

Instead of the `«\p{Latin}»` syntax you can also use `«\p{IsLatin}»`. The “Is” syntax is useful for distinguishing between scripts and blocks, as explained in the next section. Unfortunately, PCRE does not support “Is” as of this writing.

Unicode Blocks

The Unicode standard divides the Unicode character map into different blocks or ranges of code points. Each block is used to define characters of a particular script like “Tibetan” or belonging to a particular group like “Braille Patterns”. Most blocks include unassigned code points, reserved for future expansion of the Unicode standard.

Note that Unicode blocks do not correspond 100% with scripts. An essential difference between blocks and scripts is that a block is a single contiguous range of code points, as listed below. Scripts consist of characters taken from all over the Unicode character map. Blocks may include unassigned code points (i.e. code points

matched by `<\p{Cn}>`). Scripts never include unassigned code points. Generally, if you're not sure whether to use a Unicode script or Unicode block, use the script.

E.g. the Currency block does not include the dollar and yen symbols. Those are found in the Basic_Latin and Latin-1_Supplement blocks instead, for historical reasons, even though both are currency symbols, and the yen symbol is not a Latin character. You should not blindly use any of the blocks listed below based on their names. Instead, look at the ranges of characters they actually match. A tool like RegexBuddy can be very helpful with this. E.g. the Unicode property `<\p{Sc}>` or `<\p{Currency_Symbol}>` would be a better choice than the Unicode block `<\p{InCurrency}>` when trying to find all currency symbols.

1. `<\p{InBasic_Latin}>`: U+0000..U+007F
2. `<\p{InLatin-1_Supplement}>`: U+0080..U+00FF
3. `<\p{InLatin_Extended-A}>`: U+0100..U+017F
4. `<\p{InLatin_Extended-B}>`: U+0180..U+024F
5. `<\p{InIPA_Extensions}>`: U+0250..U+02AF
6. `<\p{InSpacing_Modifier_Letters}>`: U+02B0..U+02FF
7. `<\p{InCombining_Diacritical_Marks}>`: U+0300..U+036F
8. `<\p{InGreek_and_Coptic}>`: U+0370..U+03FF
9. `<\p{InCyrillic}>`: U+0400..U+04FF
10. `<\p{InCyrillic_Supplementary}>`: U+0500..U+052F
11. `<\p{InArmenian}>`: U+0530..U+058F
12. `<\p{InHebrew}>`: U+0590..U+05FF
13. `<\p{InArabic}>`: U+0600..U+06FF
14. `<\p{InSyriac}>`: U+0700..U+074F
15. `<\p{InThaana}>`: U+0780..U+07BF
16. `<\p{InDevanagari}>`: U+0900..U+097F
17. `<\p{InBengali}>`: U+0980..U+09FF
18. `<\p{InGurmukhi}>`: U+0A00..U+0A7F
19. `<\p{InGujarati}>`: U+0A80..U+0AFF
20. `<\p{InOriya}>`: U+0B00..U+0B7F
21. `<\p{InTamil}>`: U+0B80..U+0BFF
22. `<\p{InTelugu}>`: U+0C00..U+0C7F
23. `<\p{InKannada}>`: U+0C80..U+0CFF
24. `<\p{InMalayalam}>`: U+0D00..U+0D7F
25. `<\p{InSinhala}>`: U+0D80..U+0DFF
26. `<\p{InThai}>`: U+0E00..U+0E7F
27. `<\p{InLao}>`: U+0E80..U+0EFF
28. `<\p{InTibetan}>`: U+0F00..U+0FFF
29. `<\p{InMyanmar}>`: U+1000..U+109F
30. `<\p{InGeorgian}>`: U+10A0..U+10FF
31. `<\p{InHangul_Jamo}>`: U+1100..U+11FF
32. `<\p{InEthiopic}>`: U+1200..U+137F
33. `<\p{InCherokee}>`: U+13A0..U+13FF
34. `<\p{InUnified_Canadian_Aboriginal_Syllabics}>`: U+1400..U+167F
35. `<\p{InOgham}>`: U+1680..U+169F
36. `<\p{InRunic}>`: U+16A0..U+16FF
37. `<\p{InTagalog}>`: U+1700..U+171F
38. `<\p{InHanunoo}>`: U+1720..U+173F
39. `<\p{InBuhid}>`: U+1740..U+175F
40. `<\p{InTagbanwa}>`: U+1760..U+177F
41. `<\p{InKhmer}>`: U+1780..U+17FF

42. «\p{InMongolian}»: U+1800..U+18AF
43. «\p{InLimbu}»: U+1900..U+194F
44. «\p{InTai_Le}»: U+1950..U+197F
45. «\p{InKhmer_Symbols}»: U+19E0..U+19FF
46. «\p{InPhonetic_Extensions}»: U+1D00..U+1D7F
47. «\p{InLatin_Extended_Additional}»: U+1E00..U+1EFF
48. «\p{InGreek_Extended}»: U+1F00..U+1FFF
49. «\p{InGeneral_Punctuation}»: U+2000..U+206F
50. «\p{InSuperscripts_and_Subscripts}»: U+2070..U+209F
51. «\p{InCurrency_Symbols}»: U+20A0..U+20CF
52. «\p{InCombining_Diacritical_Marks_for_Symbols}»: U+20D0..U+20FF
53. «\p{InLetterlike_Symbols}»: U+2100..U+214F
54. «\p{InNumber_Forms}»: U+2150..U+218F
55. «\p{InArrows}»: U+2190..U+21FF
56. «\p{InMathematical_Operators}»: U+2200..U+22FF
57. «\p{InMiscellaneous_Technical}»: U+2300..U+23FF
58. «\p{InControl_Pictures}»: U+2400..U+243F
59. «\p{InOptical_Character_Recognition}»: U+2440..U+245F
60. «\p{InEnclosed_Alphanumerics}»: U+2460..U+24FF
61. «\p{InBox_Drawing}»: U+2500..U+257F
62. «\p{InBlock_Elements}»: U+2580..U+259F
63. «\p{InGeometric_Shapes}»: U+25A0..U+25FF
64. «\p{InMiscellaneous_Symbols}»: U+2600..U+26FF
65. «\p{InDingbats}»: U+2700..U+27BF
66. «\p{InMiscellaneous_Mathematical_Symbols-A}»: U+27C0..U+27EF
67. «\p{InSupplemental_Arrows-A}»: U+27F0..U+27FF
68. «\p{InBraille_Patterns}»: U+2800..U+28FF
69. «\p{InSupplemental_Arrows-B}»: U+2900..U+297F
70. «\p{InMiscellaneous_Mathematical_Symbols-B}»: U+2980..U+29FF
71. «\p{InSupplemental_Mathematical_Operators}»: U+2A00..U+2AFF
72. «\p{InMiscellaneous_Symbols_and_Arrows}»: U+2B00..U+2BFF
73. «\p{InCJK_Radicals_Supplement}»: U+2E80..U+2EFF
74. «\p{InKangxi_Radicals}»: U+2F00..U+2FDF
75. «\p{InIdeographic_Description_Characters}»: U+2FF0..U+2FFF
76. «\p{InCJK_Symbols_and_Punctuation}»: U+3000..U+303F
77. «\p{InHiragana}»: U+3040..U+309F
78. «\p{InKatakana}»: U+30A0..U+30FF
79. «\p{InBopomofo}»: U+3100..U+312F
80. «\p{InHangul_Compatibility_Jamo}»: U+3130..U+318F
81. «\p{InKanbun}»: U+3190..U+319F
82. «\p{InBopomofo_Extended}»: U+31A0..U+31BF
83. «\p{InKatakana_Phonetic_Extensions}»: U+31F0..U+31FF
84. «\p{InEnclosed_CJK_Letters_and_Months}»: U+3200..U+32FF
85. «\p{InCJK_Compatibility}»: U+3300..U+33FF
86. «\p{InCJK_Unified_Ideographs_Extension_A}»: U+3400..U+4DBF
87. «\p{InYijing_Hexagram_Symbols}»: U+4DC0..U+4DFF
88. «\p{InCJK_Unified_Ideographs}»: U+4E00..U+9FFF
89. «\p{InYi_Syllables}»: U+A000..U+A48F
90. «\p{InYi_Radicals}»: U+A490..U+A4CF
91. «\p{InHangul_Syllables}»: U+AC00..U+D7AF
92. «\p{InHigh_Surrogates}»: U+D800..U+DB7F
93. «\p{InHigh_Private_Use_Surrogates}»: U+DB80..U+DBFF

- 94. `«\p{InLow_Surrogates}»`: U+DC00..U+DFFF
- 95. `«\p{InPrivate_Use_Area}»`: U+E000..U+F8FF
- 96. `«\p{InCJK_Compatibility_Ideographs}»`: U+F900..U+FAFF
- 97. `«\p{InAlphabetic_Presentation_Forms}»`: U+FB00..U+FB4F
- 98. `«\p{InArabic_Presentation_Forms-A}»`: U+FB50..U+FDFF
- 99. `«\p{InVariation_Selectors}»`: U+FE00..U+FE0F
- 100. `«\p{InCombining_Half_Marks}»`: U+FE20..U+FE2F
- 101. `«\p{InCJK_Compatibility_Forms}»`: U+FE30..U+FE4F
- 102. `«\p{InSmall_Form_Variants}»`: U+FE50..U+FE6F
- 103. `«\p{InArabic_Presentation_Forms-B}»`: U+FE70..U+FEFF
- 104. `«\p{InHalfwidth_and_Fullwidth_Forms}»`: U+FF00..U+FFEF
- 105. `«\p{InSpecials}»`: U+FFF0..U+FFFF

Not all Unicode regex engines use the same syntax to match Unicode blocks. Perl and Java use the `«\p{InBlock}»` syntax as listed above. .NET and XML use `«\p{IsBlock}»` instead. The JGsoft engine supports both notations. I recommend you use the “In” notation if your regex engine supports it. “In” can only be used for Unicode blocks, while “Is” can also be used for Unicode properties and scripts, depending on the regular expression flavor you’re using. By using “In”, it’s obvious you’re matching a block and not a similarly named property or script.

In .NET and XML, you must omit the underscores but keep the hyphens in the block names. E.g. Use `«\p{IsLatinExtended-A}»` instead of `«\p{InLatin_Extended-A}»`. Perl and Java allow you to use an underscore, hyphen, space or nothing for each underscore or hyphen in the block’s name. .NET and XML also compare the names case sensitively, while Perl and Java do not. `«\p{islatinextended-a}»` throws an error in .NET, while `«\p{inlatinextended-a}»` works fine in Perl and Java.

The JGsoft engine supports all of the above notations. You can use “In” or “Is”, ignore differences in upper and lower case, and use spaces, underscores and hyphens as you like. This way you can keep using the syntax of your favorite programming language, and have it work as you’d expect in PowerGREP or EditPad Pro.

The actual names of the blocks are the same in all regular expression engines. The block names are defined in the Unicode standard. PCRE does not support Unicode blocks.

Alternative Unicode Regex Syntax

Unicode is a relatively new addition to the world of regular expressions. As you guessed from my explanations of different notations, different regex engine designers unfortunately have different ideas about the syntax to use. Perl and Java even support a few additional alternative notations that you may encounter in regular expressions created by others. I recommend against using these notations in your own regular expressions, to maintain clarity and compatibility with other regex flavors, and understandability by people more familiar with other flavors.

If you are just getting started with Unicode regular expressions, you may want to skip this section until later, to avoid confusion (if the above didn’t confuse you already).

In Perl and PCRE regular expressions, you may encounter a Unicode property like `«\p{^Lu}»` or `«\p{^Letter}»`. These are negated properties identical to `«\p{Lu}»` or `«\P{Letter}»`. Since very few regex flavors support the `«\p{^L}»` notation, and all Unicode-compatible regex flavors (including Perl and PCRE) support `«\P{L}»`, I strongly recommend you use the latter syntax.

Perl (but not PCRE) and Java support the `«\p{ISL}»` notation, prefixing one-letter and two-letter Unicode property notations with “Is”. Since very few regex flavors support the `«\p{ISL}»` notation, and all Unicode-compatible regex flavors (including Perl and Java) support `«\p{L}»`, I strongly recommend you use the latter syntax.

Perl and Java allow you to omit the “In” when matching Unicode blocks, so you can write `«\p{Arrows}»` instead of `«\p{InArrows}»`. Perl can also match Unicode scripts, and some scripts like “Hebrew” have the same name as a Unicode block. In that situation, Perl will match the Hebrew script instead of the Hebrew block when you write `«\p{Hebrew}»`. While there are no Unicode properties with the same names as blocks, the property `«\p{Currency_Symbol}»` is confusingly similar to the block `«\p{Currency}»`. As I explained in the section on Unicode blocks, the characters they match are quite different. To avoid all such confusion, I strongly recommend you use the “In” syntax for blocks, the “Is” syntax for scripts (if supported), and the shorthand syntax `«\p{Lu}»` for properties.

Again, the JGsoft engine supports all of the above oddball notations. This is only done to allow you to copy and paste regular expressions and have them work as they do in Perl or Java. You should consider these notations deprecated.

Do You Need To Worry About Different Encodings?

While you should always keep in mind the pitfalls created by the different ways in which accented characters can be encoded, you don’t always have to worry about them. If you know that your input string and your regex use the same style, then you don’t have to worry about it at all. This process is called *Unicode normalization*. All programming languages with native Unicode support, such as Java, C# and VB.NET, have library routines for normalizing strings. If you normalize both the subject and regex before attempting the match, there won’t be any inconsistencies.

If you are using Java, you can pass the `CANON_EQ` flag as the second parameter to `Pattern.compile()`. This tells the Java regex engine to consider *canonically equivalent* characters as identical. E.g. the regex `«à»` encoded as `U+00E0` will match `„ă”` encoded as `U+0061 U+0300`, and vice versa. None of the other regex engines currently support canonical equivalence while matching.

If you type the à key on the keyboard, all word processors that I know of will insert the code point `U+00E0` into the file. So if you’re working with text that you typed in yourself, any regex that you type in yourself will match in the same way.

Finally, if you’re using PowerGREP to search through text files encoded using a traditional Windows (often called “ANSI”) or ISO-8859 code page, PowerGREP will always use the one-on-one substitution. Since all the Windows or ISO-8859 code pages encode accented characters as a single code point, all software that I know of will use a single Unicode code point for each character when converting the file to Unicode.

15. Regex Matching Modes

Most regular expression engines discussed in this tutorial support the following four matching modes:

- `/i` makes the regex match case insensitive.
- `/s` enables "single-line mode". In this mode, the dot matches newlines.
- `/m` enables "multi-line mode". In this mode, the caret and dollar match before and after newlines in the subject string.
- `/x` enables "free-spacing mode". In this mode, whitespace between regex tokens is ignored, and an unescaped `#` starts a comment.

Two languages that don't support all of the above four are JavaScript and Ruby. Some regex flavors also have additional modes or options that have single letter equivalents. These are very implementation-dependent.

Most tools that support regular expressions have checkboxes or similar controls that you can use to turn these modes on or off. Most programming languages allow you to pass option flags when constructing the regex object. E.g. in Perl, `m/regex/i` turns on case insensitivity, while `Pattern.compile("regex", Pattern.CASE_INSENSITIVE)` does the same in Java.

Specifying Modes Inside The Regular Expression

Sometimes, the tool or language does not provide the ability to specify matching options. E.g. the handy `String.matches()` method in Java does not take a parameter for matching options like `Pattern.compile()` does.

In that situation, you can add a mode modifier to the start of the regex. E.g. `(?i)` turns on case insensitivity, while `(?ism)` turns on all three options.

Turning Modes On and Off for Only Part of The Regular Expression

Modern regex flavors allow you to apply modifiers to only part of the regular expression. If you insert the modifier `(?ism)` in the middle of the regex, the modifier only applies to the part of the regex to the right of the modifier. You can turn off modes by preceding them with a minus sign. All modes after the minus sign will be turned off. E.g. `(?i-sm)` turns on case insensitivity, and turns off both single-line mode and multi-line mode.

Not all regex flavors support this. JavaScript and Python apply all mode modifiers to the entire regular expression. They don't support the `(?-ismx)` syntax, since turning off an option is pointless when mode modifiers apply to the whole regular expressions. All options are off by default.

You can quickly test how the regex flavor you're using handles mode modifiers. The regex `«(?i)te(?-i)st»` should match „test” and „TEst”, but not “teST” or “TEST”.

Modifier Spans

Instead of using two modifiers, one to turn an option on, and one to turn it off, you use a modifier span. «(?i)ignorecase(?-i)casesensitive(?i)ignorecase» is equivalent to «(?i)ignorecase(?-i:casesensitive)ignorecase». You have probably noticed the resemblance between the modifier span and the non-capturing group «(?:group)». Technically, the non-capturing group is a modifier span that does not change any modifiers. It is obvious that the modifier span does not create a backreference.

Modifier spans are supported by all regex flavors that allow you to use mode modifiers in the middle of the regular expression, and by those flavors only. These include the JGsoft engine, .NET, Java, Perl and PCRE.

16. Possessive Quantifiers

When discussing the repetition operators or quantifiers, I explained the difference between greedy and lazy repetition. Greediness and laziness determine the order in which the regex engine tries the possible permutations of the regex pattern. A greedy quantifier will first try to repeat the token as many times as possible, and gradually give up matches as the engine backtracks to find an overall match. A lazy quantifier will first repeat the token as few times as required, and gradually expand the match as the engine backtracks through the regex to find an overall match.

Because greediness and laziness change the order in which permutations are tried, they can change the overall regex match. However, they do not change the fact that the regex engine will backtrack to try all possible permutations of the regular expression in case no match can be found.

Possessive quantifiers are a way to prevent the regex engine from trying all permutations. This is primarily useful for performance reasons. You can also use possessive quantifiers to eliminate certain matches.

How Possessive Quantifiers Work

Several modern regular expression flavors, including the JGsoft, Java and PCRE have a third kind of quantifier: the possessive quantifier. Like a greedy quantifier, a possessive quantifier will repeat the token as many times as possible. Unlike a greedy quantifier, it will *not* give up matches as the engine backtracks. With a possessive quantifier, the deal is all or nothing. You can make a quantifier possessive by placing an extra + after it. E.g. «*» is greedy, «*?» is lazy, and «*+» is possessive. «++», «?+» and «{n,m}+» are all possessive as well.

Let's see what happens if we try to match «"[^"]*+» against "abc". The «"» matches the „"”. «"[^"]*» matches „a”, „b” and „c” as it is repeated by the star. The final «"» then matches the final „"” and we found an overall match. In this case, the end result is the same, whether we use a greedy or possessive quantifier. There is a slight performance increase though, because the possessive quantifier doesn't have to remember any backtracking positions.

The performance increase can be significant in situations where the regex fails. If the subject is "abc" (no closing quote), the above matching process will happen in the same way, except that the second «"» fails. When using a possessive quantifier, there are no steps to backtrack to. The regular expression does not have any alternation or non-possessive quantifiers that can give up part of their match to try a different permutation of the regular expression. So the match attempt fails immediately when the second «"» fails.

Had we used a greedy quantifier instead, the engine would have backtracked. After the «"» failed at the end of the string, the «"[^"]*» would give up one match, leaving it with „ab”. The «"» would then fail to match “c”. «"[^"]*» backtracks to just „a”, and «"» fails to match “b”. Finally, «"[^"]*» backtracks to match zero characters, and «"» fails “a”. Only at this point have all backtracking positions been exhausted, and does the engine give up the match attempt. Essentially, this regex performs as many needless steps as there are characters following the unmatched opening quote.

When Possessive Quantifiers Matter

The main practical benefit of possessive quantifiers is to speed up your regular expression. In particular, possessive quantifiers allow your regex to fail faster. In the above example, when the closing quote fails to match, we *know* the regular expression couldn't have possibly skipped over a quote. So there's no need to backtrack and check for the quote. We make the regex engine aware of this by making the quantifier possessive. In fact, some engines, including the JGsoft engine detect that «`[^"]*`» and «`"`» are mutually exclusive when compiling your regular expression, and automatically make the star possessive.

Now, linear backtracking like a regex with a single quantifier does is pretty fast. It's unlikely you'll notice the speed difference. However, when you're nesting quantifiers, a possessive quantifier may save your day. Nesting quantifiers means that you have one or more repeated tokens inside a group, and the group is also repeated. That's when catastrophic backtracking often rears its ugly head. In such cases, you'll depend on possessive quantifiers and/or atomic grouping to save the day.

Possessive Quantifiers Can Change The Match Result

Using possessive quantifiers can change the result of a match attempt. Since no backtracking is done, and matches that would require a greedy quantifier to backtrack will not be found with a possessive quantifier. E.g. «`" . *`» will match „`abc`” in “`abc x`”, but «`" . *+`» will not match this string at all.

In both regular expressions, the first «`"`» will match the first „`"`” in the string. The repeated dot then matches the remainder of the string „`abc x`”. The second «`"`» then fails to match at the end of the string.

Now, the paths of the two regular expressions diverge. The possessive dot-star wants it all. No backtracking is done. Since the «`"`» failed, there are no permutations left to try, and the overall match attempt fails. The greedy dot-star, while initially grabbing everything, is willing to give back. It will backtrack one character at a time. Backtracking to „`abc`”, «`"`» fails to match “`x`”. Backtracking to „`abc`”, «`"`» matches „`"`”. An overall match „`abc`” was found.

Essentially, the lesson here is that when using possessive quantifiers, you need to make sure that whatever you're applying the possessive quantifier to should not be able to match what should follow it. The problem in the above example is that the dot also matches the closing quote. This prevents us from using a possessive quantifier. The negated character class in the previous section cannot match the closing quote, so we can make it possessive.

Using Atomic Grouping Instead of Possessive Quantifiers

Technically, possessive quantifiers are a notational convenience to place an atomic group around a single quantifier. All regex flavors that support possessive quantifiers also support atomic grouping. But not all regex flavors that support atomic grouping support possessive quantifiers. With those flavors, you can achieve the exact same results using an atomic group.

Basically, instead of «`X*+`», write «`(?>X*)`». It is important to notice that both the quantified token X and the quantifier are inside the atomic group. Even if X is a group, you still need to put an extra atomic group around it to achieve the same effect. «`(?: a | b) *+`» is equivalent to «`(?> (?: a | b) *)`» but not to «`(?> a | b) *`».

The latter is a valid regular expression, but it won't have the same effect when used as part of a larger regular expression.

E.g. `«(?:a|b)*+b»` and `«(?:?(?:a|b)*)b»` both fail to match "b". `«a|b»` will match the „b”. The star is satisfied, and the fact that it's possessive or the atomic group will cause the star to forget all its backtracking positions. The second `«b»` in the regex has nothing left to match, and the overall match attempt fails.

In the regex `«(?:>a|b)*b»`, the atomic group forces the alternation to give up its backtracking positions. I.e. if an „a” is matched, it won't come back to try `«b»` if the rest of the regex fails. Since the star is outside of the group, it is a normal, greedy star. When the second `«b»` fails, the greedy star will backtrack to zero iterations. Then, the second `«b»` matches the „b” in the subject string.

This distinction is particularly important when converting a regular expression written by somebody else using possessive quantifiers to a regex flavor that doesn't have possessive quantifiers. You could, of course, let a tool like `RegexBuddy` do the job for you.

17. Atomic Grouping

An atomic group is a group that, when the regex engine exits from it, automatically throws away all backtracking positions remembered by any tokens inside the group. Atomic groups are non-capturing. The syntax is `«(?:>group)»`. Lookaround groups are also atomic. Atomic grouping is supported by most modern regular expression flavors, including the JGsoft flavor, Java, PCRE, .NET, Perl and Ruby. The first three of these also support possessive quantifiers, which are essentially a notational convenience for atomic grouping.

An example will make the behavior of atomic groups clear. The regular expression `«a(bc|b)c»` (capturing group) matches „abcc” and „abc”. The regex `«a(?:>bc|b)c»` (atomic group) matches „abcc” but not “abc”.

When applied to “abc”, both regexes will match `«a»` to „a”, `«bc»` to „bc”, and then `«c»` will fail to match at the end of the string. Here their paths diverge. The regex with the capturing group has remembered a backtracking position for the alternation. The group will give up its match, `«b»` then matches „b” and `«c»` matches „c”. Match found!

The regex with the atomic group, however, exited from an atomic group after `«bc»` was matched. At that point, all backtracking positions for tokens inside the group are discarded. In this example, the alternation’s option to try `«b»` at the second position in the string is discarded. As a result, when `«c»` fails, the regex engine has no alternatives left to try.

Of course, the above example isn’t very useful. But it does illustrate very clearly how atomic grouping eliminates certain matches. Or more importantly, it eliminates certain match attempts.

Regex Optimization Using Atomic Grouping

Consider the regex `«\b(integer|insert|in)\b»` and the subject “integers”. Obviously, because of the word boundaries, these don’t match. What’s not so obvious is that the regex engine will spend quite some effort figuring this out.

`«\b»` matches at the start of the string, and `«integer»` matches „integer”. The regex engine makes note that there are two more alternatives in the group, and continues with `«\b»`. This fails to match between the “r” and “s”. So the engine backtracks to try the second alternative inside the group. The second alternative matches „in”, but then fails to match «s». So the engine backtracks once more to the third alternative. `«in»` matches „in”. `«\b»` fails between the “n” and “t” this time. The regex engine has no more remembered backtracking positions, so it declares failure.

This is quite a lot of work to figure out “integers” isn’t in our list of words. We can optimize this by telling the regular expression engine that if it can’t match `«\b»` after it matched „integer”, then it shouldn’t bother trying any of the other words. The word we’ve encountered in the subject string is a longer word, and it isn’t in our list.

We can do this by turning the capturing group into an atomic group: `«\b(?:>integer|insert|in)\b»`. Now, when `«integer»` matches, the engine exits from an atomic group, and throws away the backtracking positions it stored for the alternation. When `«\b»` fails, the engine gives up immediately. This savings can be significant when scanning a large file for a long list of keywords. This savings will be vital when your alternatives contain repeated tokens (not to mention repeated groups) that lead to catastrophic backtracking.

Don't be too quick to make all your groups atomic. As we saw in the first example above, atomic grouping can exclude valid matches too. Compare how `«\b(?:integer|insert|in)\b»` and `«\b(?:in|integer|insert)\b»` behave when applied to "insert". The former regex matches, while the latter fails. If the groups weren't atomic, both regexes would match. Remember that alternation tries its alternatives from left to right. If the second regex matches „in”, it won't try the two other alternatives due to the atomic group.

18. Lookahead and Lookbehind Zero-Width Assertions

Perl 5 introduced two very powerful constructs: “lookahead” and “lookbehind”. Collectively, these are called “lookaround”. They are also called “zero-width assertions”. They are zero-width just like the start and end of line, and start and end of word anchors that I already explained. The difference is that lookarounds will actually match characters, but then give up the match and only return the result: match or no match. That is why they are called “assertions”. They do not consume characters in the string, but only assert whether a match is possible or not. Lookarounds allow you to create regular expressions that are impossible to create without them, or that would get very longwinded without them.

Positive and Negative Lookahead

Negative lookahead is indispensable if you want to match something not followed by something else. When explaining character classes, I already explained why you cannot use a negated character class to match a “q” not followed by a “u”. Negative lookahead provides the solution: «q(?!u)». The negative lookahead construct is the pair of round brackets, with the opening bracket followed by a question mark and an exclamation point. Inside the lookahead, we have the trivial regex «u».

Positive lookahead works just the same. «q(=u)» matches a q that is followed by a u, without making the u part of the match. The positive lookahead construct is a pair of round brackets, with the opening bracket followed by a question mark and an equals sign.

You can use any regular expression inside the lookahead. (Note that this is not the case with lookbehind. I will explain why below.) Any valid regular expression can be used inside the lookahead. If it contains capturing parentheses, the backreferences will be saved. Note that the lookahead itself does not create a backreference. So it is not included in the count towards numbering the backreferences. If you want to store the match of the regex inside a backreference, you have to put capturing parentheses around the regex inside the lookahead, like this: «(?(= (regex)))». The other way around will not work, because the lookahead will already have discarded the regex match by the time the backreference is to be saved.

Regex Engine Internals

First, let’s see how the engine applies «q(?!u)» to the string “Iraq”. The first token in the regex is the literal «q». As we already know, this will cause the engine to traverse the string until the „q” in the string is matched. The position in the string is now the void behind the string. The next token is the lookahead. The engine takes note that it is inside a lookahead construct now, and begins matching the regex inside the lookahead. So the next token is «u». This does not match the void behind the string. The engine notes that the regex inside the lookahead failed. Because the lookahead is negative, this means that the lookahead has successfully matched at the current position. At this point, the entire regex has matched, and „q” is returned as the match.

Let’s try applying the same regex to “quit”. «q» matches „q”. The next token is the «u» inside the lookahead. The next character is the “u”. These match. The engine advances to the next character: “i”. However, it is done with the regex inside the lookahead. The engine notes success, and discards the regex match. This causes the engine to step back in the string to “u”.

Because the lookahead is negative, the successful match inside it causes the lookahead to fail. Since there are no other permutations of this regex, the engine has to start again at the beginning. Since «q» cannot match anywhere else, the engine reports failure.

Let's take one more look inside, to make sure you understand the implications of the lookahead. Let's apply «q(?=u)i» to "quit". I have made the lookahead positive, and put a token after it. Again, «q» matches „q” and «u» matches „u”. Again, the match from the lookahead must be discarded, so the engine steps back from “i” in the string to “u”. The lookahead was successful, so the engine continues with «i». But «i» cannot match “u”. So this match attempt fails. All remaining attempts will fail as well, because there are no more q's in the string.

Positive and Negative Lookbehind

Lookbehind has the same effect, but works backwards. It tells the regex engine to temporarily step backwards in the string, to check if the text inside the lookbehind can be matched there. «(?!a)b» matches a “b” that is not preceded by an “a”, using negative lookbehind. It will not match “cab”, but will match the „b” (and only the „b”) in “bed” or “debt”. «(?<=a)b» (positive lookbehind) matches the „b” (and only the „b”) in „cab”, but does not match “bed” or “debt”.

The construct for positive lookbehind is «(?<=text)»: a pair of round brackets, with the opening bracket followed by a question mark, “less than” symbol and an equals sign. Negative lookbehind is written as «(?<!text)», using an exclamation point instead of an equals sign.

More Regex Engine Internals

Let's apply «(?<=a)b» to “thingamabob”. The engine starts with the lookbehind and the first character in the string. In this case, the lookbehind tells the engine to step back one character, and see if an “a” can be matched there. The engine cannot step back one character because there are no characters before the “t”. So the lookbehind fails, and the engine starts again at the next character, the “h”. (Note that a negative lookbehind would have succeeded here.) Again, the engine temporarily steps back one character to check if an “a” can be found there. It finds a “t”, so the positive lookbehind fails again.

The lookbehind continues to fail until the regex reaches the “m” in the string. The engine again steps back one character, and notices that the „a” can be matched there. The positive lookbehind matches. Because it is zero-width, the current position in the string remains at the “m”. The next token is «b», which cannot match here. The next character is the second “a” in the string. The engine steps back, and finds out that the “m” does not match «a».

The next character is the first “b” in the string. The engine steps back and finds out that „a” satisfies the lookbehind. «b» matches „b”, and the entire regex has been matched successfully. It matches one character: the first „b” in the string.

Important Notes About Lookbehind

The good news is that you can use lookbehind anywhere in the regex, not only at the start. If you want to find a word not ending with an “s”, you could use «\b\w+(?!s)\b». This is definitely not the same as

«\b\w+[^s]\b». When applied to “John's”, the former will match „John” and the latter „John'” (including the apostrophe). I will leave it up to you to figure out why. (Hint: «\b» matches between the apostrophe and the “s”). The latter will also not match single-letter words like “a” or “I”. The correct regex without using lookbehind is «\b\w*[^s\W]\b» (star instead of plus, and \W in the character class). Personally, I find the lookbehind easier to understand. The last regex, which works correctly, has a double negation (the \W in the negated character class). Double negations tend to be confusing to humans. Not to regex engines, though.

The bad news is that most regex flavors do not allow you to use just any regex inside a lookbehind, because they cannot apply a regular expression backwards. Therefore, the regular expression engine needs to be able to figure out how many steps to step back before checking the lookbehind.

Therefore, many regex flavors, including those used by Perl and Python, only allow fixed-length strings. You can use any regex of which the length of the match can be predetermined. This means you can use literal text and character classes. You cannot use repetition or optional items. You can use alternation, but only if all options in the alternation have the same length.

PCRE is not fully Perl-compatible when it comes to lookbehind. While Perl requires alternatives inside lookbehind to have the same length, PCRE allows alternatives of variable length. Each alternative still has to be fixed-length.

Java takes things a step further by allowing finite repetition. You still cannot use the star or plus, but you can use the question mark and the curly braces with the max parameter specified. Java recognizes the fact that finite repetition can be rewritten as an alternation of strings with different, but fixed lengths. Unfortunately, the JDK 1.4 and 1.5 have some bugs when you use alternation inside lookbehind. These were fixed in JDK 1.6.

The only regex engines that allow you to use a full regular expression inside lookbehind, including infinite repetition, are the JGsoft engine and the .NET framework RegEx classes.

Finally, flavors like JavaScript, Ruby and Tcl do not support lookbehind at all, even though they do support lookahead.

Lookaround Is Atomic

The fact that lookaround is zero-width automatically makes it atomic. As soon as the lookaround condition is satisfied, the regex engine forgets about everything inside the lookaround. It will not backtrack inside the lookaround to try different permutations.

The only situation in which this makes any difference is when you use capturing groups inside the lookaround. Since the regex engine does not backtrack into the lookaround, it will not try different permutations of the capturing groups.

For this reason, the regex «(?=(\d+)\w+\1)» will never match “123x12”. First the lookaround captures „123” into «\1». «\w+» then matches the whole string and backtracks until it matches only „1”. Finally, «\w+» fails since «\1» cannot be matched at any position. Now, the regex engine has nothing to backtrack to, and the overall regex fails. The backtracking steps created by «\d+» have been discarded. It never gets to the point where the lookahead captures only “12”.

Obviously, the regex engine does try further positions in the string. If we change the subject string, the regex «`(?=(\d+)\w+\1)`» will match „56x56” in “456x56”.

If you don't use capturing groups inside lookahead, then all this doesn't matter. Either the lookahead condition can be satisfied or it cannot be. In how many ways it can be satisfied is irrelevant.

19. Testing The Same Part of a String for More Than One Requirement

Lookaround, which I introduced in detail in the previous topic, is a very powerful concept. Unfortunately, it is often underused by people new to regular expressions, because lookaround is a bit confusing. The confusing part is that the lookaround is zero-width. So if you have a regex in which a lookahead is followed by another piece of regex, or a lookbehind is preceded by another piece of regex, then the regex will traverse part of the string twice.

To make this clear, I would like to give you another, a bit more practical example. Let's say we want to find a word that is six letters long and contains the three consecutive letters "cat". Actually, we can match this without lookaround. We just specify all the options and lump them together using alternation: `«cat\w{3}|\wcat\w{2}|\w{2}cat\w|\w{3}cat»`. Easy enough. But this method gets unwieldy if you want to find any word between 6 and 12 letters long containing either "cat", "dog" or "mouse".

Lookaround to The Rescue

In this example, we basically have two requirements for a successful match. First, we want a word that is 6 letters long. Second, the word we found must contain the word "cat".

Matching a 6-letter word is easy with `«\b\w{6}\b»`. Matching a word containing "cat" is equally easy: `«\b\w*cat\w*\b»`.

Combining the two, we get: `«(?:=\b\w{6}\b)\b\w*cat\w*\b»`. Easy! Here's how this works. At each character position in the string where the regex is attempted, the engine will first attempt the regex inside the positive lookahead. This sub-regex, and therefore the lookahead, matches only when the current character position in the string is at the start of a 6-letter word in the string. If not, the lookahead will fail, and the engine will continue trying the regex from the start at the next character position in the string.

The lookahead is zero-width. So when the regex inside the lookahead has found the 6-letter word, the current position in the string is still at the beginning of the 6-letter word. At this position will the regex engine attempt the remainder of the regex. Because we already know that a 6-letter word can be matched at the current position, we know that `«\b»` matches and that the first `«\w*»` will match 6 times. The engine will then backtrack, reducing the number of characters matched by `«\w*»`, until `«cat»` can be matched. If `«cat»` cannot be matched, the engine has no other choice but to restart at the beginning of the regex, at the next character position in the string. This is at the second letter in the 6-letter word we just found, where the lookahead will fail, causing the engine to advance character by character until the next 6-letter word.

If `«cat»` can be successfully matched, the second `«\w*»` will consume the remaining letters, if any, in the 6-letter word. After that, the last `«\b»` in the regex is guaranteed to match where the second `«\b»` inside the lookahead matched. Our double-requirement-regex has matched successfully.

Optimizing Our Solution

While the above regex works just fine, it is not the most optimal solution. This is not a problem if you are just doing a search in a text editor. But optimizing things is a good idea if this regex will be used repeatedly and/or on large chunks of data in an application you are developing.

You can discover these optimizations by yourself if you carefully examine the regex and follow how the regex engine applies it, as I did above. I said the third and last «\b» are guaranteed to match. Since it is zero-width, and therefore does not change the result returned by the regex engine, we can remove them, leaving: «(?:\b\w{6}\b)\w*cat\w*». Though the last «\w*» is also guaranteed to match, we cannot remove it because it adds characters to the regex match. Remember that the lookahead discards its match, so it does not contribute to the match returned by the regex engine. If we omitted the «\w*», the resulting match would be the start of a 6-letter word containing “cat”, up to and including “cat”, instead of the entire word.

But we can optimize the first «\w*». As it stands, it will match 6 letters and then backtrack. But we know that in a successful match, there can never be more than 3 letters before “cat”. So we can optimize this to «\w{0,3}». Note that making the asterisk lazy would not have optimized this sufficiently. The lazy asterisk would find a successful match sooner, but if a 6-letter word does not contain “cat”, it would still cause the regex engine to try matching “cat” at the last two letters, at the last single letter, and even at one character beyond the 6-letter word.

So we have «(?:\b\w{6}\b)\w{0,3}cat\w*». One last, minor, optimization involves the first «\b». Since it is zero-width itself, there’s no need to put it inside the lookahead. So the final regex is: «\b(?:\w{6}\b)\w{0,3}cat\w*».

You could replace the final «\w*» with «\w{0,3}» too. But it wouldn’t make any difference. The lookahead has already checked that we’re at a 6-letter word, and «\w{0,3}cat» has already matched 3 to 6 letters of that word. Whether we end the regex with «\w*» or «\w{0,3}» doesn’t matter, because either way, we’ll be matching all the remaining word characters. Because the resulting match and the speed at which it is found are the same, we may just as well use the version that is easier to type.

A More Complex Problem

So, what would you use to find any word between 6 and 12 letters long containing either “cat”, “dog” or “mouse”? Again we have two requirements, which we can easily combine using a lookahead: «\b(?:\w{6,12}\b)\w{0,9}(cat|dog|mouse)\w*». Very easy, once you get the hang of it. This regex will also put “cat”, “dog” or “mouse” into the first backreference.

20. Continuing at The End of The Previous Match

The anchor «\G» matches at the position where the previous match ended. During the first match attempt, «\G» matches at the start of the string in the way «\A» does.

Applying «\G\w» to the string “test string” matches „t”. Applying it again matches „e”. The 3rd attempt yields „s” and the 4th attempt matches the second „t” in the string. The fifth attempt fails. During the fifth attempt, the only place in the string where «\G» matches is after the second t. But that position is not followed by a word character, so the match fails.

End of The Previous Match vs. Start of The Match Attempt

With some regex flavors or tools, «\G» matches at the start of the match attempt, rather than at the end of the previous match result. This is the case with EditPad Pro, where «\G» matches at the position of the text cursor, rather than the end of the previous match. When a match is found, EditPad Pro will select the match, and move the text cursor to the end of the match. The result is that «\G» matches at the end of the previous match result only when you do not move the text cursor between two searches. All in all, this makes a lot of sense in the context of a text editor.

\G Magic with Perl

In Perl, the position where the last match ended is a “magical” value that is remembered separately for each string variable. The position is not associated with any regular expression. This means that you can use «\G» to make a regex continue in a subject string where another regex left off.

If a match attempt fails, the stored position for «\G» is reset to the start of the string. To avoid this, specify the continuation modifier /c.

All this is very useful to make several regular expressions work together. E.g. you could parse an HTML file in the following fashion:

```
while ($string =~ m/</g) {
  if ($string =~ m/\GB>/c) {
    # Bold
  } elsif ($string =~ m/\GI>/c) {
    # Italics
  } else {
    # ...etc...
  }
}
```

The regex in the while loop searches for the tag’s opening bracket, and the regexes inside the loop check which tag we found. This way you can parse the tags in the file in the order they appear in the file, without having to write a single big regex that matches all tags you are interested in.

\G in Other Programming Languages

This flexibility is not available with most other programming languages. E.g. in Java, the position for «\G» is remembered by the Matcher object. The Matcher is strictly associated with a single regular expression and a single subject string. What you can do though is to add a line of code to make the match attempt of the second Matcher start where the match of the first Matcher ended. «\G» will then match at this position.

The «\G» token is supported by the JGsoft engine, .NET, Java, Perl and PCRE.

21. If-Then-Else Conditionals in Regular Expressions

A special construct `«(?ifthen|else)»` allows you to create conditional regular expressions. If the *if* part evaluates to true, then the regex engine will attempt to match the *then* part. Otherwise, the *else* part is attempted instead. The syntax consists of a pair of round brackets. The opening bracket must be followed by a question mark, immediately followed by the *if* part, immediately followed by the *then* part. This part can be followed by a vertical bar and the *else* part. You may omit the *else* part, and the vertical bar with it.

For the *if* part, you can use the lookahead and lookbehind constructs. Using positive lookahead, the syntax becomes `«(?(?=regex)then|else)»`. Because the lookahead has its own parentheses, the *if* and *then* parts are clearly separated.

Remember that the lookaround constructs do not consume any characters. If you use a lookahead as the *if* part, then the regex engine will attempt to match the *then* or *else part* (depending on the outcome of the lookahead) at the same position where the *if* was attempted.

Alternatively, you can check in the *if* part whether a capturing group has taken part in the match thus far. Place the number of the capturing group inside round brackets, and use that as the *if* part. Note that although the syntax for a conditional check on a backreference is the same as a number inside a capturing group, no capturing group is created. The number and the brackets are part of the if-then-else syntax started with `«(?)»`.

For the *then* and *else*, you can use any regular expression. If you want to use alternation, you will have to group the *then* or *else* together using parentheses, like in `«(?(?=condition)(then1|then2|then3)|(else1|else2|else3))»`. Otherwise, there is no need to use parentheses around the *then* and *else* parts.

Looking Inside the Regex Engine

The regex `«(a)?b(1)c|d)»` matches „bd” and „abc”. It does not match “bc”, but does match „bd” in “abd”. Let’s see how this regular expression works on each of these four subject strings.

When applied to “bd”, «a» fails to match. Since the capturing group containing «a» is optional, the engine continues with «b» at the start of the subject string. Since the whole group was optional, the group did not take part in the match. Any subsequent backreference to it like «\1» will fail. Note that «(a)?» is very different from «a?»». In the former regex, the capturing group does not take part in the match if «a» fails, and backreferences to the group will fail. In the latter group, the capturing group always takes part in the match, capturing either „a” or nothing. Backreferences to a capturing group that took part in the match and captured nothing always succeed. Conditionals evaluating such groups execute the “then” part. In short: if you want to use a reference to a group in a conditional, use «(a)?» instead of «a?»».

Continuing with our regex, «b» matches „b”. The regex engine now evaluates the conditional. The first capturing group did not take part in the match at all, so the “else” part or «d)» is attempted. «d)» matches „d)” and an overall match is found.

Moving on to our second subject string “abc”, «a)» matches „a)”, which is captured by the capturing group. Subsequently, «b)» matches „b)”. The regex engine again evaluates the conditional. The capturing group took part in the match, so the “then” part or «c)» is attempted. «c)» matches „c)” and an overall match is found.

Our third subject “bc” does not start with “a”, so the capturing group does not take part in the match attempt, like we saw with the first subject string. «b» still matches „b”, and the engine moves on to the conditional. The first capturing group did not take part in the match at all, so the “else” part or «d» is attempted. «d» does not match “c” and the match attempt at the start of the string fails. The engine does try again starting at the second character in the string, but fails since «b» does not match “c”.

The fourth subject “abd” is the most interesting one. Like in the second string, the capturing group grabs the „a” and the «b» matches. The capturing group took part in the match, so the “then” part or «c» is attempted. «c» fails to match “d”, and the match attempt fails. Note that the “else” part is not attempted at this point. The capturing group took part in the match, so only the “then” part is used. However, the regex engine isn’t done yet. It will restart the regular expression from the beginning, moving ahead one character in the subject string.

Starting at the second character in the string, «a» fails to match “b”. The capturing group does not take part in the second match attempt which started at the second character in the string. The regex engine moves beyond the optional group, and attempts «b», which matches. The regex engine now arrives at the conditional in the regex, and at the third character in the subject string. The first capturing group did not take part in the current match attempt, so the “else” part or «d» is attempted. «d» matches „d” and an overall match „bd” is found.

If you want to avoid this last match result, you need to use anchors. «^(a)?b?(1)c|d\$» does not find any matches in the last subject string. The caret will fail to match at the second and third characters in the string.

Regex Flavors

Conditionals are supported by the JGsoft engine, Perl, PCRE and the .NET framework. All these flavors, except Perl, also support named capturing groups. They allow you to use the name of a capturing group instead of its number as the *if* test, e.g.: «(<test>a)?b?(test)c|d».

Python supports conditionals using a numbered or named capturing group. Python does not support conditionals using lookahead, even though Python does support lookahead outside conditionals. Instead of a conditional like «(?(?=regex)then|else)», you can alternate two opposite lookarounds: «(?=regex)then|(?!regex)else».

Example: Extract Email Headers

The regex «^((From|To)|Subject): ((?(2)\w+@\w+\.[a-z]+|.+)» extracts the From, To, and Subject headers from an email message. The name of the header is captured into the first backreference. If the header is the From or To header, it is captured into the second backreference as well.

The second part of the pattern is the if-then-else conditional «(?(2)\w+@\w+\.[a-z]+|.+)». The *if* part checks whether the second capturing group took part in the match thus far. It will have taken part if the header is the From or To header. In that case, the *then* part of the conditional «\w+@\w+\.[a-z]+» tries to match an email address. To keep the example simple, we use an overly simple regex to match the email address, and we don’t try to match the display name that is usually also part of the From or To header.

If the second capturing group did not participate in the match this far, the *else* part «. +» is attempted instead. This simply matches the remainder of the line, allowing for any test subject.

Finally, we place an extra pair of round brackets around the conditional. This captures the contents of the email header matched by the conditional into the third backreference. The conditional itself does not capture anything. When implementing this regular expression, the first capturing group will store the name of the header (“From”, “To”, or “Subject”), and the third capturing group will store the value of the header.

You could try to match even more headers by putting another conditional into the “else” part. E.g. «[^]((From|To)|(Date)|Subject): ((?²\w+@\w+\.[a-z]+|(?³mm/dd/yyyy|.+))» would match a “From”, “To”, “Date” or “Subject”, and use the regex «mm/dd/yyyy» to check whether the date is valid. Obviously, the date validation regex is just a dummy to keep the example simple. The header is captured in the first group, and its validated contents in the fourth group.

As you can see, regular expressions using conditionals quickly become unwieldy. I recommend that you only use them if one regular expression is all your tool allows you to use. When programming, you’re far better off using the regex «[^](From|To|Date|Subject): (.+)» to capture one header with its unvalidated contents. In your source code, check the name of the header returned in the first capturing group, and then use a second regular expression to validate the contents of the header returned in the second capturing group of the first regex. Though you’ll have to write a few lines of extra code, this code will be much easier to understand and maintain. If you precompile all the regular expressions, using multiple regular expressions will be just as fast, if not faster, than the one big regex stuffed with conditionals.

22. XML Schema Character Classes

XML Schema Regular Expressions support the usual six shorthand character classes, plus four more. These four aren't supported by any other regular expression flavor. «\i» matches any character that may be the first character of an XML name, i.e. «[_:A-Za-z]». «\c» matches any character that may occur after the first character in an XML name, i.e. «[-. _:A-Za-z0-9]». «\I» and «\C» are the respective negated shorthands. Note that the «\c» shorthand syntax conflicts with the control character syntax used in many other regex flavors.

You can use these four shorthands both inside and outside character classes using the bracket notation. They're very useful for validating XML references and values in your XML schemas. The regular expression «\i\c*» matches an XML name like „xml: schema”. In other regular expression flavors, you'd have to spell this out as «[_:A-Za-z][-. _:A-Za-z0-9]*». The latter regex also works with XML's regular expression flavor. It just takes more time to type in.

The regex «<\i\c*\s*>» matches an opening XML tag without any attributes. «</\i\c*\s*>» matches any closing tag. «<\i\c*(\s+\i\c*\s*=\s*("[^"]*"|'['']*'))*\s*>» matches an opening tag with any number of attributes. Putting it all together, «<(\i\c*(\s+\i\c*\s*=\s*("[^"]*"|'['']*'))*/\i\c*)\s*>» matches either an opening tag with attributes or a closing tag.

Character Class Subtraction

While the regex flavor it defines is quite limited, the XML Schema adds a new regular expression feature not previously seen in any (popular) regular expression flavor: character class subtraction. Currently, this feature is only supported by the JGsoft and .NET regex engines (in addition to those implementing the XML Schema standard).

Character class subtraction makes it easy to match any single character present in one list (the character class), but not present in another list (the subtracted class). The syntax for this is [class-[subtract]]. If the character after a hyphen is an opening bracket, XML regular expressions interpret the hyphen as the subtraction operator rather than the range operator. E.g. «[a-z-[aeiou]]» matches a single letter that is not a vowel (i.e. a single consonant). Without the character class subtraction feature, the only way to do this would be to list all consonants: «[b-df-hj-np-tv-z]».

This feature is more than just a notational convenience, though. You can use the full character class syntax within the subtracted character class. E.g. to match all Unicode letters except ASCII letters (i.e. all non-English letters), you could easily use «[\p{L}-[\p{IsBasicLatin}]]».

Nested Character Class Subtraction

Since you can use the full character class syntax within the subtracted character class, you can subtract a class from the class being subtracted. E.g. «[0-9-[0-6-[0-3]]]» first subtracts 0-3 from 0-6, yielding «[0-9-[4-6]]», or «[0-37-9]», which matches any character in the string “0123789”.

The class subtraction must always be the last element in the character class. [0-9-[4-6]a-f] is not a valid regular expression. It should be rewritten as «[0-9a-f-[4-6]]». The subtraction works on the whole class.

E.g. `«[\p{Ll}\p{Lu}-[\p{IsBasicLatin}]]»` matches all uppercase and lowercase Unicode letters, except any ASCII letters. The `\p{IsBasicLatin}` is subtracted from the combination of `\p{Ll}\p{Lu}` rather than from `\p{Lu}` alone. This regex will not match “abc”.

While you can use nested character class subtraction, you cannot subtract two classes sequentially. To subtract ASCII letters and Greek letters from a class with all Unicode letters, combine the ASCII and Greek letters into one class, and subtract that, as in `«[\p{L}-[\p{IsBasicLatin}\p{IsGreek}]]»`.

Notational Compatibility with Other Regex Flavors

Note that a regex like `«[a-z-[aeiou]]»` will not cause any errors in regex flavors that do not support character class subtraction. But it won't match what you intended either. E.g. in Perl, this regex consists of a character class followed by a literal `«]»`. The character class matches a character that is either in the range a-z, or a hyphen, or an opening bracket, or a vowel. Since the a-z range and the vowels are redundant, you could write this character class as `«[a-z- []»` or `«[- [a-z]»`. A hyphen after a range is treated as a literal character, just like a hyphen immediately after the opening bracket. This is true in all regex flavors, including XML. E.g. `«[a-z- _]»` matches a lowercase letter, a hyphen or an underscore in both Perl and XML Schema.

While the last paragraph strictly speaking means that the XML Schema character class syntax is incompatible with Perl and the majority of other regex flavors, in practice there's no difference. Using non-alphanumeric characters in character class ranges is very bad practice, as it relies on the order of characters in the ASCII character table, which makes the regular expression hard to understand for the programmer who inherits your work. E.g. while `«[A- []»` would match any upper case letter or an opening square bracket in Perl, this regex is much clearer when written as `«[A-Z []»`. The former regex would cause an error in XML Schema, because it interprets `- []` as an empty subtracted class, leaving an unbalanced `[`.

23. POSIX Bracket Expressions

POSIX bracket expressions are a special kind of character classes. POSIX bracket expressions match one character out of a set of characters, just like regular character classes. They use the same syntax with square brackets. A hyphen creates a range, and a caret at the start negates the bracket expression.

One key syntactic difference is that the backslash is NOT a metacharacter in a POSIX bracket expression. So in POSIX, the regular expression «`[\d]`» matches a „\” or a „d”. To match a „]”, put it as the first character after the opening `[` or the negating `^`. To match a „-”, put it right before the closing `]`. To match a „^”, put it before the final literal `-` or the closing `]`. Put together, «`[^\d^-]`» matches „]”, „\”, „d”, „^” or „-”.

The main purpose of the bracket expressions is that they adapt to the user’s or application’s locale. A locale is a collection of rules and settings that describe language and cultural conventions, like sort order, date format, etc. The POSIX standard also defines these locales.

Generally, only POSIX-compliant regular expression engines have proper and full support for POSIX bracket expressions. Some non-POSIX regex engines support POSIX character classes, but usually don’t support collating sequences and character equivalents. Regular expression engines that support Unicode use Unicode properties and scripts to provide functionality similar to POSIX bracket expressions. In Unicode regex engines, shorthand character classes like «`\w`» normally match all relevant Unicode characters, alleviating the need to use locales.

Character Classes

Don’t confuse the POSIX term “character class” with what is normally called a regular expression character class. «`[x-z0-9]`» is an example of what we call a “character class” and POSIX calls a “bracket expression”. `[:digit:]` is a POSIX character class, used inside a bracket expression like «`[x-z[:digit:]]`». These two regular expressions match exactly the same: a single character that is either „x”, „y”, „z” or a digit. The class names must be written all lowercase.

POSIX bracket expressions can be negated. «`^[x-z[:digit:]]`» matches a single character that is not x, y, z or a digit. A major difference between POSIX bracket expressions and the character classes in other regex flavors is that POSIX bracket expressions treat the backslash as a literal character. This means you can’t use backslashes to escape the closing bracket `]`, the caret `^` and the hyphen `-`. To include a caret, place it anywhere except right after the opening bracket. «`[x^]`» matches an x or a caret. You can put the closing bracket right after the opening bracket, or the negating caret. «`[]x]`» matches a closing bracket or an x. «`[^]x]`» matches any character that is not a closing bracket or an x. The hyphen can be included right after the opening bracket, or right before the closing bracket, or right after the negating caret. Both «`[-x]`» and «`[x-]`» match an x or a hyphen.

Exactly which POSIX character classes are available depends on the POSIX locale. The following are usually supported, often also by regex engines that don’t support POSIX itself. I’ve also indicated equivalent character classes that you can use in ASCII and Unicode regular expressions if the POSIX classes are unavailable. Some classes also have Perl-style shorthand equivalents.

Java does not support POSIX bracket expressions, but does support POSIX character classes using the `\p` operator. Though the `\p` syntax is borrowed from the syntax for Unicode properties, the POSIX classes in Java only match ASCII characters as indicated below. The class names are case sensitive. Unlike the POSIX

syntax which can only be used inside a bracket expression, Java's `\p` can be used inside and outside bracket expressions.

POSIX: `«[:alnum:]»`
 Description: Alphanumeric characters
 ASCII: `«[a-zA-Z0-9]»`
 Unicode: `«[\p{L&}\p{Nd}]»`
 Shorthand:
 Java: `«\p{Alnum}»`

POSIX: `«[:alpha:]»`
 Description: Alphabetic characters
 ASCII: `«[a-zA-Z]»`
 Unicode: `«\p{L&}»`
 Shorthand:
 Java: `«\p{Alpha}»`

POSIX: `«[:ascii:]»`
 Description: ASCII characters
 ASCII: `«[\x00-\x7F]»`
 Unicode: `«\p{InBasicLatin}»`
 Shorthand:
 Java: `«\p{ASCII}»`

POSIX: `«[:blank:]»`
 Description: Space and tab
 ASCII: `«[\t]»`
 Unicode: `«[\p{Zs}\t]»`
 Shorthand:
 Java: `«\p{Blank}»`

POSIX: `«[:cntrl:]»`
 Description: Control characters
 ASCII: `«[\x00-\x1F\x7F]»`
 Unicode: `«\p{Cc}»`
 Shorthand:
 Java: `«\p{Cntrl}»`

POSIX: `«[:digit:]»`
 Description: Digits
 ASCII: `«[0-9]»`
 Unicode: `«\p{Nd}»`
 Shorthand: `«\d»`
 Java: `«\p{Digit}»`

POSIX: `«[:graph:]»`
 Description: Visible characters (i.e. anything except spaces, control characters, etc.)
 ASCII: `«[\x21-\x7E]»`
 Unicode: `«[^\p{Z}\p{C}]»`
 Shorthand:
 Java: `«\p{Graph}»`

POSIX:	«[:lower:]»
Description:	Lowercase letters
ASCII:	«[a-z]»
Unicode:	«\p{Ll}»
Shorthand:	
Java:	«\p{Lower}»
POSIX:	«[:print:]»
Description:	Visible characters and spaces (i.e. anything except control characters, etc.)
ASCII:	«[\x20-\x7E]»
Unicode:	«\P{C}»
Shorthand:	
Java:	«\p{Print}»
POSIX:	«[:punct:]»
Description:	Punctuation and symbols.
ASCII:	«[!"#\$%&'()*+,-./:;<=>?@[\\]^_`{ }~]»
Unicode:	«[\p{P}\p{S}]»
Shorthand:	
Java:	«\p{Punct}»
POSIX:	«[:space:]»
Description:	All whitespace characters, including line breaks
ASCII:	«[\t\r\n\v\f]»
Unicode:	«[\p{Z}\t\r\n\v\f]»
Shorthand:	«\s»
Java:	«\p{Space}»
POSIX:	«[:upper:]»
Description:	Uppercase letters
ASCII:	«[A-Z]»
Unicode:	«\p{Lu}»
Shorthand:	
Java:	«\p{Upper}»
POSIX:	«[:word:]»
Description:	Word characters (letters, numbers and underscores)
ASCII:	«[A-Za-z0-9_]»
Unicode:	«[\p{L}\p{N}\p{Pc}]»
Shorthand:	«\w»
Java:	
POSIX:	«[:xdigit:]»
Description:	Hexadecimal digits
ASCII:	«[A-Fa-f0-9]»
Unicode:	«[A-Fa-f0-9]»
Shorthand:	
Java:	«\p{XDigit}»

Collating Sequences

A POSIX locale can have collating sequences to describe how certain characters or groups of characters should be ordered. E.g. in Spanish, “ll” like in “tortilla” is treated as one character, and is ordered between “l” and “m” in the alphabet. You can use the collating sequence element `[.span-ll.]` inside a bracket expression to match „ll”. E.g. the regex `«torti[.span-ll.]a»` matches „tortilla”. Notice the double square brackets. One pair for the bracket expression, and one pair for the collating sequence.

I do not know of any regular expression engine that support collating sequences, other than POSIX-compliant engines part of a POSIX-compliant system.

Note that a fully POSIX-compliant regex engine will treat “ll” as a single character when the locale is set to Spanish. This means that `«torti[^x]a»` also matches „tortilla”. `«[^x]»` matches a single character that is not an “x”, which includes „ll” in the Spanish POSIX locale.

In any other regular expression engine, or in a POSIX engine not using the Spanish locale, `«torti[^x]a»` will match the misspelled word „tortila” but will not match „tortilla”, as `«[^x]»` cannot match the two characters “ll”.

Finally, note that not all regex engines claiming to implement POSIX regular expressions actually have full support for collating sequences. Sometimes, these engines use the regular expression syntax defined by POSIX, but don’t have full locale support. You may want to try the above matches to see if the engine you’re using does. E.g. Tcl’s `regexp` command supports collating sequences, but Tcl only supports the Unicode locale, which does not define any collating sequences. The result is that in Tcl, a collating sequence specifying a single character will match just that character, and all other collating sequences will result in an error.

Character Equivalents

A POSIX locale can define character equivalents that indicate that certain characters should be considered as identical for sorting. E.g. in French, accents are ignored when ordering words. “élève” comes before “être” which comes before “événement”. “é” and “ê” are all the same as “e”, but “l” comes before “t” which comes before “v”. With the locale set to French, a POSIX-compliant regular expression engine will match „e”, „é”, „è” and „ê” when you use the collating sequence `[=e=]` in the bracket expression `«[=e=]»`.

If a character does not have any equivalents, the character equivalence token simply reverts to the character itself. E.g. `«[=x=][=z=]»` is the same as `«[xz]»` in the French locale.

Like collating sequences, POSIX character equivalents are not available in any regex engine that I know of, other than those following the POSIX standard. And those that do may not have the necessary POSIX locale support. Here too Tcl’s `regexp` command supports character equivalents, but Unicode locale, the only one Tcl supports, does not define any character equivalents. This effectively means that `«[=x=]»` and `«[x]»` are exactly the same in Tcl, and will only match „x”, for any character you may try instead of “x”.

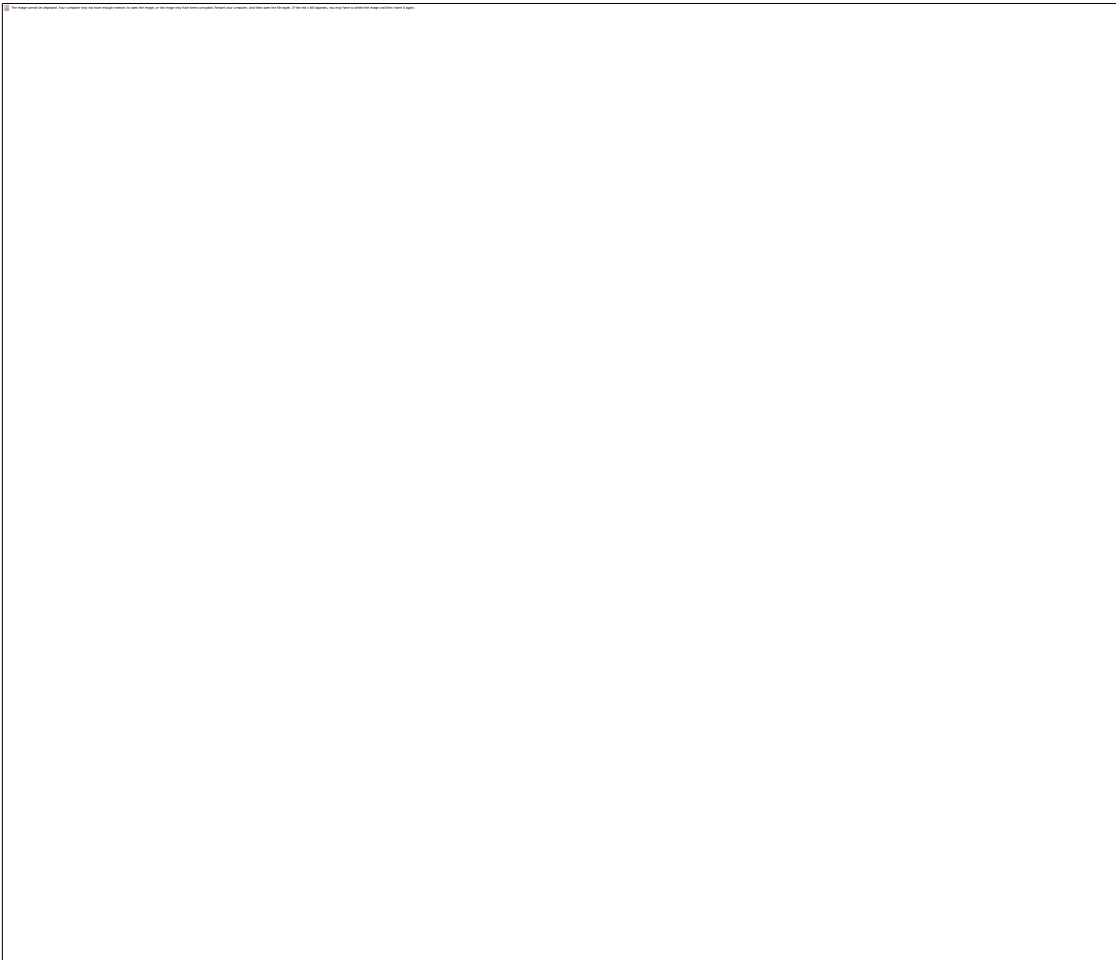
24. Adding Comments to Regular Expressions

If you have worked through the entire tutorial, I guess you will agree that regular expressions can quickly become rather cryptic. Therefore, many modern regex flavors allow you to insert comments into regexes. The syntax is «(?#comment)» where “comment” can be whatever you want, as long as it does not contain a closing round bracket. The regex engine ignores everything after the «(?#» until the first closing round bracket.

E.g. I could clarify the regex to match a valid date by writing it as «(?#year)(19|20)\d\d[- /.](?#month)(0[1-9]|1[012])[- /.](?#day)(0[1-9]|1[12][0-9]|3[01])» . Now it is instantly obvious that this regex matches a date in yyyy-mm-dd format. Some software, such as RegexBuddy, EditPad Pro and PowerGREP can apply syntax coloring to regular expressions while you write them. That makes the comments really stand out, enabling the right comment in the right spot to make a complex regular expression much easier to understand.

Regex comments are supported by the JGsoft engine, .NET, Perl, PCRE, Python and Ruby.

To make your regular expression even more readable, you can turn on free-spacing mode. All flavors that support comments also support free-spacing mode. In addition, Java supports free-spacing mode, even though it doesn't support (?#)-style comments.



25. Free-Spacing Regular Expressions

The JGsoft engine, .NET, Java, Perl, PCRE, Python, Ruby and XPath support a variant of the regular expression syntax called free-spacing mode. You can turn on this mode with the «(?x)» mode modifier, or by turning on the corresponding option in the application or passing it to the regex constructor in your programming language.

In free-spacing mode, whitespace between regular expression tokens is ignored. Whitespace includes spaces, tabs and line breaks. Note that only whitespace *between* tokens is ignored. E.g. «a b c» is the same as «abc» in free-spacing mode, but «\ d» and «\d» are not the same. The former matches „ d”, while the latter matches a digit. «\d» is a single regex token composed of a backslash and a “d”. Breaking up the token with a space gives you an escaped space (which matches a space), and a literal “d”.

Likewise, grouping modifiers cannot be broken up. «(?>atomic)» is the same as «(?> ato mic)» and as «(?>ato mic)». They all match the same atomic group. They’re not the same as (?>atomic). In fact, the latter will cause a syntax error. The ?> grouping modifier is a single element in the regex syntax, and must stay together. This is true for all such constructs, including lookahead, named groups, etc.

A character class is also treated as a single token. «[abc]» is not the same as «[a b c]». The former matches one of three letters, while the latter matches those three letters or a space. In other words: free-spacing mode has no effect inside character classes. Spaces and line breaks inside character classes will be included in the character class.

This means that in free-spacing mode, you can use «\ » or «[]» to match a single space. Use whichever you find more readable.

Java, however, does not treat a character class as a single token in free-spacing mode. Java does ignore whitespace and comments inside character classes. So in Java’s free-spacing mode, «[abc]» is identical to «[a b c]» and «\ » is the only way to match a space. However, even in free-spacing mode, the negating caret must appear immediately after the opening bracket. «[^ a b c]» matches any of the four characters „^”, „a”, „b” or „c” just like «[abc^]» would. With the negating caret in the proper place, «[^ a b c]» matches any character that is not “a”, “b” or “c”.

Comments in Free-Spacing Mode

Another feature of free-spacing mode is that the # character starts a comment. The comment runs until the end of the line. Everything from the # until the next line break character is ignored.

The XPath flavor does not support comments within the regular expression. The # is always treated as a literal character.

Putting it all together, I could clarify the regex to match a valid date by writing it across multiple lines as:

```
# Match a 20th or 21st century date in yyyy-mm-dd format
(19|20)\d\d          # year (group 1)
[- /.]              # separator
(0[1-9]|1[012])     # month (group 2)
[- /.]              # separator
(0[1-9]|1[12][0-9]|3[01]) # day (group 3)
```

Part 5

Regular Expression Examples

1. Sample Regular Expressions

Below, you will find many example patterns that you can use for and adapt to your own purposes. Key techniques used in crafting each regex are explained, with links to the corresponding pages in the tutorial where these concepts and techniques are explained in great detail.

If you are new to regular expressions, you can take a look at these examples to see what is possible. Regular expressions are very powerful. They do take some time to learn. But you will earn back that time quickly when using regular expressions to automate searching or editing tasks in EditPad Pro or PowerGREP, or when writing scripts or applications in a variety of languages.

RegexBuddy offers the fastest way to get up to speed with regular expressions. RegexBuddy will analyze any regular expression and present it to you in a clearly to understand, detailed outline. The outline links to RegexBuddy's regex tutorial (the same one you find on this website), where you can always get in-depth information with a single click.

Oh, and you definitely do not need to be a programmer to take advantage of regular expressions!

Grabbing HTML Tags

«`<TAG\b[^>]*>(.*?)</TAG>`» matches the opening and closing pair of a specific HTML tag. Anything between the tags is captured into the first backreference. The question mark in the regex makes the star lazy, to make sure it stops before the first closing tag rather than before the last, like a greedy star would do. This regex will not properly match tags nested inside themselves, like in «`<TAG>one<TAG>two</TAG>one</TAG>`».

«`(<[A-Z][A-Z0-9]*)\b[^>]*>(.*?)</\1>`» will match the opening and closing pair of any HTML tag. Be sure to turn off case sensitivity. The key in this solution is the use of the backreference «`\1`» in the regex. Anything between the tags is captured into the second backreference. This solution will also not match tags nested in themselves.

Trimming Whitespace

You can easily trim unnecessary whitespace from the start and the end of a string or the lines in a text file by doing a regex search-and-replace. Search for «`^[\t]+`» and replace with nothing to delete leading whitespace (spaces and tabs). Search for «`[\t]+$`» to trim trailing whitespace. Do both by combining the regular expressions into «`^[\t]+|[\t]+$`». Instead of `[\t]` which matches a space or a tab, you can expand the character class into «`[\t\r\n]`» if you also want to strip line breaks. Or you can use the shorthand «`\s`» instead.

IP Addresses

Matching an IP address is another good example of a trade-off between regex complexity and exactness. «`\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b`» will match any IP address just fine, but will also match

„999.999.999.999” as if it were a valid IP address. Whether this is a problem depends on the files or data you intend to apply the regex to. To restrict all 4 numbers in the IP address to 0..255, you can use this complex beast: `«\b(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.\.»«(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.»«(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.»«(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b»` (everything on a single line). The long regex stores each of the 4 numbers of the IP address into a capturing group. You can use these groups to further process the IP number.

If you don't need access to the individual numbers, you can shorten the regex with a quantifier to: `«\b(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.)\{3}\.»«(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b»`. Similarly, you can shorten the quick regex to `«\b(?:\d{1,3}\.)\{3}\d{1,3}\b»`

More Detailed Examples

Numeric Ranges. Since regular expressions work with text rather than numbers, matching specific numeric ranges requires a bit of extra care.

Matching a Floating Point Number. Also illustrates the common mistake of making everything in a regular expression optional.

Matching an Email Address. There's a lot of controversy about what is a proper regex to match email addresses. It's a perfect example showing that you need to know exactly what you're trying to match (and what not), and that there's always a trade-off between regex complexity and accuracy.

Matching Valid Dates. A regular expression that matches 31-12-1999 but not 31-13-1999.

Finding or Verifying Credit Card Numbers. Validate credit card numbers entered on your order form. Find credit card numbers in documents for a security audit.

Matching Complete Lines. Shows how to match complete lines in a text file rather than just the part of the line that satisfies a certain requirement. Also shows how to match lines in which a particular regex does *not* match.

Removing Duplicate Lines or Items. Illustrates simple yet clever use of capturing parentheses or backreferences.

Regex Examples for Processing Source Code. How to match common programming language syntax such as comments, strings, numbers, etc.

Two Words Near Each Other. Shows how to use a regular expression to emulate the “near” operator that some tools have.

Common Pitfalls

Catastrophic Backtracking. If your regular expression seems to take forever, or simply crashes your application, it has likely contracted a case of catastrophic backtracking. The solution is usually to be more

specific about what you want to match, so the number of matches the engine has to try doesn't rise exponentially.

Making Everything Optional. If all the parts in your regex are optional, it will match a zero-width string anywhere. Your regex will need to express the facts that different parts are optional depending on which parts are present.

Repeating a Capturing Group vs. Capturing a Repeated Group. Repeating a capturing group will capture only the last iteration of the group. Capture a repeated group if you want to capture all iterations.

Mixing Unicode and 8-bit Character Codes. Using 8-bit character codes like «\x80» with a Unicode engine and subject string may give unexpected results.

2. Matching Floating Point Numbers with a Regular Expression

In this example, I will show you how you can avoid a common mistake often made by people inexperienced with regular expressions. As an example, we will try to build a regular expression that can match any floating point number. Our regex should also match integers, and floating point numbers where the integer part is not given (i.e. zero). We will not try to match numbers with an exponent, such as 1.5e8 (150 million in scientific notation).

At first thought, the following regex seems to do the trick: `«[-+]?[0-9]*\.?[0-9]*»`. This defines a floating point number as an optional sign, followed by an optional series of digits (integer part), followed by an optional dot, followed by another optional series of digits (fraction part).

Spelling out the regex in words makes it obvious: everything in this regular expression is optional. This regular expression will consider a sign by itself or a dot by itself as a valid floating point number. In fact, it will even consider an empty string as a valid floating point number. This regular expression can cause serious trouble if it is used in a scripting language like Perl or PHP to verify user input.

Not escaping the dot is also a common mistake. A dot that is not escaped will match any character, including a dot. If we had not escaped the dot, “4.4” would be considered a floating point number, and “4X4” too.

When creating a regular expression, it is more important to consider what it should *not* match, than what it should. The above regex will indeed match a proper floating point number, because the regex engine is greedy. But it will also match many things we do not want, which we have to exclude.

Here is a better attempt: `«[-+]?([0-9]*\.[0-9]+|[0-9]+)»`. This regular expression will match an optional sign, that is either followed by zero or more digits followed by a dot and one or more digits (a floating point number with optional integer part), or followed by one or more digits (an integer).

This is a far better definition. Any match will include at least one digit, because there is no way around the `«[0-9]+»` part. We have successfully excluded the matches we do not want: those without digits.

We can optimize this regular expression as: `«[-+]?[0-9]*\.[0-9]+»`.

If you also want to match numbers with exponents, you can use: `«[-+]?[0-9]*\.[0-9]+([eE][-+]?[0-9]+)?»`. Notice how I made the entire exponent part optional by grouping it together, rather than making each element in the exponent optional.

Finally, if you want to validate if a particular string holds a floating point number, rather than finding a floating point number within longer text, you’ll have to anchor your regex: `«^[-+]?[0-9]*\.[0-9]+$»` or `«^[-+]?[0-9]*\.[0-9]+([eE][-+]?[0-9]+)?$»`. You can find additional variations of these regexes in RegxBuddy’s library.

3. How to Find or Validate an Email Address

The regular expression I receive the most feedback, not to mention “bug” reports on, is the one you’ll find right in the tutorial’s introduction: `«\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b»`. This regular expression, I claim, matches any email address. Most of the feedback I get refutes that claim by showing one email address that this regex doesn’t match. Usually, the “bug” report also includes a suggestion to make the regex “perfect”.

As I explain below, my claim only holds true when one accepts my definition of what a valid email address really is, and what it’s not. If you want to use a different definition, you’ll have to adapt the regex. Matching a valid email address is a perfect example showing that (1) before writing a regex, you have to know exactly what you’re trying to match, and what not; and (2) there’s often a trade-off between what’s exact, and what’s practical.

The virtue of my regular expression above is that it matches 99% of the email addresses in use today. All the email address it matches can be handled by 99% of all email software out there. If you’re looking for a quick solution, you only need to read the next paragraph. If you want to know all the trade-offs and get plenty of alternatives to choose from, read on.

If you want to use the regular expression above, there’s two things you need to understand. First, long regexes make it difficult to nicely format paragraphs. So I didn’t include `«a-z»` in any of the three character classes. This regex is intended to be used with your regex engine’s “case insensitive” option turned on. (You’d be surprised how many “bug” reports I get about that.) Second, the above regex is delimited with word boundaries, which makes it suitable for extracting email addresses from files or larger blocks of text. If you want to check whether the user typed in a valid email address, replace the word boundaries with start-of-string and end-of-string anchors, like this: `«^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$»`.

The previous paragraph also applies to all following examples. You may need to change word boundaries into start/end-of-string anchors, or vice versa. And you will need to turn on the case insensitive matching option.

Trade-Offs in Validating Email Addresses

Yes, there are a whole bunch of email addresses that my pet regex doesn’t match. The most frequently quoted example are addresses on the `.museum` top level domain, which is longer than the 4 letters my regex allows for the top level domain. I accept this trade-off because the number of people using `.museum` email addresses is extremely low. I’ve never had a complaint that the order forms or newsletter subscription forms on the JGsoft websites refused a `.museum` address (which they would, since they use the above regex to validate the email address).

To include `.museum`, you could use `«^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,6}$»`. However, then there’s another trade-off. This regex will match `„john@mail.office”`. It’s far more likely that John forgot to type in the `.com` top level domain rather than having just created a new `.office` top level domain without ICANN’s permission.

This shows another trade-off: do you want the regex to check if the top level domain exists? My regex doesn’t. Any combination of two to four letters will do, which covers all existing and planned top level domains except `.museum`. But it will match addresses with invalid top-level domains like

„asdf@asdf.asdf”. By not being overly strict about the top-level domain, I don’t have to update the regex each time a new top-level domain is created, whether it’s a country code or generic domain.

«`^[A-Z0-9._%+~]+@[A-Z0-9.-]+\.(?:[A-Z]{2}|com|org|net|edu|gov|mil|biz|info|mobi|name|aero|asia|jobs|museum)$`» could be used to allow any two-letter country code top level domain, and only specific generic top level domains. By the time you read this, the list might already be out of date. If you use this regular expression, I recommend you store it in a global constant in your application, so you only have to update it in one place. You could list all country codes in the same manner, even though there are almost 200 of them.

Email addresses can be on servers on a subdomain, e.g. „john@server.department.company.com”. All of the above regexes will match this email address, because I included a dot in the character class after the @ symbol. However, the above regexes will also match „john@aol...com” which is not valid due to the consecutive dots. You can exclude such matches by replacing «`[A-Z0-9.-]+\.`» with «`(?:[A-Z0-9-]+\.)+`» in any of the above regexes. I removed the dot from the character class and instead repeated the character class and the following literal dot. E.g. «`\b[A-Z0-9._%+~]+@(?:[A-Z0-9-]+\.)+[A-Z]{2,4}\b`» will match „john@server.department.company.com” but not “john@aol...com”.

Another trade-off is that my regex only allows English letters, digits and a few special symbols. The main reason is that I don’t trust all my email software to be able to handle much else. Even though John.O'Hara@theoharas.com is a syntactically valid email address, there’s a risk that some software will misinterpret the apostrophe as a delimiting quote. E.g. blindly inserting this email address into a SQL will cause it to fail if strings are delimited with single quotes. And of course, it’s been many years already that domain names can include non-English characters. Most software and even domain name registrars, however, still stick to the 37 characters they’re used to.

The conclusion is that to decide which regular expression to use, whether you’re trying to match an email address or something else that’s vaguely defined, you need to start with considering all the trade-offs. How bad is it to match something that’s not valid? How bad is it not to match something that is valid? How complex can your regular expression be? How expensive would it be if you had to change the regular expression later? Different answers to these questions will require a different regular expression as the solution. My email regex does what I want, but it may not do what you want.

Regexes Don’t Send Email

Don’t go overboard in trying to eliminate invalid email addresses with your regular expression. If you have to accept .museum domains, allowing any 6-letter top level domain is often better than spelling out a list of all current domains. The reason is that you don’t really know whether an address is valid until you try to send an email to it. And even that might not be enough. Even if the email arrives in a mailbox, that doesn’t mean somebody still reads that mailbox.

The same principle applies in many situations. When trying to match a valid date, it’s often easier to use a bit of arithmetic to check for leap years, rather than trying to do it in a regex. Use a regular expression to find potential matches or check if the input uses the proper syntax, and do the actual validation on the potential matches returned by the regular expression. Regular expressions are a powerful tool, but they’re far from a panacea.

The Official Standard: RFC 2822

Maybe you're wondering why there's no "official" fool-proof regex to match email addresses. Well, there is an official definition, but it's hardly fool-proof.

The official standard is known as RFC 2822. It describes the syntax that valid email addresses must adhere to. You can (but you shouldn't--read on) implement it with this regular expression:

```
«(?:[a-z0-9!#$%&'*/+=?^_`{|}~-]+(?:\. [a-z0-9!#$%&'*/+=?^_`{|}~-]+)*|"(?:[\x01-
\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\[\x01-\x09\x0b\x0c\x0e-
\x7f])*)@(?:([a-z0-9]([a-z0-9]*[a-z0-9])?\.)+[a-z0-9]([a-z0-9]*[a-z0-
9])?)|\[(?:([0-9]([0-9]*[0-9])?)|([a-z0-9]([a-z0-9]*[a-z0-9])?:([a-z0-9]([a-z0-9]*[a-z0-9])?)|
[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f])\])\]»
```

This regex has two parts: the part before the @, and the part after the @. There are two alternatives for the part before the @: it can either consist of a series of letters, digits and certain symbols, including one or more dots. However, dots may not appear consecutively or at the start or end of the email address. The other alternative requires the part before the @ to be enclosed in double quotes, allowing any string of ASCII characters between the quotes. Whitespace characters, double quotes and backslashes must be escaped with backslashes.

The part after the @ also has two alternatives. It can either be a fully qualified domain name (e.g. regular-expressions.info), or it can be a literal Internet address between square brackets. The literal Internet address can either be an IP address, or a domain-specific routing address.

The reason you shouldn't use this regex is that it only checks the basic syntax of email addresses. john@aol.com.nospam would be considered a valid email address according to RFC 2822. Obviously, this email address won't work, since there's no "nospam" top-level domain. It also doesn't guarantee your email software will be able to handle it. Not all applications support the syntax using double quotes or square brackets. In fact, RFC 2822 itself marks the notation using square brackets as obsolete.

We get a more practical implementation of RFC 2822 if we omit the syntax using double quotes and square brackets. It will still match 99.99% of all email addresses in actual use today.

```
«[a-z0-9!#$%&'*/+=?^_`{|}~-]+(?:\. [a-z0-9!#$%&'*/+=?^_`{|}~-]+)*@(?:[a-z0-9]([a-z0-9]*[a-z0-9])?\.)+[a-z0-9]([a-z0-9]*[a-z0-9])?»
```

A further change you could make is to allow any two-letter country code top level domain, and only specific generic top level domains. This regex filters dummy email addresses like asdf@adsf.adsf. You will need to update it as new top-level domains are added.

```
«[a-z0-9!#$%&'*/+=?^_`{|}~-]+(?:\. [a-z0-9!#$%&'*/+=?^_`{|}~-]+)*@(?:[a-z0-9]([a-z0-9]*[a-z0-9])?\.)+(?:[A-Z]{2}|com|org|net|edu|gov|mil|biz|info|mobi|name|aero|asia|jobs|museum)\b»
```

So even when following official standards, there are still trade-offs to be made. Don't blindly copy regular expressions from online libraries or discussion forums. Always test them on your own data and with your own applications.

4. Matching a Valid Date

«`^(19|20)\d\d[- /.](0[1-9]|1[012])[- /.](0[1-9]|12)[0-9]|3[01])$`» matches a date in yyyy-mm-dd format from between 1900-01-01 and 2099-12-31, with a choice of four separators. The anchors make sure the entire variable is a date, and not a piece of text containing a date. The year is matched by «`(19|20)\d\d`». I used alternation to allow the first two digits to be 19 or 20. The round brackets are mandatory. Had I omitted them, the regex engine would go looking for 19 or the remainder of the regular expression, which matches a date between 2000-01-01 and 2099-12-31. Round brackets are the only way to stop the vertical bar from splitting up the entire regular expression into two options.

The month is matched by «`0[1-9]|1[012]`», again enclosed by round brackets to keep the two options together. By using character classes, the first option matches a number between 01 and 09, and the second matches 10, 11 or 12.

The last part of the regex consists of three options. The first matches the numbers 01 through 09, the second 10 through 29, and the third matches 30 or 31.

Smart use of alternation allows us to exclude invalid dates such as 2000-00-00 that could not have been excluded without using alternation. To be really perfectionist, you would have to split up the month into various options to take into account the length of the month. The above regex still matches 2003-02-31, which is not a valid date. Making leading zeros optional could be another enhancement.

If you want to require the delimiters to be consistent, you could use a backreference. «`^(19|20)\d\d([- /.])(0[1-9]|1[012])\2(0[1-9]|12)[0-9]|3[01])$`» will match „1999-01-01” but not “1999/01-01”.

Again, how complex you want to make your regular expression depends on the data you are using it on, and how big a problem it is if an unwanted match slips through. If you are validating the user’s input of a date in a script, it is probably easier to do certain checks outside of the regex. For example, excluding February 29th when the year is not a leap year is far easier to do in a scripting language. It is far easier to check if a year is divisible by 4 (and not divisible by 100 unless divisible by 400) using simple arithmetic than using regular expressions.

Here is how you could check a valid date in Perl. I also added round brackets to capture the year into a backreference.

```
sub isvaliddate {
    my $input = shift;
    if ($input =~ m!^(?:19|20)\d\d([- /.])(0[1-9]|1[012])\2(0[1-9]|12)[0-9]|3[01])$!) {
        # At this point, $1 holds the year, $2 the month and $3 the day of the date entered
        if ($3 == 31 and ($2 == 4 or $2 == 6 or $2 == 9 or $2 == 11)) {
            return 0; # 31st of a month with 30 days
        } elsif ($3 >= 30 and $2 == 2) {
            return 0; # February 30th or 31st
        } elsif ($2 == 2 and $3 == 29 and not ($1 % 4 == 0 and ($1 % 100 != 0 or $1 % 400 == 0))) {
            return 0; # February 29th outside a leap year
        } else {
            return 1; # Valid date
        }
    } else {
        return 0; # Not a date
    }
}
```

To match a date in mm/dd/yyyy format, rearrange the regular expression to «`^(0[1-9]|1[012])[- /.](0[1-9]|12)[0-9]|3[01])[- /.](19|20)\d\d$`». For dd-mm-yyyy format, use «`^(0[1-9]|12)[0-9]|3[01])[- /.](0[1-9]|1[012])[- /.](19|20)\d\d$`». You can find additional variations of these regexes in RegexBuddy's library.

5. Finding or Verifying Credit Card Numbers

With a few simple regular expressions, you can easily verify whether your customer entered a valid credit card number on your order form. You can even determine the type of credit card being used. Each card issuer has its own range of card numbers, identified by the first 4 digits.

You can use a slightly different regular expression to find credit card numbers, or number sequences that might be credit card numbers, within larger documents. This can be very useful to prove in a security audit that you're not improperly exposing your clients' financial details.

We'll start with the order form.

Stripping Spaces and Dashes

The first step is to remove all non-digits from the card number entered by the customer. Physical credit cards have spaces within the card number to group the digits, making it easier for humans to read or type in. So your order form should accept card numbers with spaces or dashes in them.

To remove all non-digits from the card number, simply use the “replace all” function in your scripting language to search for the regex «`^[^0-9]+`» and replace it with nothing. If you only want to replace spaces and dashes, you could use «`[-]+`». If this regex looks odd, remember that in a character class, the hyphen is a literal when it occurs right before the closing bracket (or right after the opening bracket or negating caret).

If you're wondering what the plus is for: that's for performance. If the input has consecutive non-digits, e.g. “1===2”, then the regex will match the three equals signs at once, and delete them in one replacement. Without the plus, three replacements would be required. In this case, the savings are only a few microseconds. But it's a good habit to keep regex efficiency in the back of your mind. Though the savings are minimal here, so is the effort of typing the extra plus.

Validating Credit Card Numbers on Your Order Form

Validating credit card numbers is the ideal job for regular expressions. They're just a sequence of 13 to 16 digits, with a few specific digits at the start that identify the card issuer. You can use the specific regular expressions below to alert customers when they try to use a kind of card you don't accept, or to route orders using different cards to different processors. All these regexes were taken from RegexBuddy's library.

- Visa: «`^4[0-9]{12}(?:[0-9]{3})?$`» All Visa card numbers start with a 4. New cards have 16 digits. Old cards have 13.
- MasterCard: «`^5[1-5][0-9]{14}$`» All MasterCard numbers start with the numbers 51 through 55. All have 16 digits.
- American Express: «`^3[47][0-9]{13}$`» American Express card numbers start with 34 or 37 and have 15 digits.
- Diners Club: «`^3(?:0[0-5]|[68][0-9])[0-9]{11}$`» Diners Club card numbers begin with 300 through 305, 36 or 38. All have 14 digits. There are Diners Club cards that begin with 5 and have 16 digits. These are a joint venture between Diners Club and MasterCard, and should be processed like a MasterCard.

- Discover: «`^6(?:011|5[0-9]{2})[0-9]{12}$`» Discover card numbers begin with 6011 or 65. All have 16 digits.
- JCB: «`(?:2131|1800|35\d{3})\d{11}$`» JCB cards beginning with 2131 or 1800 have 15 digits. JCB cards beginning with 35 have 16 digits.

If you just want to check whether the card number looks valid, without determining the brand, you can combine the above six regexes into «`(?:4[0-9]{12}(?:[0-9]{3})?|5[1-5][0-9]{14}|6(?:011|5[0-9][0-9])[0-9]{12}|3[47][0-9]{13}|3(?:0[0-5]|[68][0-9])[0-9]{11}|(?:2131|1800|35\d{3})\d{11})$`». You'll see I've simply alternated all the regexes, and used a non-capturing group to put the anchors outside the alternation. You can easily delete the card types you don't accept from the list.

These regular expressions will easily catch numbers that are invalid because the customer entered too many or too few digits. They won't catch numbers with incorrect digits. For that, you need to follow the Luhn algorithm, which cannot be done with a regex. And of course, even if the number is mathematically valid, that doesn't mean a card with this number was issued or if there's money in the account. The benefit of the regular expression is that you can put it in a bit of JavaScript to instantly check for obvious errors, instead of making the customer wait 30 seconds for your credit card processor to fail the order. And if your card processor charges for failed transactions, you'll really want to implement both the regex and the Luhn validation.

Finding Credit Card Numbers in Documents

With two simple modifications, you could use any of the above regexes to find card numbers in larger documents. Simply replace the caret and dollar with a word boundary, e.g.: «`\b4[0-9]{12}(?:[0-9]{3})?\b`».

If you're planning to search a large document server, a simpler regular expression will speed up the search. Unless your company uses 16-digit numbers for other purposes, you'll have few false positives. The regex «`\b\d{13,16}\b`» will find any sequence of 13 to 16 digits.

When searching a hard disk full of files, you can't strip out spaces and dashes first like you can when validating a single card number. To find card numbers with spaces or dashes in them, use «`\b(?:\d[-]*?)\{13,16\}\b`». This regex allows any amount of spaces and dashes anywhere in the number. This is really the only way. Visa and MasterCard put digits in sets of 4, while Amex and Discover use groups of 4, 5 and 6 digits. People typing in the numbers may have different ideas yet.

6. Matching Whole Lines of Text

Often, you want to match complete lines in a text file rather than just the part of the line that satisfies a certain requirement. This is useful if you want to delete entire lines in a search-and-replace in a text editor, or collect entire lines in an information retrieval tool. To keep this example simple, let's say we want to match lines containing the word "John". The regex «`John`» makes it easy enough to locate those lines. But the software will only indicate „John” as the match, not the entire line containing the word.

The solution is fairly simple. To specify that we need an entire line, we will use the caret and dollar sign and turn on the option to make them match at embedded newlines. In software aimed at working with text files like EditPad Pro and PowerGREP, the anchors always match at embedded newlines. To match the parts of the line before and after the match of our original regular expression «`John`», we simply use the dot and the star. Be sure to turn *off* the option for the dot to match newlines.

The resulting regex is: «`^.*John.*$`». You can use the same method to expand the match of any regular expression to an entire line, or a block of complete lines. In some cases, such as when using alternation, you will need to group the original regex together using round brackets.

Finding Lines Containing or Not Containing Certain Words

If a line can meet any out of series of requirements, simply use alternation in the regular expression. «`^.*\b(one|two|three)\b.*$`» matches a complete line of text that contains any of the words “one”, “two” or “three”. The first backreference will contain the word the line actually contains. If it contains more than one of the words, then the last (rightmost) word will be captured into the first backreference. This is because the star is greedy. If we make the first star lazy, like in «`^.*?\b(one|two|three)\b.*$`», then the backreference will contain the first (leftmost) word.

If a line must satisfy all of multiple requirements, we need to use lookahead. «`^(?=.*?\bone\b)(?=.*?\btwo\b)(?=.*?\bthree\b).*$`» matches a complete line of text that contains *all* of the words “one”, “two” and “three”. Again, the anchors must match at the start and end of a line and the dot must not match line breaks. Because of the caret, and the fact that lookahead is zero-width, all of the three lookaheads are attempted at the start of the each line. Each lookahead will match any piece of text on a single line («`.*?`») followed by one of the words. All three must match successfully for the entire regex to match. Note that instead of words like «`\bword\b`», you can put any regular expression, no matter how complex, inside the lookahead. Finally, «`.*$`» causes the regex to actually match the line, after the lookaheads have determined it meets the requirements.

If your condition is that a line should *not* contain something, use negative lookahead. «`^(?!regexp).*$`» matches a complete line that does *not* match «`regexp`». Notice that unlike before, when using positive lookahead, I repeated both the negative lookahead and the dot together. For the positive lookahead, we only need to find one location where it can match. But the negative lookahead must be tested at each and every character position in the line. We must test that «`regexp`» fails everywhere, not just somewhere.

Finally, you can combine multiple positive and negative requirements as follows: «`^(?=.*?\bmust-have\b)(?=.*?\bmandatory\b)(?!avoid|illegal).*$`». When checking multiple positive requirements, the «`.*`» at the end of the regular expression full of zero-width assertions made sure that we actually matched something. Since the negative requirement must match the entire line, it is easy to replace the «`.*`» with the negative test.

7. Deleting Duplicate Lines From a File

If you have a file in which all lines are sorted (alphabetically or otherwise), you can easily delete (consecutive) duplicate lines. Simply open the file in your favorite text editor, and do a search-and-replace searching for `«^(. *) (\r?\n\1)+$»` matches a single-line string that does not allow the quote character to appear inside the string. Using the negated character class is more efficient than using a lazy dot. `«"[^"]*"»` allows the string to span across multiple lines.

`«"[\r\n]*(?:\\.[^\r\n]*)*»` matches a single-line string in which the quote character can appear if it is escaped by a backslash. Though this regular expression may seem more complicated than it needs to be, it is much faster than simpler solutions which can cause a whole lot of backtracking in case a double quote appears somewhere all by itself rather than part of a string. `«"[\r\n]*(?:\\.[^"]*)*»` allows the string to span multiple lines.

You can adapt the above regexes to match any sequence delimited by two (possibly different) characters. If we use “b” for the starting character, “e” and the end, and “x” as the escape character, the version without escape becomes `«b[^e\r\n]*e»`, and the version with escape becomes `«b[^\ex\r\n]*(?:x.[^\ex\r\n]*)*e»`.

Numbers

`«\b\d+\b»` matches a positive integer number. Do not forget the word boundaries! `«[-+]? \b\d+\b»` allows for a sign.

`«\b0[xX][0-9a-fA-F]+\b»` matches a C-style hexadecimal number.

`«((\b[0-9]+)?\.)?[0-9]+\b»` matches an integer number as well as a floating point number with optional integer part. `«(\b[0-9]+\.\.([0-9]+\b)?|\.[0-9]+\b)»` matches a floating point number with optional integer as well as optional fractional part, but does not match an integer number.

`«((\b[0-9]+)?\.)?\b[0-9]+([eE][-+]?[0-9]+)?\b»` matches a number in scientific notation. The mantissa can be an integer or floating point number with optional integer part. The exponent is optional.

`«\b[0-9]+(\.[0-9]+)?(e[+-]?[0-9]+)?\b»` also matches a number in scientific notation. The difference with the previous example is that if the mantissa is a floating point number, the integer part is mandatory.

If you read through the floating point number example, you will notice that the above regexes are different from what is used there. The above regexes are more stringent. They use word boundaries to exclude numbers that are part of other things like identifiers. You can prepend `«[-+]?»` to all of the above regexes to include an optional sign in the regex. I did not do so above because in programming languages, the + and - are usually considered operators rather than signs.

Reserved Words or Keywords

Matching reserved words is easy. Simply use alternation to string them together: `«\b(first|second|third|etc)\b»` Again, do not forget the word boundaries.

9. Find Two Words Near Each Other

Some search tools that use boolean operators also have a special operator called "near". Searching for "term1 near term2" finds all occurrences of term1 and term2 that occur within a certain "distance" from each other. The distance is a number of words. The actual number depends on the search tool, and is often configurable.

You can easily perform the same task with the proper regular expression.

Emulating "near" with a Regular Expression

With regular expressions you can describe almost any text pattern, including a pattern that matches two words near each other. This pattern is relatively simple, consisting of three parts: the first word, a certain number of unspecified words, and the second word. An unspecified word can be matched with the shorthand character class «\w+». The spaces and other characters between the words can be matched with «\W+» (uppercase W this time).

The complete regular expression becomes «\bword1\W+(?:\w+\W+){1,6}?word2\b». The quantifier «{1,6}?» makes the regex require at least one word between "word1" and "word2", and allow at most six words.

If the words may also occur in reverse order, we need to specify the opposite pattern as well: «\b(?:word1\W+(?:\w+\W+){1,6}?word2|word2\W+(?:\w+\W+){1,6}?word1)\b»

If you want to find any pair of two words out of a list of words, you can use: «\b(word1|word2|word3)(?:\W+\w+){1,6}?\W+(word1|word2|word3)\b». This regex will also find a word near itself, e.g. it will match „word2 near word2”.

10. Runaway Regular Expressions: Catastrophic Backtracking

Consider the regular expression `«(x+x+)+y»`. Before you scream in horror and say this contrived example should be written as `«xx+y»` to match exactly the same without those terribly nested quantifiers: just assume that each “x” represents something more complex, with certain strings being matched by both “x”. See the section on HTML files below for a real example.

Let’s see what happens when you apply this regex to “xxxxxxxxxy”. The first `«x+»` will match all 10 „x” characters. The second `«x+»` fails. The first `«x+»` then backtracks to 9 matches, and the second one picks up the remaining „x”. The group has now matched once. The group repeats, but fails at the first `«x+»`. Since one repetition was sufficient, the group matches. `«y»` matches „y” and an overall match is found. The regex is declared functional, the code is shipped to the customer, and his computer explodes. Almost.

The above regex turns ugly when the “y” is missing from the subject string. When `«y»` fails, the regex engine backtracks. The group has one iteration it can backtrack into. The second `«x+»` matched only one „x”, so it can’t backtrack. But the first `«x+»` can give up one “x”. The second `«x+»` promptly matches „xx”. The group again has one iteration, fails the next one, and the `«y»` fails. Backtracking again, the second `«x+»` now has one backtracking position, reducing itself to match „x”. The group tries a second iteration. The first `«x+»` matches but the second is stuck at the end of the string. Backtracking again, the first `«x+»` in the group’s first iteration reduces itself to 7 characters. The second `«x+»` matches „xxx”. Failing `«y»`, the second `«x+»` is reduced to „xx” and then „x”. Now, the group can match a second iteration, with one „x” for each `«x+»`. But this (7,1),(1,1) combination fails too. So it goes to (6,4) and then (6,2)(1,1) and then (6,1),(2,1) and then (6,1),(1,2) and then I think you start to get the drift.

If you try this regex on a 10x string in RegexBuddy’s debugger, it’ll take 2558 steps to figure out the final `«y»` is missing. For an 11x string, it needs 5118 steps. For 12, it takes 10238 steps. Clearly we have an exponential complexity of $O(2^n)$ here. At 21x the debugger bows out at 2.8 million steps, diagnosing a bad case of catastrophic backtracking.

RegexBuddy is forgiving in that it detects it’s going in circles, and aborts the match attempt. Other regex engines (like .NET) will keep going forever, while others will crash with a stack overflow (like Perl, before version 5.10). Stack overflows are particularly nasty on Windows, since they tend to make your application vanish without a trace or explanation. Be very careful if you run a web service that allows users to supply their own regular expressions. People with little regex experience have surprising skill at coming up with exponentially complex regular expressions.

Possessive Quantifiers and Atomic Grouping to The Rescue

In the above example, the sane thing to do is obviously to rewrite it as `«xx+y»` which eliminates the nested quantifiers entirely. Nested quantifiers are repeated or alternated tokens inside a group that is itself repeated or alternated. These almost always lead to catastrophic backtracking. About the only situation where they don’t is when the start of each alternative inside the group is not optional, and mutually exclusive with the start of all the other alternatives, and mutually exclusive with the token that follows it (inside its alternative inside the group). E.g. `«(a+b+|c+d+)+y»` is safe. If anything fails, the regex engine will backtrack through the whole regex, but it will do so linearly. The reason is that all the tokens are mutually exclusive. None of them can match any characters matched by any of the others. So the match attempt at each backtracking

position will fail, causing the regex engine to backtrack linearly. If you test this on “aaaabbbbccccdddd”, RegexBuddy needs only 13 steps rather than millions of steps to figure it out.

However, it’s not always possible or easy to rewrite your regex to make everything mutually exclusive. So we need a way to tell the regex engine not to backtrack. When we’ve grabbed all the x’s, there’s no need to backtrack. There couldn’t possibly be a “y” in anything matched by either «x+». Using a possessive quantifier, our regex becomes «(x+x+)+y». This fails the 21x string in merely 7 steps. That’s 6 steps to match all the x’s, and 1 step to figure out that «y» fails. Almost no backtracking is done. Using an atomic group, the regex becomes «(?>(x+x+)+)y» with the exact same results.

A Real Example: Matching CSV Records

Here’s a real example from a technical support case I once handled. The customer was trying to find lines in a comma-delimited text file where the 12th item on a line started with a “P”. He was using the innocently-looking regexp «^(. *?,) {11}P».

At first sight, this regex looks like it should do the job just fine. The lazy dot and comma match a single comma-delimited field, and the {11} skips the first 11 fields. Finally, the P checks if the 12th field indeed starts with P. In fact, this is exactly what will happen when the 12th field indeed starts with a P.

The problem rears its ugly head when the 12th field does not start with a P. Let’s say the string is “1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13”. At that point, the regex engine will backtrack. It will backtrack to the point where «^(. *?,) {11}» had consumed „1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11”, giving up the last match of the comma. The next token is again the dot. The dot matches a comma. *The dot matches the comma!* However, the comma does not match the “1” in the 12th field, so the dot continues until the 11th iteration of «. *?, » has consumed „11, 12, ”. You can already see the root of the problem: the part of the regex (the dot) matching the contents of the field also matches the delimiter (the comma). Because of the double repetition (star inside {11}), this leads to a catastrophic amount of backtracking.

The regex engine now checks whether the 13th field starts with a P. It does not. Since there is no comma after the 13th field, the regex engine can no longer match the 11th iteration of «. *?, ». But it does not give up there. It backtracks to the 10th iteration, expanding the match of the 10th iteration to „10, 11, ”. Since there is still no P, the 10th iteration is expanded to „10, 11, 12, ”. Reaching the end of the string again, the same story starts with the 9th iteration, subsequently expanding it to „9, 10, ”, „9, 10, 11, ”, „9, 10, 11, 12, ”. But between each expansion, there are more possibilities to be tried. When the 9th iteration consumes „9, 10, ”, the 10th could match just „11, ” as well as „11, 12, ”. Continuously failing, the engine backtracks to the 8th iteration, again trying all possible combinations for the 9th, 10th, and 11th iterations.

You get the idea: the possible number of combinations that the regex engine will try for each line where the 12th field does not start with a P is huge. All this would take a long time if you ran this regex on a large CSV file where most rows don’t have a P at the start of the 12th field.

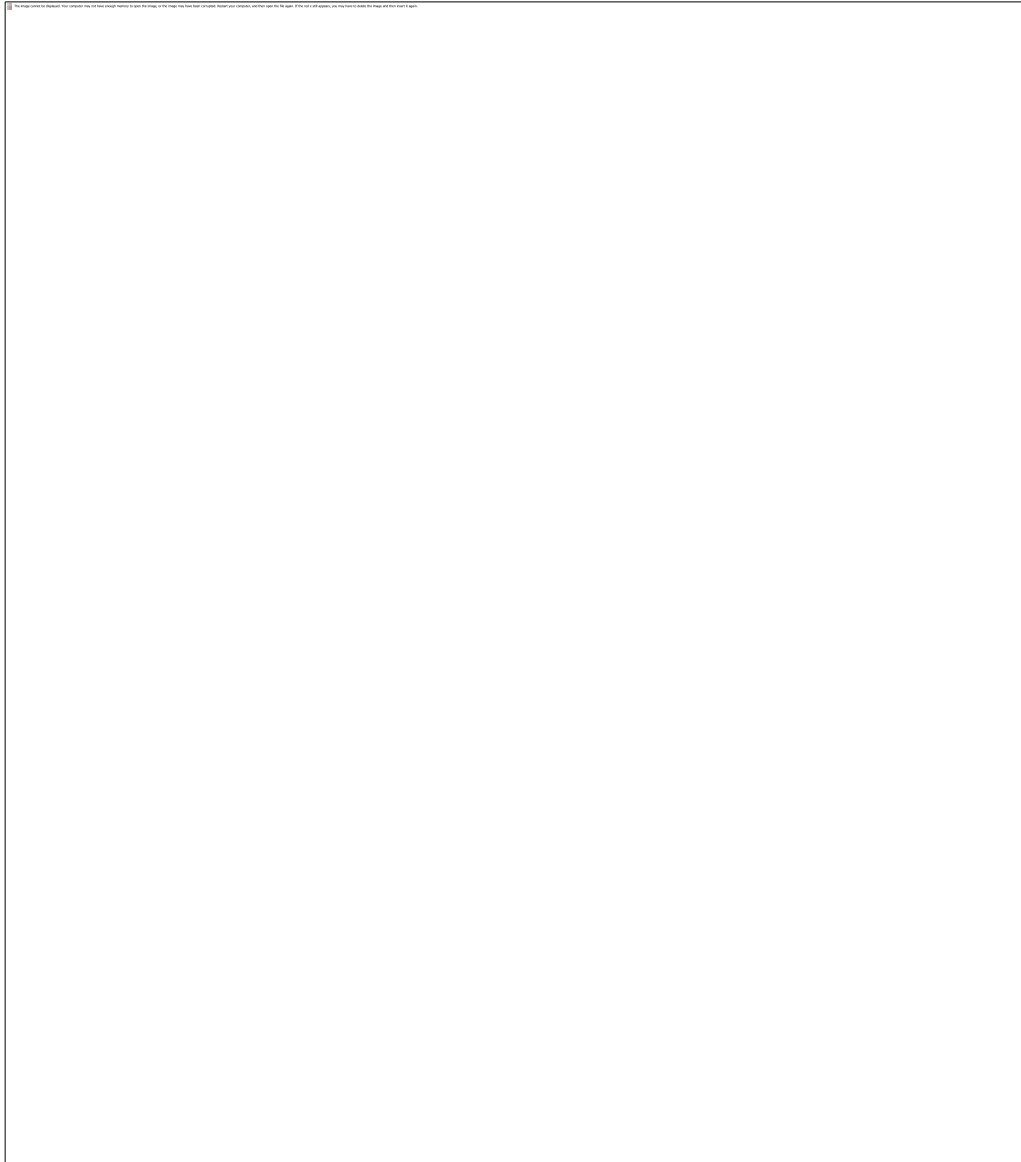
Preventing Catastrophic Backtracking

The solution is simple. When nesting repetition operators, make absolutely sure that there is only one way to match the same match. If repeating the inner loop 4 times and the outer loop 7 times results in the same overall match as repeating the inner loop 6 times and the outer loop 2 times, you can be sure that the regex engine will try all those combinations.

In our example, the solution is to be more exact about what we want to match. We want to match 11 comma-delimited fields. The fields must not contain comma's. So the regex becomes: `«^[^\r\n]*,){11}P»`. If the P cannot be found, the engine will still backtrack. But it will backtrack only 11 times, and each time the `«[^\r\n]»` is not able to expand beyond the comma, forcing the regex engine to the previous one of the 11 iterations immediately, without trying further options.

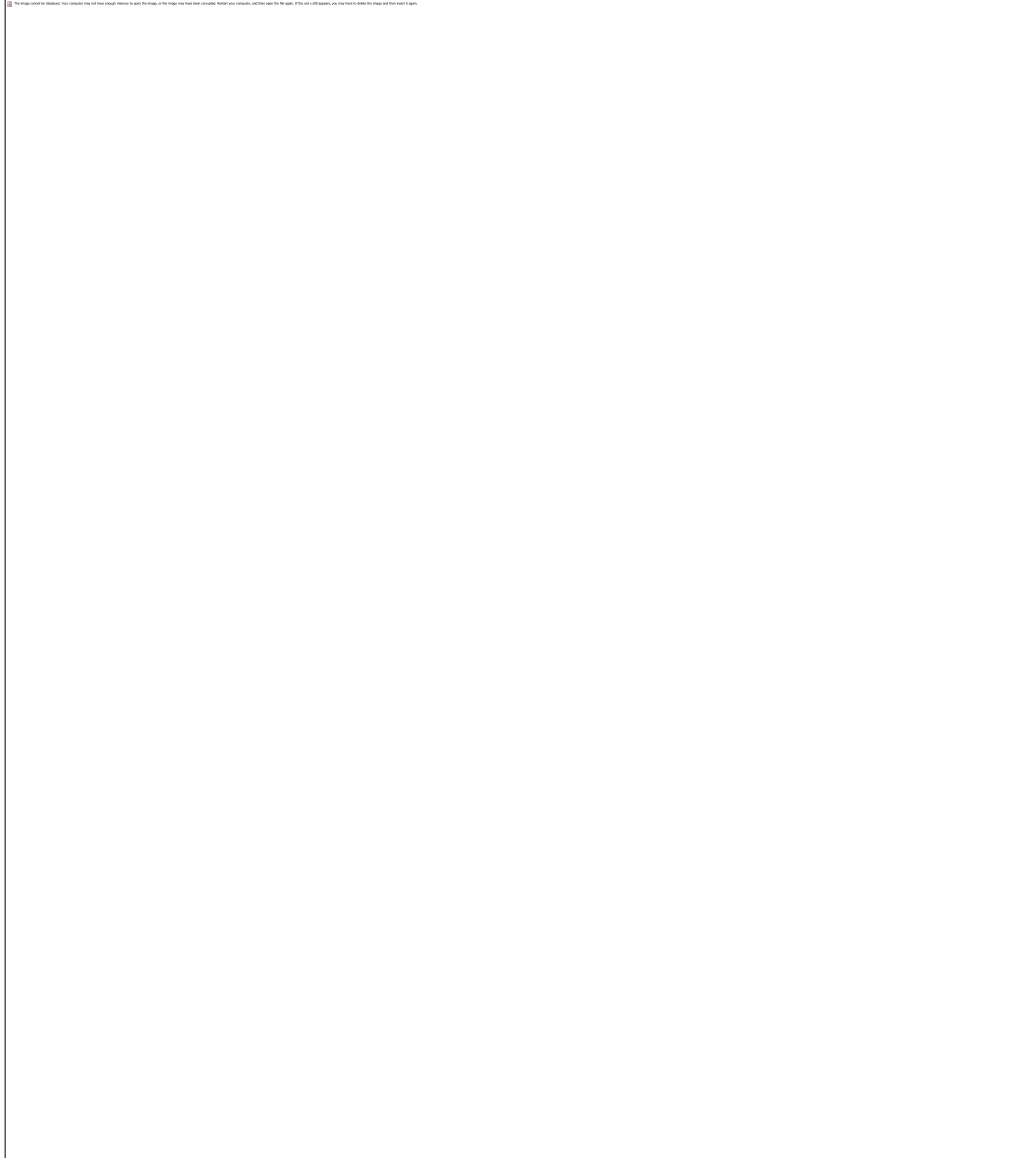
See the Difference with RegexBuddy

If you try this example with RegexBuddy's debugger, you will see that the original regex `«^(.*?,){11}P»` needs 29,685 steps to conclude there regex cannot match "1,2,3,4,5,6,7,8,9,10,11,12". If the string is "1,2,3,4,5,6,7,8,9,10,11,12,13", just 3 characters more, the number of steps doubles to 60,313. It's not too hard to imagine that at this kind of exponential rate, attempting this regex on a large file with long lines could easily take forever.



Our improved regex « $^([\^, \r \n]^*,)\{11\}P$ », however, needs just forty-eight steps to fail, whether the subject string has 12 numbers, 13 numbers, 16 numbers or a billion. While the complexity of the original regex was exponential, the complexity of the improved regex is constant with respect to whatever follows the 12th field. The reason is the regex fails immediately when it discovers the 12th field doesn't start with a P. It simply backtracks 12 times without expanding again, and that's it.

The complexity of the improved regex is linear to the length of the first 11 fields. 36 steps are needed in our example. That's the best we can do, since the engine does have to scan through all the characters of the first 11 fields to find out where the 12th one begins. Our improved regex is a perfect solution.



Alternative Solution Using Atomic Grouping

In the above example, we could easily reduce the amount of backtracking to a very low level by better specifying what we wanted. But that is not always possible in such a straightforward manner. In that case, you should use atomic grouping to prevent the regex engine from backtracking.

Using atomic grouping, the above regex becomes `«^(?>(.*?,){11})P»`. Everything between `(?>)` is treated as one single token by the regex engine, once the regex engine leaves the group. Because the entire group is one token, no backtracking can take place once the regex engine has found a match for the group. If backtracking is required, the engine has to backtrack to the regex token before the group (the caret in our example). If there is no token before the group, the regex must retry the entire regex at the next position in the string.

Let's see how `«^(?>(.*?,){11})P»` is applied to `"1,2,3,4,5,6,7,8,9,10,11,12,13"`. The caret matches at the start of the string and the engine enters the atomic group. The star is lazy, so the dot is initially skipped. But the comma does not match `"1"`, so the engine backtracks to the dot. That's right: backtracking is allowed here. The star is not possessive, and is not immediately enclosed by an atomic group. That is, the regex engine did not cross the closing round bracket of the atomic group. The dot matches `„1"`, and the comma matches too. `«{11}»` causes further repetition until the atomic group has matched `„1,2,3,4,5,6,7,8,9,10,11,„`.

Now, the engine leaves the atomic group. Because the group is atomic, all backtracking information is discarded and the group is now considered a single token. The engine now tries to match `«P»` to the `"1"` in the 12th field. This fails.

So far, everything happened just like in the original, troublesome regular expression. Now comes the difference. `«P»` failed to match, so the engine backtracks. The previous token is an atomic group, so the group's entire match is discarded and the engine backtracks further to the caret. The engine now tries to match the caret at the next position in the string, which fails. The engine walks through the string until the end, and declares failure. Failure is declared after 30 attempts to match the caret, and just one attempt to match the atomic group, rather than after 30 attempts to match the caret and a huge number of attempts to try all combinations of both quantifiers in the regex.

That is what atomic grouping and possessive quantifiers are for: efficiency by disallowing backtracking. The most efficient regex for our problem at hand would be `«^(?>((?>[^\r\n]*)){11})P»`, since possessive, greedy repetition of the star is faster than a backtracking lazy dot. If possessive quantifiers are available, you can reduce clutter by writing `«^(?>([^\r\n]*+){11})P»`.

Quickly Matching a Complete HTML File

Another common situation where catastrophic backtracking occurs is when trying to match `"something"` followed by `"anything"` followed by `"another something"` followed by `"anything"`, where the lazy dot `«. *?»` is used. The more `"anything"`, the more backtracking. Sometimes, the lazy dot is simply a symptom of a lazy programmer. `«. *?»` is not appropriate to match a double-quoted string, since you don't really want to allow anything between the quotes. A string can't have (unescaped) embedded quotes, so `«"[^\r\n]*"»` is more appropriate, and won't lead to catastrophic backtracking when combined in a larger regular expression. However, sometimes `"anything"` really is just that. The problem is that `"another something"` also qualifies as `"anything"`, giving us a genuine `«x+x+»` situation.

Suppose you want to use a regular expression to match a complete HTML file, and extract the basic parts from the file. If you know the structure of HTML files, writing the regex «<html>.*?<head>.*?<title>.*?</title>.*?</head>.*?<body[^>]*>.*?</body>.*?</html>» is very straight-forward. With the “dot matches newlines” or “single line” matching mode turned on, it will work just fine on valid HTML files.

Unfortunately, this regular expression won’t work nearly as well on an HTML file that misses some of the tags. The worst case is a missing </html> tag at the end of the file. When «</html>» fails to match, the regex engine backtracks, giving up the match for «</body>.*?». It will then further expand the lazy dot before «</body>», looking for a second closing “</body>” tag in the HTML file. When that fails, the engine gives up «<body[^>]*>.*?», and starts looking for a second opening “<body[^>]*>” tag all the way to the end of the file. Since that also fails, the engine proceeds looking all the way to the end of the file for a second closing head tag, a second closing title tag, etc.

If you run this regex in RegexBuddy’s debugger, the output will look like a sawtooth. The regex matches the whole file, backs up a little, matches the whole file again, backs up some more, backs up yet some more, matches everything again, etc. until each of the 7 «.*?» tokens has reached the end of the file. The result is that this regular has a worst case complexity of N^7 . If you double the length of the HTML file with the missing <html> tag by appending text at the end, the regular expression will take 128 times (2^7) as long to figure out the HTML file isn’t valid. This isn’t quite as disastrous as the 2^N complexity of our first example, but will lead to very unacceptable performance on larger invalid files.

In this situation, we know that each of the literal text blocks in our regular expression (the HTML tags, which function as delimiters) will occur only once in a valid HTML file. That makes it very easy to package each of the lazy dots (the delimited content) in an atomic group.

«<html>(?.*?<head>)(?.*?<title>)(?.*?</title>)(?.*?</head>)(?.*?<body[^>]*>)(?.*?</body>).*?</html>» will match a valid HTML file in the same number of steps as the original regex. The gain is that it will fail on an invalid HTML file almost as fast as it matches a valid one. When «</html>» fails to match, the regex engine backtracks, giving up the match for the last lazy dot. But then, there’s nothing further to backtrack to. Since all of the lazy dots are in an atomic group, the regex engines has discarded their backtracking positions. The groups function as a “do not expand further” roadblock. The regex engine is forced to announce failure immediately.

I’m sure you’ve noticed that each atomic group also contains an HTML tag after the lazy dot. This is critical. We do allow the lazy dot to backtrack until its matching HTML tag was found. E.g. when «.*?</body>» is processing “Last paragraph</p></body>”, the «</» regex tokens will match „</” in “</p>”. However, «b» will fail “p”. At that point, the regex engine will backtrack and expand the lazy dot to include „</p>”. Since the regex engine hasn’t left the atomic group yet, it is free to backtrack inside the group. Once «</body>» has matched, and the regex engine leaves the atomic group, it discards the lazy dot’s backtracking positions. Then it can no longer be expanded.

Essentially, what we’ve done is to bind a repeated regex token (the lazy dot to match HTML content) to the non-repeated regex token that follows it (the literal HTML tag). Since anything, including HTML tags, can appear between the HTML tags in our regular expression, we cannot use a negated character class instead of the lazy dot to prevent the delimiting HTML tags from being matched as HTML content. But we can and did achieve the same result by combining each lazy dot and the HTML tag following it into an atomic group. As soon as the HTML tag is matched, the lazy dot’s match is locked down. This ensures that the lazy dot will never match the HTML tag that should be matched by the literal HTML tag in the regular expression.

11. Repeating a Capturing Group vs. Capturing a Repeated Group

When creating a regular expression that needs a capturing group to grab part of the text matched, a common mistake is to repeat the capturing group instead of capturing a repeated group. The difference is that the repeated capturing group will capture only the last iteration, while a group capturing another group that's repeated will capture all iterations. An example will make this clear.

Let's say you want to match a tag like „!abc!” or „!123!”. Only these two are possible, and you want to capture the „abc” or „123” to figure out which tag you got. That's easy enough: «!(abc|123)!» will do the trick. Now let's say that the tag can contain multiple sequences of “abc” and “123”, like „!abc123!” or „!123abcabc!”. The quick and easy solution is «!(abc|123)+!». This regular expression will indeed match these tags. However, it no longer meets our requirement to capture the tag's label into the capturing group. When this regex matches „!abc123!”, the capturing group stores only „123”. When it matches „!123abcabc!”, it only stores „abc”.

This is easy to understand if we look at how the regex engine applies «!(abc|123)+!» to “!abc123!”. First, «!» matches „!”. The engine then enters the capturing group. It makes note that capturing group #1 was entered when the engine reached the position between the first and second character in the subject string. The first token in the group is «abc», which matches „abc”. A match is found, so the second alternative isn't tried. (The engine does store a backtracking position, but this won't be used in this example.) The engine now leaves the capturing group. It makes note that capturing group #1 was exited when the engine reached the position between the 4th and 5th characters in the string.

After having exited from the group, the engine notices the plus. The plus is greedy, so the group is tried again. The engine enters the group again, and takes note that capturing group #1 was entered between the 4th and 5th characters in the string. It also makes note that since the plus is not possessive, it may be backtracked. That is, if the group cannot be matched a second time, that's fine. In this backtracking note, the regex engine also saves the entrance and exit positions of the group during the previous iteration of the group. «abc» fails to match “123”, but «123» succeeds. The group is exited again. The exit position between characters 7 and 8 is stored.

The plus allows for another iteration, so the engine tries again. Backtracking info is stored, and the new entrance position for the group is saved. But now, both «abc» and «123» fail to match “!”. The group fails, and the engine backtracks. While backtracking, the engine restores the capturing positions for the group. Namely, the group was entered between characters 4 and 5, and existed between characters 7 and 8.

The engine proceeds with «!», which matches „!”. An overall match is found. The overall match spans the whole subject string. The capturing group spans characters 5, 6 and 7, or „123”. Backtracking information is discarded when a match is found, so there's no way to tell after the fact that the group had a previous iteration that matched „abc”. (The only exception to this is the .NET regex engine, which does preserve backtracking information for capturing groups after the match attempt.)

The solution to capturing „abc123” in this example should be obvious now: the regex engine should enter and leave the group only once. This means that the plus should be inside the capturing group rather than outside. Since we do need to group the two alternatives, we'll need to place a second capturing group around the repeated group: «!((abc|123)+)!». When this regex matches „!abc123!”, capturing group #1 will store „abc123”, and group #2 will store „123”. Since we're not interested in the inner group's match, we can optimize this regular expression by making the inner group non-capturing: «!(?:abc|123)+!».

12. Mixing Unicode and 8-bit Character Codes

Internally, computers deal with numbers, not with characters. When you save a text file, each character is mapped to a number, and the numbers are stored on disk. When you open a text file, the numbers are read and mapped back to characters. When processing text with a regular expression, the regular expression needs to use the same mapping as you used to create the file or string you want the regex to process.

When you simply type in all the characters in your regular expression, you normally don't have anything to worry about. The application or programming library that provides the regular expression functionality will know what text encodings your subject string uses, and process it accordingly. So if you want to search for the euro currency symbol, and you have a European keyboard, just press AltGr+E. Your regex «€» will find all euro symbols just fine.

But you can't press AltGr+E on a US keyboard. Or perhaps you like your source code to be 7-bit clean (i.e. plain ASCII). In those cases, you'll need to use a character escape in your regular expression.

If your regular expression engine supports Unicode, simply use the Unicode escape «\u20AC» (most Unicode flavors) or «\x{20AC}» (Perl and PCRE). U+20AC is the Unicode code point for the euro symbol. It will always match the euro symbol, whether your subject string is encoded in UTF-8, UTF-16, UCS-2 or whatever. Even when your subject string is encoded with a legacy 8-bit code page, there's no confusion. You may need to tell the application or regex engine what encoding your file uses. But «\u20AC» is always the euro symbol.

Most Unicode regex engines also support the 8-bit character escape «\xFF». However, its use is not recommended. For characters «\x00» through «\x7F», there's usually no trouble. The first 128 Unicode code points are identical to the ASCII table that most 8-bit code pages are based on.

But the interpretation of «\x80» and above may vary. A pure Unicode engine will treat this identical to «\u0080», which represents a Latin-1 control code. But what most people expect is that «\x80» matches the euro symbol, as that occupies position 80h in all Windows code pages. And it will when using an 8-bit regex engine if your text file is encoded using a Windows code page.

Since most people expect «\x80» to be treated as an 8-bit character rather than the Unicode code point «\u0080», some Unicode regex engines do exactly that. Some are hard-wired to use a particular code page, say Windows 1252 or your computer's default code page, to interpret 8-bit character codes.

Other engines will let it depend on the input string. E.g. the JGsoft engine will treat «\x80» as «\u0080» when searching through a Unicode text file, but as «\u20AC» when searching through a Windows 1252 text file. There's no magic here. It matches the character with index 80h in the text file, regardless of the text file's encoding. Unicode code point U+0080 is a Latin-1 control code, while Windows 1252 character index 80h is the euro symbol. In reverse, if you type in the euro symbol in a text editor, saving it as UTF-16 will save two bytes AC 20, while saving as Windows 1252 will give you one byte 80.

If you find the above confusing, simply don't use «\x80» through «\xFF» with a regex engine that supports Unicode.

8-bit Regex Engines

When working with a legacy (obsolete?) regular expression engine that works on 8-bit data only, you can't use Unicode escapes like «\u20AC». «\x80» is all you have. Note that even modern engines have legacy modes. E.g. the popular regex library PCRE runs as an 8-bit engine by default. You need to explicitly enable UTF-8 support if you want to use Unicode features. When you do, PCRE also expects you to convert your subject strings to UTF-8.

When crafting a regular expression for an 8-bit engine, you'll have to take into account which character set or code page you'll be working with. 8-bit regex engines just don't care. If you type «\x80» into your regex, it will match any byte 80h, regardless of what that byte represents. That'll be the euro symbol in a Windows 1252 text file, a control code in a Latin-1 file, and the digit zero in an EBCDIC file.

Even for literal characters in your regex, you'll have to match up the encoding you're using in the regular expression with the subject encoding. If your application is using the Latin-1 code page, and you use the regex «Ä», it'll match «Ř» when you search through a Latin-2 text file. The application would duly display this as „Ä” on the screen, because it's using the wrong code page. This problem is not really specific to regular expressions. You'll encounter it any time you're working with files and applications that use different 8-bit encodings.

So when working with 8-bit data, open the actual data you're working with in a hex editor. See the bytes being used, and specify those in your regular expression.

Where it gets really hairy is if you're processing Unicode files with an 8-bit engine. Let's go back to our text file with just a euro symbol. When saved as little endian UTF-16 (called “Unicode” on Windows), an 8-bit regex engine will see two bytes AC 20 (remember that little endian reverses the bytes). When saved as UTF-8 (which has no endianness), our 8-bit engine will see three bytes E2 82 AC. You'd need «\xE2\x82\xAC» to match the euro symbol in an UTF-8 file with an 8-bit regex engine.

Part 6

Regular Expression Reference

1. Basic Syntax Reference

Characters

Character:	Any character except [<code>\^\$. ?*+()</code>]
Description:	All characters except the listed special characters match a single instance of themselves. { and } are literal characters, unless they're part of a valid regular expression token (e.g. the {n} quantifier).
Example:	<code><<a></code> matches <code>„a”</code>
Character:	<code>\</code> (backslash) followed by any of [<code>\^\$. ?*+()</code>]
Description:	A backslash escapes special characters to suppress their special meaning.
Example:	<code><<\+></code> matches <code>„+”</code>
Character:	<code>\Q . . \E</code>
Description:	Matches the characters between <code>\Q</code> and <code>\E</code> literally, suppressing the meaning of special characters.
Example:	<code><<\Q+ - * / \E></code> matches <code>„+ - * /”</code>
Character:	<code>\xFF</code> where FF are 2 hexadecimal digits
Description:	Matches the character with the specified ASCII/ANSI value, which depends on the code page used. Can be used in character classes.
Example:	<code><<\xA9></code> matches <code>„©”</code> when using the Latin-1 code page.
Character:	<code>\n</code> , <code>\r</code> and <code>\t</code>
Description:	Match an LF character, CR character and a tab character respectively. Can be used in character classes.
Example:	<code><<\r\n></code> matches a DOS/Windows CRLF line break.
Character:	<code>\a</code> , <code>\e</code> , <code>\f</code> and <code>\v</code>
Description:	Match a bell character (<code>\x07</code>), escape character (<code>\x1B</code>), form feed (<code>\x0C</code>) and vertical tab (<code>\x0B</code>) respectively. Can be used in character classes.
Character:	<code>\cA</code> through <code>\cZ</code>
Description:	Match an ASCII character Control+A through Control+Z, equivalent to <code><<\x01></code> through <code><<\x1A></code> . Can be used in character classes.
Example:	<code><<\cM\cJ></code> matches a DOS/Windows CRLF line break.

Character Classes or Character Sets [abc]

Character:	[(opening square bracket)
Description:	Starts a character class. A character class matches a single character out of all the possibilities offered by the character class. Inside a character class, different rules apply. The rules in this section are only valid inside character classes. The rules outside this section are not valid in character classes, except for a few character escapes that are indicated with “can be used inside character classes”.
Character:	Any character except ^-] \ add that character to the possible matches for the character class.
Description:	All characters except the listed special characters.
Example:	«[abc]» matches „a”, „b” or „c”
Character:	\ (backslash) followed by any of ^-] \
Description:	A backslash escapes special characters to suppress their special meaning.
Example:	«[\^]» matches „^” or „]”
Character:	- (hyphen) except immediately after the opening [
Description:	Specifies a range of characters. (Specifies a hyphen if placed immediately after the opening [)
Example:	«[a-zA-Z0-9]» matches any letter or digit
Character:	^ (caret) immediately after the opening [
Description:	Negates the character class, causing it to match a single character <i>not</i> listed in the character class. (Specifies a caret if placed anywhere except after the opening [)
Example:	«[^a-d]» matches „x” (any character except a, b, c or d)
Character:	\d, \w and \s
Description:	Shorthand character classes matching digits, word characters (letters, digits, and underscores), and whitespace (spaces, tabs, and line breaks). Can be used inside and outside character classes.
Example:	«[\d\s]» matches a character that is a digit or whitespace
Character:	\D, \W and \S
Description:	Negated versions of the above. Should be used only outside character classes. (Can be used inside, but that is confusing.)
Example:	«\D» matches a character that is not a digit
Character:	[\b]
Description:	Inside a character class, \b is a backspace character.
Example:	«[\b\t]» matches a backspace or tab character

Dot

Character:	. (dot)
Description:	Matches any single character except line break characters \r and \n. Most regex flavors have an option to make the dot match line break characters too.
Example:	«. » matches „x” or (almost) any other character

Anchors

- Character: `^` (caret)
 Description: Matches at the start of the string the regex pattern is applied to. Matches a position rather than a character. Most regex flavors have an option to make the caret match after line breaks (i.e. at the start of a line in a file) as well.
 Example: `«^.»` matches „a” in “abc\ndef”. Also matches „d” in “multi-line” mode.
- Character: `$` (dollar)
 Description: Matches at the end of the string the regex pattern is applied to. Matches a position rather than a character. Most regex flavors have an option to make the dollar match before line breaks (i.e. at the end of a line in a file) as well. Also matches before the very last line break if the string ends with a line break.
 Example: `«. $»` matches „f” in “abc\ndef”. Also matches „c” in “multi-line” mode.
- Character: `\A`
 Description: Matches at the start of the string the regex pattern is applied to. Matches a position rather than a character. Never matches after line breaks.
 Example: `«\A.»` matches „a” in “abc”
- Character: `\Z`
 Description: Matches at the end of the string the regex pattern is applied to. Matches a position rather than a character. Never matches before line breaks, except for the very last line break if the string ends with a line break.
 Example: `«. \Z»` matches „f” in “abc\ndef”
- Character: `\z`
 Description: Matches at the end of the string the regex pattern is applied to. Matches a position rather than a character. Never matches before line breaks.
 Example: `«. \z»` matches „f” in “abc\ndef”

Word Boundaries

- Character: `\b`
 Description: Matches at the position between a word character (anything matched by `«\w»`) and a non-word character (anything matched by `«[^\w]»` or `«\W»`) as well as at the start and/or end of the string if the first and/or last characters in the string are word characters.
 Example: `«. \b»` matches „c” in “abc”
- Character: `\B`
 Description: Matches at the position between two word characters (i.e. the position between `«\w\w»`) as well as at the position between two non-word characters (i.e. `«\W\W»`).
 Example: `«\B.\B»` matches „b” in “abc”

Alternation

Character:	(pipe)
Description:	Causes the regex engine to match either the part on the left side, or the part on the right side. Can be strung together into a series of options.
Example:	«abc def xyz» matches „abc”, „def” or „xyz”
Character:	(pipe)
Description:	The pipe has the lowest precedence of all operators. Use grouping to alternate only part of the regular expression.
Example:	«abc(def xyz)» matches „abcdef” or „abcxyz”

Quantifiers

Character:	? (question mark)
Description:	Makes the preceding item optional. Greedy, so the optional item is included in the match if possible.
Example:	«abc?» matches „ab” or „abc”
Character:	??
Description:	Makes the preceding item optional. Lazy, so the optional item is excluded in the match if possible. This construct is often excluded from documentation because of its limited use.
Example:	«abc??» matches „ab” or „abc”
Character:	* (star)
Description:	Repeats the previous item zero or more times. Greedy, so as many items as possible will be matched before trying permutations with less matches of the preceding item, up to the point where the preceding item is not matched at all.
Example:	«".*» matches „def" "ghi"” in “abc "def" "ghi" jkl”
Character:	*? (lazy star)
Description:	Repeats the previous item zero or more times. Lazy, so the engine first attempts to skip the previous item, before trying permutations with ever increasing matches of the preceding item.
Example:	«".*?» matches „def"” in “abc "def" "ghi" jkl”
Character:	+ (plus)
Description:	Repeats the previous item once or more. Greedy, so as many items as possible will be matched before trying permutations with less matches of the preceding item, up to the point where the preceding item is matched only once.
Example:	«".+» matches „def" "ghi"” in “abc "def" "ghi" jkl”
Character:	+? (lazy plus)
Description:	Repeats the previous item once or more. Lazy, so the engine first matches the previous item only once, before trying permutations with ever increasing matches of the preceding item.
Example:	«".+?» matches „def"” in “abc "def" "ghi" jkl”

- Character: $\{n\}$ where n is an integer ≥ 1
 Description: Repeats the previous item exactly n times.
 Example: $\langle\langle a\{3\}\rangle\rangle$ matches „aaa”
- Character: $\{n,m\}$ where $n \geq 0$ and $m \geq n$
 Description: Repeats the previous item between n and m times. Greedy, so repeating m times is tried before reducing the repetition to n times.
 Example: $\langle\langle a\{2,4\}\rangle\rangle$ matches „aaaa”, „aaa” or „aa”
- Character: $\{n,m\}?$ where $n \geq 0$ and $m \geq n$
 Description: Repeats the previous item between n and m times. Lazy, so repeating n times is tried before increasing the repetition to m times.
 Example: $\langle\langle a\{2,4\}?\rangle\rangle$ matches „aa”, „aaa” or „aaaa”
- Character: $\{n, \}$ where $n \geq 0$
 Description: Repeats the previous item at least n times. Greedy, so as many items as possible will be matched before trying permutations with less matches of the preceding item, up to the point where the preceding item is matched only n times.
 Example: $\langle\langle a\{2, \}\rangle\rangle$ matches „aaaaa” in “aaaaa”
- Character: $\{n, \}?$ where $n \geq 0$
 Description: Repeats the previous item n or more times. Lazy, so the engine first matches the previous item n times, before trying permutations with ever increasing matches of the preceding item.
 Example: $\langle\langle a\{2, \}?\rangle\rangle$ matches „aa” in “aaaaa”

2. Advanced Syntax Reference

Grouping and Backreferences

Character:	(regex)
Description:	Round brackets group the regex between them. They capture the text matched by the regex inside them that can be reused in a backreference, and they allow you to apply regex operators to the entire grouped regex.
Example:	«(abc){3}» matches „abcabcabc”. First group matches „abc”.
Character:	(?:regex)
Description:	Non-capturing parentheses group the regex so you can apply regex operators, but do not capture anything and do not create backreferences.
Example:	«(?:abc){3}» matches „abcabcabc”. No groups.
Character:	\1 through \9
Description:	Substituted with the text matched between the 1st through 9th pair of capturing parentheses. Some regex flavors allow more than 9 backreferences.
Example:	«(abc def)=\1» matches „abc=abc” or „def=def”, but not “abc=def” or “def=abc”.

Modifiers

Character:	(?i)
Description:	Turn on case insensitivity for the remainder of the regular expression. (Older regex flavors may turn it on for the entire regex.)
Example:	«te(?i)st» matches „teST” but not “TEST”.
Character:	(?-i)
Description:	Turn off case insensitivity for the remainder of the regular expression.
Example:	«(?i)te(?-i)st» matches „TESt” but not “TEST”.

Character:	(?s)
Description:	Turn on “dot matches newline” for the remainder of the regular expression. (Older regex flavors may turn it on for the entire regex.)
Character:	(?-s)
Description:	Turn off “dot matches newline” for the remainder of the regular expression.
Character:	(?m)
Description:	Caret and dollar match after and before newlines for the remainder of the regular expression. (Older regex flavors may apply this to the entire regex.)
Character:	(?-m)
Description:	Caret and dollar only match at the start and end of the string for the remainder of the regular expression.
Character:	(?x)
Description:	Turn on free-spacing mode to ignore whitespace between regex tokens, and allow # comments.
Character:	(?-x)
Description:	Turn off free-spacing mode.
Character:	(?i-sm)
Description:	Turns on the option “i” and turns off “s” and “m” for the remainder of the regular expression. (Older regex flavors may apply this to the entire regex.)
Character:	(?i-sm:regex)
Description:	Matches the regex inside the span with the option “i” turned on and “m” and “s” turned off.
Example:	«(?i:te)st» matches „TESt” but not “TEST”.

Atomic Grouping and Possessive Quantifiers

Character:	(?>regex)
Description:	Atomic groups prevent the regex engine from backtracking back into the group (forcing the group to discard part of its match) after a match has been found for the group. Backtracking can occur inside the group before it has matched completely, and the engine can backtrack past the entire group, discarding its match entirely. Eliminating needless backtracking provides a speed increase. Atomic grouping is often indispensable when nesting quantifiers to prevent a catastrophic amount of backtracking as the engine needlessly tries pointless permutations of the nested quantifiers.
Example:	«x(?>\w+)x» is more efficient than «x\w+x» if the second x cannot be matched.
Character:	?+, *+, ++ and {m, n}+
Description:	Possessive quantifiers are a limited yet syntactically cleaner alternative to atomic grouping. Only available in a few regex flavors. They behave as normal greedy quantifiers, except that they will not give up part of their match for backtracking.
Example:	«x++» is identical to «(?>x+)»

Lookaround

Character:	<code>(?=regex)</code>
Description:	Zero-width positive lookahead. Matches at a position where the pattern inside the lookahead can be matched. Matches only the position. It does not consume any characters or expand the match. In a pattern like <code>«one(=two)three»</code> , both <code>«two»</code> and <code>«three»</code> have to match at the position where the match of <code>«one»</code> ends.
Example:	<code>«t(=s)»</code> matches the second <code>„t”</code> in <code>„streets”</code> .
Character:	<code>(?!regex)</code>
Description:	Zero-width negative lookahead. Identical to positive lookahead, except that the overall match will only succeed if the regex inside the lookahead fails to match.
Example:	<code>«t(!s)»</code> matches the first <code>„t”</code> in <code>„streets”</code> .
Character:	<code>(?<=regex)</code>
Description:	Zero-width positive lookbehind. Matches at a position if the pattern inside the lookahead can be matched ending at that position (i.e. to the left of that position). Depending on the regex flavor you’re using, you may not be able to use quantifiers and/or alternation inside lookbehind.
Example:	<code>«(?<=s)t»</code> matches the first <code>„t”</code> in <code>„streets”</code> .
Character:	<code>(?<!regex)</code>
Description:	Zero-width negative lookbehind. Matches at a position if the pattern inside the lookahead cannot be matched ending at that position.
Example:	<code>«(?<!s)t»</code> matches the second <code>„t”</code> in <code>„streets”</code> .

Continuing from The Previous Match

Character:	<code>\G</code>
Description:	Matches at the position where the previous match ended, or the position where the current match attempt started (depending on the tool or regex flavor). Matches at the start of the string during the first match attempt.
Example:	<code>«\G[a-z]»</code> first matches <code>„a”</code> , then matches <code>„b”</code> and then fails to match in <code>“ab_cd”</code> .

Conditionals

Character:	<code>(?(?=regex)then else)</code>
Description:	If the lookahead succeeds, the “then” part must match for the overall regex to match. If the lookahead fails, the “else” part must match for the overall regex to match. Not just positive lookahead, but all four lookarounds can be used. Note that the lookahead is zero-width, so the “then” and “else” parts need to match and consume the part of the text matched by the lookahead as well.
Example:	<code>«(? (?<=a)b c)»</code> matches the second <code>„b”</code> and the first <code>„c”</code> in <code>“babxcac”</code>

Character: `(?(1)then|else)`

Description: If the first capturing group took part in the match attempt thus far, the “then” part must match for the overall regex to match. If the first capturing group did not take part in the match, the “else” part must match for the overall regex to match.

Example: `«(a)?(?(1)b|c)»` matches „ab”, the first „c” and the second „c” in “babxcac”

Comments

Character: `(?#comment)`

Description: Everything between `(?#` and `)` is ignored by the regex engine.

Example: `«a(?#foobar)b»` matches „ab”

3. Unicode Syntax Reference

Unicode Characters

Character: `\X`
 Description: Matches a single Unicode grapheme, whether encoded as a single code point or multiple code points using combining marks. A grapheme most closely resembles the everyday concept of a “character”.

Example: `«\X»` matches „à” encoded as U+0061 U+0300, „à” encoded as U+00E0, „©”, etc.

Character: `\uFFFF` where FFFF are 4 hexadecimal digits
 Description: Matches a specific Unicode code point. Can be used inside character classes.
 Example: `«\u00E0»` matches „à” encoded as U+00E0 only. `«\u00A9»` matches „©”

Character: `\x{FFFF}` where FFFF are 1 to 4 hexadecimal digits
 Description: Perl syntax to match a specific Unicode code point. Can be used inside character classes.
 Example: `«\x{E0}»` matches „à” encoded as U+00E0 only. `«\x{A9}»` matches „©”

Unicode Properties, Scripts and Blocks

Character: `\p{L}` or `\p{Letter}`
 Description: Matches a single Unicode code point that has the property “letter”. See Unicode Character Properties in the tutorial for a complete list of properties. Each Unicode code point has exactly one property. Can be used inside character classes.
 Example: `«\p{L}»` matches „à” encoded as U+00E0; `«\p{S}»` matches „©”

Character: `\p{Arabic}`
 Description: Matches a single Unicode code point that is part of the Unicode script “Arabic”. See Unicode Scripts in the tutorial for a complete list of scripts. Each Unicode code point is part of exactly one script. Can be used inside character classes.
 Example: `«\p{Thai}»` matches one of 83 code points in Thai script, from „ñ” until „ᩉ”

Character: `\p{InBasicLatin}`
 Description: Matches a single Unicode code point that is part of the Unicode block “BasicLatin”. See Unicode Blocks in the tutorial for a complete list of blocks. Each Unicode code point is part of exactly one block. Blocks may contain unassigned code points. Can be used inside character classes.
 Example: `«\p{InLatinExtended-A}»` any of the code points in the block U+100 until U+17F („Ā” until „ƒ”)

Character: `\P{L}` or `\P{Letter}`
 Description: Matches a single Unicode code point that does *not* have the property “letter”. You can also use `\P` to match a code point that is not part of a particular Unicode block or script. Can be used inside character classes.
 Example: `«\P{L}»` matches „©”

4. Syntax Reference for Specific Regex Flavors

.NET Syntax for Named Capture and Backreferences

- Character: `(?<name>regex)`
 Description: Round brackets group the regex between them. They capture the text matched by the regex inside them that can be referenced by the name between the sharp brackets. The name may consist of letters and digits.
- Character: `(?'name' regex)`
 Description: Round brackets group the regex between them. They capture the text matched by the regex inside them that can be referenced by the name between the single quotes. The name may consist of letters and digits.
- Character: `\k<name>`
 Description: Substituted with the text matched by the capturing group with the given name.
 Example: `<<(?'group'abc|def)=\k'group'>>` matches „abc=abc” or „def=def”, but not “abc=def” or “def=abc”.
- Character: `\k'name'`
 Description: Substituted with the text matched by the capturing group with the given name.
 Example: `<<(?'group'abc|def)=\k'group'>>` matches „abc=abc” or „def=def”, but not “abc=def” or “def=abc”.
- Character: `(?(name)then|else)`
 Description: If the capturing group “name” took part in the match attempt thus far, the “then” part must match for the overall regex to match. If the capturing group “name” did not take part in the match, the “else” part must match for the overall regex to match.
 Example: `<<(?'group'a)?(?'group'b|c)>>` matches „ab”, the first „c” and the second „c” in “babxcac”

Python Syntax for Named Capture and Backreferences

- Character: `(?P<name>regex)`
 Description: Round brackets group the regex between them. They capture the text matched by the regex inside them that can be referenced by the name between the sharp brackets. The name may consist of letters and digits.
- Character: `(?P=name)`
 Description: Substituted with the text matched by the capturing group with the given name. Not a group, despite the syntax using round brackets.
 Example: `<<(?'group'abc|def)=(?'group')>>` matches „abc=abc” or „def=def”, but not “abc=def” or “def=abc”.

XML Character Classes

Character:	<code>\i</code>
Description:	Matches any character that may be the first character of an XML name, i.e. « <code>[_: A-Za-z]</code> ».
Character:	<code>\c</code>
Description:	« <code>\c</code> » matches any character that may occur after the first character in an XML name, i.e. « <code>[-._: A-Za-z0-9]</code> »
Example:	« <code>\i \c*</code> » matches an XML name like „ <code>xml: schema</code> ”
Character:	<code>\I</code>
Description:	Matches any character that cannot be the first character of an XML name, i.e. « <code>[^_ : A-Za-z]</code> ».
Character:	<code>\C</code>
Description:	Matches any character that cannot occur in an XML name, i.e. « <code>[^-. _ : A-Za-z0-9]</code> ».
Character:	<code>[abc - [xyz]]</code>
Description:	Subtracts character class “xyz” from character class “abc”. The result matches any single character that occurs in the character class “abc” but not in the character class “xyz”.
Example:	« <code>[a-z - [aeiou]]</code> » matches any letter that is not a vowel (i.e. a consonant).

POSIX Bracket Expressions

Character:	<code>[:alpha:]</code>
Description:	Matches one character from a POSIX character class. Can only be used in a bracket expression.
Example:	« <code>[[:digit:]][:lower:]]</code> » matches one of „0” through „9” or „a” through „z”
Character:	<code>[.span-ll.]</code>
Description:	Matches a POSIX collation sequence. Can only be used in a bracket expression.
Example:	« <code>[.span-ll.]</code> » matches „ll” in the Spanish locale
Character:	<code>[=x=]</code>
Description:	Matches a POSIX character equivalence. Can only be used in a bracket expression.
Example:	« <code>[=e=]</code> » matches „e”, „é”, „è” and „ê” in the French locale

5. Regular Expression Flavor Comparison

The table below compares which regular expression flavors support which regex features and syntax. The features are listed in the same order as in the regular expression reference.

The comparison shows regular expression flavors rather than particular applications or programming languages implementing one of those regular expression flavors.

- JGsoft: This flavor is used by the Just Great Software products, including PowerGREP and EditPad Pro.
- .NET: This flavor is used by programming languages based on the Microsoft .NET framework versions 1.x, 2.0 or 3.x. It is generally also the regex flavor used by applications developed in these programming languages.
- Java: The regex flavor of the `java.util.regex` package, available in the Java 4 (JDK 1.4.x) and later. A few features were added in Java 5 (JDK 1.5.x) and Java 6 (JDK 1.6.x). It is generally also the regex flavor used by applications developed in Java.
- Perl: The regex flavor used in the Perl programming language, versions 5.6 and 5.8. Versions prior to 5.6 do not support Unicode.
- PCRE: The open source PCRE library. The feature set described here is available in PCRE 5.x and 6.x. PCRE is the regex engine used by the TPerlRegEx Delphi component and the RegularExpressions and RegularExpressionsCore units in Delphi XE and C++Builder XE.
- ECMA (JavaScript): The regular expression syntax defined in the 3rd edition of the ECMA-262 standard, which defines the scripting language commonly known as JavaScript.
- Python: The regex flavor supported by Python's built-in `re` module.
- Ruby: The regex flavor built into the Ruby programming language.
- Tcl ARE: The regex flavor developed by Henry Spencer for the `regexp` command in Tcl 8.2 and 8.4, dubbed Advanced Regular Expressions.
- POSIX BRE: Basic Regular Expressions as defined in the IEEE POSIX standard 1003.2.
- POSIX ERE: Extended Regular Expressions as defined in the IEEE POSIX standard 1003.2.
- GNU BRE: GNU Basic Regular Expressions, which are POSIX BRE with GNU extensions, used in the GNU implementations of classic UNIX tools.
- GNU ERE: GNU Extended Regular Expressions, which are POSIX ERE with GNU extensions, used in the GNU implementations of classic UNIX tools.
- XML: The regular expression flavor defined in the XML Schema standard.
- XPath: The regular expression flavor defined in the XQuery 1.0 and XPath 2.0 Functions and Operators standard.

Applications and languages implementing one of the above flavors are:

- AceText: Version 2 and later use the JGsoft engine. Version 1 did not support regular expressions at all.
- awk: The awk UNIX tool and programming language uses POSIX ERE.
- C#: As a .NET programming language, C# can use the `System.Text.RegularExpressions` classes, listed as ".NET" below.
- Delphi for .NET: As a .NET programming language, the .NET version of Delphi can use the `System.Text.RegularExpressions` classes, listed as ".NET" below.
- Delphi for Win32: Delphi for Win32 does not have built-in regular expression support. Many free PCRE wrappers are available.

- EditPad Pro: Version 6 and later use the JGsoft engine. Earlier versions used PCRE, without Unicode support.
- egrep: The traditional UNIX egrep command uses the “POSIX ERE” flavor, though not all implementations fully adhere to the standard. Linux usually ships with the GNU implementation, which use “GNU ERE”.
- grep: The traditional UNIX grep command uses the “POSIX BRE” flavor, though not all implementations fully adhere to the standard. Linux usually ships with the GNU implementation, which use “GNU BRE”.
- Emacs: The GNU implementation of this classic UNIX text editor uses the “GNU ERE” flavor, except that POSIX classes, collations and equivalences are not supported.
- Java: The regex flavor of the java.util.regex package is listed as “Java” in the table below.
- JavaScript: JavaScript’s regex flavor is listed as “ECMA” in the table below.
- MySQL: MySQL uses POSIX Extended Regular Expressions, listed as “POSIX ERE” in the table below.
- Oracle: Oracle Database 10g implements POSIX Extended Regular Expressions, listed as “POSIX ERE” in the table below. Oracle supports backreferences \1 through \9, though these are not part of the POSIX ERE standard.
- Perl: Perl’s regex flavor is listed as “Perl” in the table below.
- PHP: PHP’s ereg functions implement the “POSIX ERE” flavor, while the preg functions implement the “PCRE” flavor.
- PostgreSQL: PostgreSQL 7.4 and later uses Henry Spencer’s “Advanced Regular Expressions” flavor, listed as “Tcl ARE” in the table below. Earlier versions used POSIX Extended Regular Expressions, listed as POSIX ERE.
- PowerGREP: Version 3 and later use the JGsoft engine. Earlier versions used PCRE, without Unicode support.
- PowerShell: PowerShell’s built-in -match and -replace operators use the .NET regex flavor. PowerShell can also use the System.Text.RegularExpressions classes directly.
- Python: Python’s regex flavor is listed as “Python” in the table below.
- R: The regular expression functions in the R language for statistical programming use either the POSIX ERE flavor (default), the PCRE flavor (perl = true) or the POSIX BRE flavor (perl = false, extended = false).
- REALbasic: REALbasic’s RegEx class is a wrapper around PCRE.
- RegexBuddy: Version 3 and later use a special version of the JGsoft engine that emulates all the regular expression flavors in this comparison. Version 2 supported the JGsoft regex flavor only. Version 1 used PCRE, without Unicode support.
- Ruby: Ruby’s regex flavor is listed as “Ruby” in the table below.
- sed: The sed UNIX tool uses POSIX BRE. Linux usually ships with the GNU implementation, which use “GNU BRE”.
- Tcl: Tcl’s Advanced Regular Expression flavor, the default flavor in Tcl 8.2 and later, is listed as “Tcl ARE” in the table below. Tcl’s Extended Regular Expression and Basic Regular Expression flavors are listed as “POSIX ERE” and “POSIX BRE” in the table below.
- VBScript: VBScript’s RegExp object uses the same regex flavor as JavaScript, which is listed as “ECMA” in the table below.
- Visual Basic 6: Visual Basic 6 does not have built-in support for regular expressions, but can easily use the "Microsoft VBScript Regular Expressions 5.5" COM object, which implements the “ECMA” flavor listed below.
- Visual Basic.NET: As a .NET programming language, VB.NET can use the System.Text.RegularExpressions classes, listed as ".NET" below.
- wxWidgets: The wxRegEx class supports 3 flavors. wxRE_ADVANCED is the “Tcl ARE” flavor, wxRE_EXTENDED is “POSIX ERE” and wxRE_BASIC is “POSIX BRE”.

- XML Schema: The XML Schema regular expression flavor is listed as “XML” in the table below.
- XPath: The regex flavor used by XPath functions is listed as “XPath” in the table below.
- XQuery: The regex flavor used by XQuery functions is listed as “XPath” in the table below.

Characters

Feature: Backslash escapes one metacharacter
 Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE, GNU BRE, GNU ERE, XML, XPath

Feature: `\Q...\E` escapes a string of metacharacters
 Supported by: JGsoft, Java, Perl, PCRE

Feature: `\x00` through `\xFF` (ASCII character)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE

Feature: `\n` (LF), `\r` (CR) and `\t` (tab)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, XML, XPath

Feature: `\f` (form feed) and `\v` (vtab)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE

Feature: `\a` (bell)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, Python, Ruby, Tcl ARE

Feature: `\e` (escape)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, Ruby, Tcl ARE

Feature: `\b` (backspace) and `\B` (backslash)
 Supported by: Tcl ARE

Feature: `\cA` through `\cZ` (control character)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Tcl ARE

Feature: `\ca` through `\cz` (control character)
 Supported by: JGsoft, .NET, Perl, PCRE, JavaScript, Tcl ARE

Character Classes or Character Sets [abc]

Feature: [abc] character class
 Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE, GNU BRE, GNU ERE, XML, XPath

Feature: [^abc] negated character class
 Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE, GNU BRE, GNU ERE, XML, XPath

- Feature: [a-z] character class range
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE, GNU BRE, GNU ERE, XML, XPath
- Feature: Hyphen in [\d-z] is a literal
Supported by: JGsoft, .NET, Java, Perl, PCRE
- Feature: Hyphen in [a-\d] is a literal
Supported by: JGsoft, PCRE
- Feature: Backslash escapes one character class metacharacter
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, XML, XPath
- Feature: \Q...\E escapes a string of character class metacharacters
Supported by: JGsoft, Java, Perl, PCRE
- Feature: \d shorthand for digits
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, XML, XPath
- Feature: \w shorthand for word characters
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, GNU BRE, GNU ERE, XML, XPath
- Feature: \s shorthand for whitespace
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, GNU BRE, GNU ERE, XML, XPath
- Feature: \D, \W and \S shorthand negated character classes
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, GNU BRE, GNU ERE, XML, XPath
- Feature: [\b] backspace
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE

Dot

- Feature: . (dot; any character except line break)
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE, GNU BRE, GNU ERE, XML, XPath

Anchors

- Feature: ^ (start of string/line)
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE, GNU BRE, GNU ERE, XPath

Feature: `$` (end of string/line)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE, GNU BRE, GNU ERE, XPath

Feature: `\A` (start of string)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, Python, Ruby, Tcl ARE

Feature: `\Z` (end of string, before final line break)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, Ruby, Tcl ARE

Feature: `\z` (end of string)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, Python, Ruby

Feature: `\`` (start of string)
 Supported by: GNU BRE, GNU ERE

Feature: `\'` (end of string)
 Supported by: GNU BRE, GNU ERE

Word Boundaries

Feature: `\b` (at the beginning or end of a word)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, GNU BRE, GNU ERE

Feature: `\B` (NOT at the beginning or end of a word)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, GNU BRE, GNU ERE

Feature: `\y` (at the beginning or end of a word)
 Supported by: JGsoft, Tcl ARE

Feature: `\Y` (NOT at the beginning or end of a word)
 Supported by: JGsoft, Tcl ARE

Feature: `\m` (at the beginning of a word)
 Supported by: JGsoft, Tcl ARE

Feature: `\M` (at the end of a word)
 Supported by: JGsoft, Tcl ARE

Feature: `\<` (at the beginning of a word)
 Supported by: GNU BRE, GNU ERE

Feature: `\>` (at the end of a word)
 Supported by: GNU BRE, GNU ERE

Alternation

Feature: | (alternation)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX ERE, GNU BRE, GNU ERE, XML, XPath

Quantifiers

Feature: ? (0 or 1)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX ERE, GNU BRE, GNU ERE, XML, XPath

Feature: * (0 or more)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE, GNU BRE, GNU ERE, XML, XPath

Feature: + (1 or more)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX ERE, GNU BRE, GNU ERE, XML, XPath

Feature: {n} (exactly n)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE, GNU BRE, GNU ERE, XML, XPath

Feature: {n, m} (between n and m)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE, GNU BRE, GNU ERE, XML, XPath

Feature: {n, } (n or more)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE, GNU BRE, GNU ERE, XML, XPath

Feature: ? after any of the above quantifiers to make it “lazy”
 Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, XPath

Grouping and Backreferences

Feature: (regex) (numbered capturing group)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE, GNU BRE, GNU ERE, XML, XPath

Feature: (? : regex) (non-capturing group)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE

Feature: \1 through \9 (backreferences)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, GNU BRE, GNU ERE, XPath

Feature: `\10` through `\99` (backreferences)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, XPath

Feature: Forward references `\1` through `\9`
 Supported by: JGsoft, .NET, Java, Perl, PCRE, Ruby

Feature: Nested references `\1` through `\9`
 Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Ruby

Feature: Backreferences non-existent groups are an error
 Supported by: JGsoft, .NET, Java, Perl, PCRE, Python, Tcl ARE, POSIX BRE, GNU BRE, GNU ERE, XPath

Feature: Backreferences to failed groups also fail
 Supported by: JGsoft, .NET, Java, Perl, PCRE, Python, Ruby, Tcl ARE, POSIX BRE, GNU BRE, GNU ERE, XPath

Modifiers

Feature: `(?i)` (case insensitive)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, Python, Ruby, Tcl ARE

Feature: `(?s)` (dot matches newlines)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, Python, Ruby

Feature: `(?m)` (`^` and `$` match at line breaks)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, Python

Feature: `(?x)` (free-spacing mode)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, Python, Ruby, Tcl ARE

Feature: `(?n)` (explicit capture)
 Supported by: JGsoft, .NET

Feature: `(?-ismxn)` (turn off mode modifiers)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, Ruby

Feature: `(?ismxn:group)` (mode modifiers local to group)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, Ruby

Atomic Grouping and Possessive Quantifiers

Feature: `(?>regex)` (atomic group)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, Ruby

Feature: `?+`, `*+`, `++` and `{m, n}+` (possessive quantifiers)
 Supported by: JGsoft, Java, PCRE

Lookaround

Feature: `(?=regex)` (positive lookahead)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE

Feature: `(?!regex)` (negative lookahead)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE

Feature: `(?<=text)` (positive lookbehind)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, Python

Feature: `(?<!text)` (negative lookbehind)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, Python

Continuing from The Previous Match

Feature: `\G` (start of match attempt)
 Supported by: JGsoft, .NET, Java, Perl, PCRE, Ruby

Conditionals

Feature: `(?(?=regex)then|else)` (using any lookaround)
 Supported by: JGsoft, .NET, Perl, PCRE

Feature: `(?(regex)then|else)`
 Supported by: .NET

Feature: `(?(1)then|else)`
 Supported by: JGsoft, .NET, Perl, PCRE, Python

Feature: `(?(group)then|else)`
 Supported by: JGsoft, .NET, PCRE, Python

Comments

Feature: `(?#comment)`
 Supported by: JGsoft, .NET, Perl, PCRE, Python, Ruby, Tcl ARE

Free-Spacing Syntax

Feature: Free-spacing syntax supported
 Supported by: JGsoft, .NET, Java, Perl, PCRE, Python, Ruby, Tcl ARE, XPath

Feature: Character class is a single token
Supported by: JGsoft, .NET, Perl, PCRE, Python, Ruby, Tcl ARE, XPath

Feature: # starts a comment
Supported by: JGsoft, .NET, Java, Perl, PCRE, Python, Ruby, Tcl ARE

Unicode Characters

Feature: \X (Unicode grapheme)
Supported by: JGsoft, Perl, PCRE

Feature: \u0000 through \uFFFF (Unicode character)
Supported by: JGsoft, .NET, Java, JavaScript, Python, Tcl ARE

Feature: \x{0} through \x{FFFF} (Unicode character)
Supported by: JGsoft, Perl, PCRE

Unicode Properties, Scripts and Blocks

Feature: \pL through \pC (Unicode properties)
Supported by: JGsoft, Java, Perl, PCRE

Feature: \p{L} through \p{C} (Unicode properties)
Supported by: JGsoft, .NET, Java, Perl, PCRE, XML, XPath

Feature: \p{Lu} through \p{Cn} (Unicode property)
Supported by: JGsoft, .NET, Java, Perl, PCRE, XML, XPath

Feature: \p{L&} and \p{Letter&} (equivalent of [\p{Lu}\p{Ll}\p{Lt}]) (Unicode properties)
Supported by: JGsoft, Perl, PCRE

Feature: \p{IsL} through \p{IsC} (Unicode properties)
Supported by: JGsoft, Java, Perl

Feature: \p{IsLu} through \p{IsCn} (Unicode property)
Supported by: JGsoft, Java, Perl

Feature: \p{Letter} through \p{Other} (Unicode properties)
Supported by: JGsoft, Perl

Feature: \p{Lowercase_Letter} through \p{Not_Assigned} (Unicode property)
Supported by: JGsoft, Perl

Feature: \p{IsLetter} through \p{IsOther} (Unicode properties)
Supported by: JGsoft, Perl

Feature: `\p{IsLowercase_Letter}` through `\p{IsNot_Assigned}` (Unicode property)
Supported by: JGsoft, Perl

Feature: `\p{Arabic}` through `\p{Yi}` (Unicode script)
Supported by: JGsoft, Perl, PCRE

Feature: `\p{IsArabic}` through `\p{IsYi}` (Unicode script)
Supported by: JGsoft, Perl

Feature: `\p{BasicLatin}` through `\p{Specials}` (Unicode block)
Supported by: JGsoft, Perl

Feature: `\p{InBasicLatin}` through `\p{InSpecials}` (Unicode block)
Supported by: JGsoft, Java, Perl

Feature: `\p{IsBasicLatin}` through `\p{IsSpecials}` (Unicode block)
Supported by: JGsoft, .NET, Perl, XML, XPath

Feature: Part between `{ }` in all of the above is case insensitive
Supported by: JGsoft, Perl

Feature: Spaces, hyphens and underscores allowed in all long names listed above (e.g. BasicLatin can be written as Basic-Latin or Basic_Latin or Basic Latin)
Supported by: JGsoft, Java, Perl

Feature: `\P` (negated variants of all `\p` as listed above)
Supported by: JGsoft, .NET, Java, Perl, PCRE, XML, XPath

Feature: `\p{^ . . . }` (negated variants of all `\p{ . . . }` as listed above)
Supported by: JGsoft, Perl, PCRE

Named Capture and Backreferences

Feature: `(?<name>regex)` (.NET-style named capturing group)
Supported by: JGsoft, .NET

Feature: `(?'name' regex)` (.NET-style named capturing group)
Supported by: JGsoft, .NET

Feature: `\k<name>` (.NET-style named backreference)
Supported by: JGsoft, .NET

Feature: `\k'name'` (.NET-style named backreference)
Supported by: JGsoft, .NET

Feature: `(?P<name>regex)` (Python-style named capturing group)
Supported by: JGsoft, PCRE, Python

Feature: (?P=name) (Python-style named backreference)
Supported by: JGsoft, PCRE, Python

Feature: multiple capturing groups can have the same name
Supported by: JGsoft, .NET

XML Character Classes

Feature: \i, \I, \c and \C shorthand XML name character classes
Supported by: XML, XPath

Feature: [abc - [abc]] character class subtraction
Supported by: JGsoft, .NET, XML, XPath

POSIX Bracket Expressions

Feature: [:alpha:] POSIX character class
Supported by: JGsoft, Perl, PCRE, Ruby, Tcl ARE, POSIX BRE, POSIX ERE, GNU BRE, GNU ERE

Feature: \p{Alpha} POSIX character class
Supported by: JGsoft, Java

Feature: \p{IsAlpha} POSIX character class
Supported by: JGsoft, Perl

Feature: [.span-11.] POSIX collation sequence
Supported by: Tcl ARE, POSIX BRE, POSIX ERE, GNU BRE, GNU ERE

Feature: [=x=] POSIX character equivalence
Supported by: Tcl ARE, POSIX BRE, POSIX ERE, GNU BRE, GNU ERE

6. Replacement Text Reference

The table below compares the various tokens that the various tools and languages discussed in this book recognize in the replacement text during search-and-replace operations.

The list of replacement text flavors is not the same as the list of regular expression flavors in the regex features comparison. The reason is that the replacements are not made by the regular expression engine, but by the tool or programming library providing the search-and-replace capability. The result is that tools or languages using the same regex engine may behave differently when it comes to making replacements. E.g. The PCRE library does not provide a search-and-replace function. All tools and languages implementing PCRE use their own search-and-replace feature, which may result in differences in the replacement text syntax. So these are listed separately.

To make the table easier to read, I did group tools and languages that use the exact same replacement text syntax. The labels for the replacement text flavors are only relevant in the table below. E.g. the .NET framework does have built-in search-and-replace function in its `Regex` class, which is used by all tools and languages based on the .NET framework. So these are listed together under ".NET".

Note that the escape rules below only refer to the replacement text syntax. If you type the replacement text in an input box in the application you're using, or if you retrieve the replacement text from user input in the software you're developing, these are the only escape rules that apply. If you pass the replacement text as a literal string in programming language source code, you'll need to apply the language's string escape rules on top of the replacement text escape rules. E.g. for languages that require backslashes in string literals to be escaped, you'll need to use `"\\1"` instead of `"\1"` to get the first backreference.

A flavor can have four levels of support (or non-support) for a particular token:

- A "YES" in the table below indicates the token will be substituted.
 - A "no" indicates the token will remain in the replacement as literal text. Note that languages that use variable interpolation in strings may still replace tokens indicated as unsupported below, if the syntax of the token corresponds with the variable interpolation syntax. E.g. in Perl, `$0` is replaced with the name of the script.
 - The "string" label indicates that the syntax is supported by string literals in the language's source code. For languages like PHP that have interpolated (double quotes) and non-interpolated (single quotes) variants, you'll need to use the interpolated string style. String-level support also means that the character escape won't be interpreted for replacement text typed in by the user or read from a file.
 - Finally, "error" indicates the token will result in an error condition or exception, preventing any replacements being made at all.
- JGsoft: This flavor is used by the Just Great Software products, including PowerGREP, EditPad Pro and AceText. It is also used by the TPerlRegEx Delphi component and the RegularExpressions and RegularExpressionsCore units in Delphi XE and C++Builder XE.
 - .NET: This flavor is used by programming languages based on the Microsoft .NET framework versions 1.x, 2.0 or 3.0. It is generally also the regex flavor used by applications developed in these programming languages.
 - Java: The regex flavor of the `java.util.regex` package, available in the Java 4 (JDK 1.4.x) and later. A few features were added in Java 5 (JDK 1.5.x) and Java 6 (JDK 1.6.x). It is generally also the regex flavor used by applications developed in Java.
 - Perl: The regex flavor used in the Perl programming language, as of version 5.8.

- ECMA (JavaScript): The regular expression syntax defined in the 3rd edition of the ECMA-262 standard, which defines the scripting language commonly known as JavaScript. The VBScript RegExp object, which is also commonly used in VB 6 applications uses the same implementation with the same search-and-replace features. However, VBScript and VB strings don't support `\xFF` and `\uFFFF` escapes.
- Python: The regex flavor supported by Python's built-in `re` module.
- Ruby: The regex flavor built into the Ruby programming language.
- Tcl: The regex flavor used by the `regsub` command in Tcl 8.2 and 8.4, dubbed Advanced Regular Expressions in the Tcl man pages. `wxWidgets` uses the same flavor.
- PHP `ereg`: The replacement text syntax used by the `ereg_replace` and `eregi_replace` functions in PHP.
- PHP `preg`: The replacement text syntax used by the `preg_replace` function in PHP.
- REALbasic: The replacement text syntax used by the `ReplaceText` property of the `Regex` class in REALbasic.
- Oracle: The replacement text syntax used by the `REGEXP_REPLACE` function in Oracle Database 10g.
- Postgres: The replacement text syntax used by the `regexp_replace` function in PostgreSQL.
- XPath: The replacement text syntax used by the `fn:replace` function in XQuery and XPath.
- R: The replacement text syntax used by the `sub` and `gsub` functions in the R language. Though R supports three regex flavors, it has only one replacement syntax for all three.

Syntax Using Backslashes

Feature: `\&` (whole regex match)

Supported by: JGsoft, Ruby, Postgres

Feature: `\0` (whole regex match)

Supported by: JGsoft, Ruby, Tcl, PHP `ereg`, PHP `preg`, REALbasic

Feature: `\1` through `\9` (backreference)

Supported by: JGsoft, Perl, Python, Ruby, Tcl, PHP `ereg`, PHP `preg`, REALbasic, Oracle, Postgres, R

Feature: `\10` through `\99` (backreference)

Supported by: JGsoft, Python, PHP `preg`, REALbasic

Feature: `\10` through `\99` treated as `\1` through `\9` (and a literal digit) if fewer than 10 groups

Supported by: JGsoft

Feature: `\g<group>` (named backreference)

Supported by: JGsoft, Python

Feature: `\`` (backtick; subject text to the left of the match)

Supported by: JGsoft, Ruby

Feature: `\'` (straight quote; subject text to the right of the match)

Supported by: JGsoft, Ruby

Feature: `\+` (highest-numbered participating group)

Supported by: JGsoft, Ruby

Feature: Backslash escapes one backslash and/or dollar
 Supported by: JGsoft, Java, Perl, Python, Ruby, Tcl, PHP ereg, PHP preg, REALbasic, Oracle, Postgres, XPath, R

Feature: Unescaped backslash as literal text
 Supported by: JGsoft, .NET, Perl, JavaScript, Python, Ruby, Tcl, PHP ereg, PHP preg, Oracle, Postgres

Character Escapes

Feature: `\u0000` through `\uFFFF` (Unicode character)
 Supported by: JGsoft, Java, JavaScript, Python, Tcl, R

Feature: `\x{0}` through `\x{FFFF}` (Unicode character)
 Supported by: JGsoft, Perl

Feature: `\x00` through `\xFF` (ASCII character)
 Supported by: JGsoft, Perl, JavaScript, Python, Tcl, PHP ereg, PHP preg, REALbasic, R

Syntax Using Dollar Signs

Feature: `&` (whole regex match)
 Supported by: JGsoft, .NET, Perl, JavaScript, REALbasic

Feature: `0` (whole regex match)
 Supported by: JGsoft, .NET, Java, PHP preg, REALbasic, XPath

Feature: `1` through `9` (backreference)
 Supported by: JGsoft, .NET, Java, Perl, JavaScript, PHP preg, REALbasic, XPath

Feature: `10` through `99` (backreference)
 Supported by: JGsoft, .NET, Java, Perl, JavaScript, PHP preg, REALbasic, XPath

Feature: `10` through `99` treated as `1` through `9` (and a literal digit) if fewer than 10 groups
 Supported by: JGsoft, Java, JavaScript, XPath

Feature: `{1}` through `{99}` (backreference)
 Supported by: JGsoft, .NET, Perl, PHP preg

Feature: `{group}` (named backreference)
 Supported by: JGsoft, .NET

Feature: ``` (backtick; subject text to the left of the match)
 Supported by: JGsoft, .NET, Perl, JavaScript, REALbasic

Feature: `'` (straight quote; subject text to the right of the match)
 Supported by: JGsoft, .NET, Perl, JavaScript, REALbasic

Feature: \$_ (entire subject string)
Supported by: JGsoft, .NET, Perl

Feature: \$+ (highest-numbered participating group)
Supported by: JGsoft, Perl

Feature: \$+ (highest-numbered group in the regex)
Supported by: .NET

Feature: \$\$ (escape dollar with another dollar)
Supported by: JGsoft, .NET, JavaScript

Feature: \$ (unescaped dollar as literal text)
Supported by: JGsoft, .NET, JavaScript, Python, Ruby, Tcl, PHP ereg, PHP preg, Oracle, Postgres, R

Tokens Without a Backslash or Dollar

Feature: & (whole regex match)
Supported by: Tcl

General Replacement Text Behavior

Feature: Backreferences to non-existent groups are silently removed
Supported by: JGsoft, Perl, Ruby, Tcl, PHP preg, REALbasic, Oracle, Postgres, XPath

Highest-Numbered Capturing Group

The \$+ token is listed twice, because it doesn't have the same meaning in the languages that support it. It was introduced in Perl, where the \$+ variable holds the text matched by the highest-numbered capturing group that actually participated in the match. In several languages and libraries that intended to copy this feature, such as .NET and JavaScript, \$+ is replaced with the highest-numbered capturing group, whether it participated in the match or not.

E.g. in the regex «a(\d)|x(\w)» the highest-numbered capturing group is the second one. When this regex matches „a4”, the first capturing group matches „4”, while the second group doesn't participate in the match attempt at all. In Perl, \$+ will hold the „4” matched by the first capturing group, which is the highest-numbered group that actually participated in the match. In .NET or JavaScript, \$+ will be substituted with nothing, since the highest-numbered group in the regex didn't capture anything. When the same regex matches „xy”, Perl, .NET and JavaScript will all store „y” in \$+.

Also note that .NET numbers named capturing groups after all non-named groups. This means that in .NET, \$+ will always be substituted with the text matched by the last named group in the regex, whether it is followed by non-named groups or not, and whether it actually participated in the match or not.

Index

- `$`. *see* dollar sign
- `[`. *see* square bracket
- `]`. *see* square bracket
- `\`. *see* backslash
- `^`. *see* caret
- `..` *see* dot
- `|`. *see* vertical bar
- `?`. *see* question mark
- `*`. *see* star
- `+`. *see* plus
- `(`. *see* round bracket
- `)`. *see* round bracket
- `\t`. *see* tab
- `\r`. *see* carriage return
- `\n`. *see* line feed
- `\a`. *see* bell
- `\e`. *see* escape
- `\f`. *see* form feed
- `\v`. *see* vertical tab
- `\d`. *see* digit
- `\D`. *see* digit
- `\w`. *see* word character
- `\W`. *see* word character
- `\s`. *see* whitespace
- `\S`. *see* whitespace
- `\.` *see* start file
- `\'` *see* end file
- `\b`. *see* word boundary
- `\y`. *see* word boundary
- `\m`. *see* word boundary
- `\<`. *see* word boundary
- `\>`. *see* word boundary
- `{`. *see* curly braces
- `\1`. *see* backreference
- `\G`. *see* previous match
- `\c`. *see* control characters *or* XML names
- `\C`. *see* control characters *or* XML names
- `\i`. *see* XML names
- `\I`. *see* XML names
- .bak files, 190
- .exe, 199
- .war, 200
- .xps files, 36, 198
- Abort, 143
- Action menu, 141
- Action panel, 105
- action preferences, 201
- Action to Sequence, 150
- action type, 107
- adapt case of replacement text, 120
- Add to Library, 142, 149
- all search terms, 29, 125
- alnum, 316
- alpha, 316
- alphabet, 213
- alternation, 273
- always skip, 193
- anchor, 127, 266, 295, 302, 308
- and, 29
- any character, 264
- Apache logs, 72, 75, 77
- archive formats, 197
- archives, 94, 100
- arrange panels, 182
- detect ascii files using `\uffff`, ``; or ``, 215
- ASCII, 213, 257, 316
- assertion, 302
- Assistant, 89
- assistant font, 224
- asterisk. *see* star
- Auto Indent, 175
- Automatic Update, 161
- awk, 258
- `\b`. *see* word boundary
- backreference, 279
 - in a character class, 283
 - number, 280
 - repetition, 343
- backslash, 256, 257
 - in a character class, 260
- backtick, 267
- backtracking, 277, 337
- backup, 138, 243
- backup file destination type, 139
- backup file location, 139
- backup file naming style, 139
- backup files, 190, 216
- backup type, 138
- bak files, 190
- begin file, 127, 267
- begin line, 266
- begin string, 127, 266

- bell, 257
- between collected text, 134
- binary data, 114
- binary file, 214
- binary files, 100, 213
- blank, 316
- bookmarks, 163
- boolean, 29
- braces. *see* curly braces
- bracket. *see* square bracket *or* parenthesis
- bracket expressions, 315
- byte order marker, 213
- \c. *see* control characters *or* XML names
- \C. *see* control characters *or* XML names
- canonical equivalence
 - Java, 294
- capturing group, 279
- caret, 127, 256, 266, 295
 - in a character class, 260
- carriage return, 120, 257
- case adaptive, 120
- case conversion, 235
- case insensitive, 120, 295
- case sensitive, 120
- catastrophic backtracking, 337
- character class, 93, 260
 - negated, 260
 - negated shorthand, 262
 - repeating, 263
 - shorthand, 261
 - special characters, 260
 - subtract, 313
 - XML names, 313
- character equivalents, 318
- character range, 260
- character set. *see* character class
- character sets, 212
- characters, 256
 - ASCII, 257
 - categories, 287
 - control, 257
 - digit, 261
 - in a character class, 260
 - invisible, 257
 - metacharacters, 256
 - non-printable, 257
 - non-word, 261, 270
 - special, 256
 - Unicode, 257, 286
 - whitespace, 261
 - word, 261, 270
- charset meta tag, 215
- choice, 273
- class, 260
- Clean History, 178
- Clear
 - Action, 141
 - File Selector, 98
 - Results, 160
 - Sequence, 148
- Clear File or Folder, 7, 99
- Clear Folder and its Files and Subfolders, 99
- Clear Sequence Results, 151
- Clear Step Results, 151
- closing bracket, 289
- closing quote, 289
- cntrl, 316
- code page, 344, 345
- code pages, 212
- code point, 287
- collapse
 - Editor, 173
 - Results, 163
- collating sequences, 318
- collect, 108, 241
- collect data, 108
- collect headers and footers, 135
- collect whole sections, 125
- combining character, 288
- combining mark*, 286
- combining multiple regexes, 273
- comma-delimited, 64, 66
- command line, 240
 - backup, 243
 - collect, 241
 - context, 242
 - contextextra, 242
 - delete, 241
 - delimitprefix, 241
 - delimitreplace, 242
 - delimitsearch, 242
 - execute, 245
 - file, 244
 - fileexclude, 244
 - find, 241
 - findname, 241
 - folder, 244
 - folderexclude, 244
 - folderrecurse, 244
 - literal, 241
 - masks, 244
 - merge, 241

- nocache, 246
- noundo, 246
- noundomanager, 246
- optadaptive, 242
- optarchives, 243
- optbinary, 243
- optcase, 242
- optdotall, 242
- optinvert, 242
- optnonoverlap, 242
- optwords, 242
- preview, 245
- quick, 245
- quit, 246
- regex, 241
- rename, 241
- replace, 241
- replacebytes, 241
- replacetext, 241
- resultsoptions, 244
- reuse, 246
- save, 245
- search, 240
- searchbytes, 241
- searchbytesfile, 241
- searchtext, 241
- searchtextfile, 241
- silent, 246
- simple, 240
- split, 241
- target, 242
- comments, 105, 319, 320
- compatibility, 251
- compressed file formats supported by powergrep, 197
- compressed files, 197
- concurrent search, 116, 127
- condition
 - if-then-else, 310
- conditions
 - many in one regex, 306
- content-type meta tag, 215
- context, 131, 242
- context type, 133
- contextextra, 242
- continue
 - from previous match, 308
- control characters, 257, 289
- conversion cache, 195
- conversion manager, 196
- copy all searched files, 136
- copy files, 137
- Copy Files
 - File Selector, 103
 - Results, 166
- copy matching files, 136
- copy only modified files, 136
- copyright, 45
- count all context, 133
- CR, 120
- cross. *see plus*
- CSV files, 64, 66
- ctrl+wheel scrolls whole pages instead of
 - zooming, 219
- curly braces, 276
- currency sign, 288
- Custom Layouts, 183
- \d. *see digit*
- \D. *see digit*
- dash, 289
- data, 251
- date, 330
 - file, 94, 202
- decode and convert to text prior to searching, 193
- decode binary files using ifilter, 194
- decompressing files, 197
- default editor, 222
- default layout, 182
- define masks, 93
- delete, 241
- Delete Action, 179
- Delete Backup Files, 179
- Delete Files
 - File Selector, 102
 - Results, 165
- Delete Item, 156
- delete matching files, 136, 202
- Delete Step, 149
- delimited binary data, 116
- delimited literal text, 116
- delimited regular expressions, 116
- delimitprefix, 116, 241
- delimitreplace, 116, 242
- delimitsearch, 116, 242
- DFA engine, 258
- digit, 261, 288, 316
- directory placeholders, 237
- display replacements, 159
- distance, 32, 336
- do not save results, 136
- do not section files, 124
- docked panel, 180
- DOCX files, 34, 198

- dollar, 295
- dollar sign, 127, 256, 266
- DOS, 213
- dot, 256, 264, 295
 - misuse, 338
 - newlines, 264
 - vs. negated character class, 265
- dot matches newlines, 120
- double quote, 257
- Dual Monitor Layout, 183
- duplicate lines, 335
- eager, 258, 273
- EBCDIC, 213
- Edit File
 - File Selector, 102
 - Results, 164
- editor
 - external, 221
- Editor, 168
- Editor menu, 171
- editor preferences, 218
- EditPad
 - File Selector, 102
 - Results, 165
- EditPad Pro, 252
- egrep, 258
- else, 310
- email address, 26, 327
- enclosing mark, 288
- encoding, 212
- end file, 127, 267
- end line, 266
- end of line, 257
- end string, 127, 266
- engine, 251, 258
- entire string, 266
- escape, 256, 257
 - in a character class, 260
- euro, 344
- example
 - copyright, 45
 - date, 330
 - delete lines, 39
 - duplicate lines, 335
 - email address, 26
 - exponential number, 326, 335
 - extract lines, 39
 - file names, 23
 - floating point number, 326
 - Google search terms, 74
 - HTML tags, 52, 323
 - HTML title, 49
 - integer number, 335
 - keywords, 335
 - not meeting a condition, 334
 - number, 335
 - prepend lines, 267
 - quoted string, 265
 - reserved words, 335
 - same line, 33
 - scientific number, 326, 335
 - source code, 60
 - trimming whitespace, 323
 - web logs, 72, 75, 77
 - whole line, 334
 - word pairs, 28
- Exclude File or Folder, 7, 99
- exclude files, 93, 190
- exclude files and folders using regular expressions
 - instead of file masks, 190
- exclude folders, 190
- execute, 245
- Execute, 142, 152
 - abort, 143
 - preview, 142, 151
 - quick, 143, 152
 - safe, 142, 151
- expand
 - Editor, 174
 - Results, 163
- expand to whole lines, 133
- Export, 161
- external editors preferences, 221
- extra context after the match, 132
- extra context before the match, 131
- extra processing, 130
- faster searches, 143, 152
- Favorites
 - Action, 141
 - Editor, 171
 - File Selector, 98
 - Library, 155
 - Results, 160
 - Sequence, 148
- feeds, 188
- FF, 120
- file, 244
 - exclude, 93, 190
 - include, 93
- file formats, 192, 247
- file listings, 96
- file masks, 93, 198
- file modification dates, 94, 202
- file names, 23

- file placeholders, 237
- file sectioning, 60, 124
- file selection preferences, 189
- File Selector, 7, 91
 - keyboard, 92
- File Selector menu, 98
- File Selector to Sequence, 149
- file sizes, 95
- file tree, 91
- fileexclude, 244
- files
 - hidden, 190
 - system, 190
- filter files, 121
- find, 241
- find all search terms, 29, 125
- findname, 241
- flavor, 251
- flex, 258
- floating panel, 180
- floating panels, 182
- floating point number, 326
- fold
 - Editor, 173
 - Results, 163
- fold files, 217
- folder, 244
 - exclude, 190
- folder placeholders, 237
- folder to use for temporary files, 203
- folder tree, 91
- folderexclude, 244
- folderrecurse, 244
- folders and files, 91
- follow keyboard focus and mouse pointer, 224
- follow keyboard focus only, 224
- form feed, 120, 257
- free-spacing, 114, 320
- full path, 216
- full stop. *see* dot
- getting started, 5
- Google search terms, 74
- graph, 316
- grapheme, 286
- greedy*, 275, 276
- group, 279
 - capturing, 279
 - in a character class, 283
 - named, 284
 - nested, 337
 - repetition, 343
- group identical matches, 107
- group results for all files, 108
- group search matches, 158
- hexadecimal, 213
- hidden files, 190
- history, 176
- HTML, 161
 - charset, 215
- HTML tags, 52, 323
- HTML title, 49
- HTTP proxy, 184
- hyphen, 289
 - in a character class, 260
- \i. *see* XML names
- \I. *see* XML names
- IBM mainframes, 213
- icons, 182
- if-then-else, 310
- ignore whitespace, 114, 320
- import file listings, 96
- Include File or Folder, 7, 99
- include files, 93
- Include Folder and Subfolders, 7, 99
- Insert File, 116
- instances, 154
- integer number, 335
- invert search results, 25, 124
- invisible characters, 257
- ISO-8859, 213
- keywords, 335
- LAN, 91
- Large Toolbar Icons, 182
- layout, 182
- lazy, 277
 - better alternative, 277
- leftmost match, 258
- letter, 288, *see* word character
- lex, 258
- LF, 120
- library, 142, 149
- Library, 153
- Library menu, 155
- line, 266
 - begin, 266
 - duplicate, 335
 - end, 266
 - not meeting a condition, 334
 - prepend, 267
- line break, 120, 257, 295
- line by line, 124
- line feed, 120, 257
- Line Numbers, 175

- line separator, 288
- line terminator, 257
- Linux, 213
- list of binary data, 115
- list of literal text, 115
- list of regular expressions, 115
- list only files matching all terms, 29, 107
- list only sections matching all items, 29
- list only sections matching all terms, 125
- listings
 - files, 96
- literal, 114, 241
- literal characters, 256
- literal text, 114
- local area network, 91
- locale, 315
- Lock Toolbars, 182
- log files, 72, 75, 77, 78
- lookahead, 302
- lookaround, 302
 - many conditions in one regex, 306
- lookbehind, 303
 - limitations, 303
- lower, 317
- lowercase, 120, 235
- lowercase letter, 288
- \m. *see* word boundary
- Make Replacement
 - Editor, 173
 - Results, 162
- many conditions in one regex, 306
- mark, 288
- Mark Files with Search Results, 100
- masks, 93, 244
- match, 251
- match all search terms, 29, 125
- match mode, 295
- match placeholders, 67, 114, 202, 229
- match whole sections only, 125
- matches to delete, 112
- matching files, 136
- mathematical symbol, 288
- maximum length of a line of context, 217
- maximum memory usage to display results, 217
- merge, 241
- merge based on search matches, 137
- merge into a single file, 137
- metacharacters, 256
 - in a character class, 260
- Microsoft Office Open XML files, 34, 198
- minimum number of execution threads, 203
- minimum number of occurrences, 108
- mode modifier, 295
- mode span, 296
- modification dates, 94, 202
- modifier, 295
- modifier span, 296
- modify original files, 136, 137
- Move Files
 - File Selector, 104
 - Results, 167
- move matching files, 136
- Move Step Down, 149
- Move Step Up, 149
- MRU, 98, 141, 148, 160
- MS-DOS, 213
- multi-line, 295
- multi-line mode, 266
- multi-monitor, 180
- multiple instances, 154
- multiple regexes combined, 273
- My Documents folder, 191
- MySQL, 258
- named group, 284
- narrow down search, 101
- near, 32, 336
- negated character class, 260
- negated shorthand, 262
- negative lookahead, 302
- negative lookbehind, 303
- nested grouping, 337
- network, 91, 190
- never use ifilter for files matching these file masks, 195
- New
 - Editor, 171
 - Library, 155
- new instance, 189
- New Step, 149
- newline, 120, 264, 295
- news feeds, 188
- next and previous buttons should also select the match in the results, 218
- next and previous match buttons can advance to the next or previous file, 218
- Next File
 - Editor, 172
 - Results, 162
- Next Match
 - Editor, 172
 - Results, 161
- NFA engine, 258
- no context, 131
- nocache, 246

- non-overlapping search, 116, 127
- non-printable characters, 257
- non-spacing mark, 288
- not, 29
- noundo, 246
- noundomanager, 246
- NULL, 100
- NULL characters, 213
- number, 261, 288, 335
 - backreference, 280
 - exponential, 326, 335
 - floating point, 326
 - scientific, 326, 335
- Office 2003 Display Style, 182
- Office Open XML files, 34, 198
- once or more, 276
- only use ifilter for files matching these file masks, 194
- Open
 - Action, 141
 - Editor, 171
 - File Selector, 98
 - Library, 155
 - Results, 160
 - Sequence, 148
- Open Action, 150
- open all files command for this editor, 222
- Open File in EditPad
 - File Selector, 102
 - Results, 165
- Open File Selection, 150
- OpenDocument Format, 37, 198
- opening bracket, 289
- opening quote, 289
- OpenOffice files, 37, 198
- optadaptive, 120, 242
- optarchives, 100, 243
- optbinary, 100, 243
- optcase, 120, 242
- optdotall, 120, 242
- optinvert, 124, 242
- option, 273, 275, 276
- optnonoverlap, 116, 242
- optwords, 120, 242
- or, 29
 - one character or another, 260
 - one regex or another, 273
- order of collected matches, 138
- overlapping search, 116, 127
- overwrite files that have the read-only attribute set, 202
- pad, 66
- padding, 235
- page breaks, 44
- paragraph separator, 288
- parenthesis. *see* round bracket
- path
 - full, 216
 - relative, 216
- Path field, 15, 92
- path placeholders, 114, 137, 202, 237
- pattern, 251
- Pause, 143
- period. *see* dot
- permanent exclusion, 190
- PGA files, 141, 148, 247
- PGF files, 98, 247
- PGL files, 155, 247
- PGR files, 160, 247
- PGU files, 178, 247
- pipe symbol. *see* vertical bar
- placeholders, 229, 237
- plus, 256, 276
 - possessive quantifiers, 297
- positive lookahead, 302
- positive lookbehind, 303
- POSIX, 315
- possessive, 297
- post processing, 130
- PowerGREPConversionManager.exe, 196
- PowerGREPUndoManager.exe, 177
- precedence, 273, 279
- preferences
 - action, 201
 - editor, 218
 - external editors, 221
 - file selection, 189
 - regex colors, 225
 - results, 216
 - results colors, 226
 - syntax colors, 227
 - text encoding, 212
- prepend lines, 267
- preview, 245
- Preview, 142, 151
- Previous File
 - Editor, 172
 - Results, 162
- previous match, 308
- Previous Match
 - Editor, 172
 - Results, 161
- print, 317

- Print
 - Editor, 172
 - Results, 161
- processing, 130
- Procmail, 258
- progress meter, 157
- properties
 - Unicode, 287
- proprietary file formats, 192
- proxy, 184
- punct, 317
- punctuation, 289
- quantifier
 - backreference, 343
 - backtracking, 277
 - curly braces, 276
 - greedy, 276
 - group, 343
 - lazy, 277
 - nested, 337
 - once or more, 276
 - once-only, 297
 - plus, 276
 - possessive, 297
 - question mark, 275
 - reluctant, 277
 - specific amount, 276
 - star, 276
 - ungreedy, 277
 - zero or more, 276
 - zero or once, 275
- question mark, 256, 275
 - common mistake, 326
 - lazy quantifiers, 277
- quick, 245
- Quick Execute, 143, 152
- Quick Replace, 143, 162, 173
- quick start, 16
- quit, 246
- quote, 257
- quoted string, 265
- range of characters, 260
- recent files, 98, 141, 148, 160
- recursion, 99
- recycle bin, 202
- Refresh, 102
- regex, 114, 241
- regex color configuration, 225
- regex engine, 258
- regex-directed engine, 258
- regular expression, 114, 251
- relative path, 216
- reluctant, 277
- remember search terms, 189
- rename, 241
- rename or move files, 137
- repetition
 - backreference, 343
 - backtracking, 277
 - curly braces, 276
 - greedy, 276
 - group, 343
 - lazy, 277
 - nested, 337
 - once or more, 276
 - once-only, 297
 - plus, 276
 - possessive, 297
 - question mark, 275
 - reluctant, 277
 - specific amount, 276
 - star, 276
 - ungreedy, 277
 - zero or more, 276
 - zero or once, 275
- replace, 241
- replace whole sections, 125
- replacebytes, 116, 241
- replacetext, 116, 241
- requirements
 - many in one regex, 306
- reserved characters, 256
- Restore Default Layout, 182
- results
 - hexadecimal, 213
- Results, 157
- results color configuration, 226
- Results menu, 160
- results preferences, 216
- resultsoptions, 157, 244
- Resume, 143
- reuse, 246
 - part of the match, 279
- Revert Replacement
 - Editor, 173
 - Results, 162
- round bracket, 256, 279
- RSS feeds, 188
- RTF, 161
- \s. *see* whitespace
- \S. *see* whitespace
- same masks, 93, 94
- save, 160, 245

Save

- Action, 141
- Editor, 171
- File Selector, 98
- Library, 155
- Results, 160
- Sequence, 148

Save Action, 151

Save As, 171

Save File Selection, 150

save list of matching files, 136

save one file for each searched file, 137

save results, 136, 137

Save Results, 151

sawtooth, 342

script, 289

search, 107, 240

- search and collect sections, 61, 126
- search and replace, 252
- search for sections, 60, 126
- search item delimiter, 116

Search Only through Files with Results, 101

- search pair delimiter, 116
- search prefix label delimiter, 116
- search subfolders with powergrep, 223
- search terms, 114
 - load from file, 116

Search through Archives, 100

Search through Binary Files, 100

search through raw binary data, 193

search through raw binary data when a file cannot be decoded, 193

search through the file's raw xml content, 193

search with powergrep, 223

searchbytes, 114, 241

searchbytesfile, 241

searchtext, 114, 241

searchtextfile, 241

second monitor, 180

section collect, 126

section files, 124

Select History, 178

select text to edit, 135

self-extracting archives, 199

send to menu shortcut, 223

send to menu shortcut that includes subfolders, 224

sensitive to case, 120

separator, 288

Sequence menu, 148

Sequence to Action, 150

Sequence to File Selector, 150

Sequence to Results, 151

several conditions in one regex, 306

SFX, 199

shorthand character class, 261

- negated, 262
- XML names, 313

Show All, 101

Show Files with Results, 101

Show Folders, 101

Show Included Files, 101

show line numbers, 132

Side by Side Layout, 182

silent, 246

simple, 240

single quote, 257, 267

single-line, 295

single-line mode, 264

size

- file, 95

skipped binary files, 216

SOCKS proxy, 184

sort collected matches, 108

source code

- comments, 60
- strings, 60

space, 317

space separator, 288

spacing combining mark, 288

special characters, 256

- in a character class, 260
- in programming languages, 257

specific amount, 276

split, 241

split along delimiters, 60, 126

spreadsheets, 64

square bracket, 256, 260

star, 256, 276

- common mistake, 326

start file, 127, 267

start line, 266

start string, 127, 266

step, 149

string, 251

- begin, 127, 266
- end, 127, 266
- matching entirely, 266
- quoted, 265

subfolders, 99

subtract character class, 313

surrogate, 289

symbol, 288

synchronized, 154

- syntax color configuration, 227
- system files, 190
- tab, 257
- tab character, 15
- tab size, 219
- tabbed panel, 180
- target, 136, 242
- target file destination type, 137
- target file line break style, 138
- target file location, 137
- target file text encoding, 138
- target type, 136
- task, 105
- Tcl
 - word boundaries, 271
- terminate lines, 257
- text, 251
- text editor, 252
- text encoding, 212, 344
 - HTML, 215
 - XML, 214
- text-directed engine, 258
- time
 - file, 202
- titlecase letter, 288
- toolbar
 - icons, 182
- toolbars
 - lock, 182
- transform plain text files using ifilter, 194
- tree of folders and files, 91
- trimming whitespace, 323
- tutorial, 251
- underscore, 261
- Undo Action, 178
- Undo History, 176
- Undo History menu, 178
- undo manager, 177
- undoable, 176
- unfold
 - Editor, 174
 - Results, 163
- ungreedy, 277
- Unicode, 212, 286
 - blocks, 290
 - canonical equivalence, 294
 - categories, 287
 - characters, 286
 - code point, 287
 - combining mark*, 286
 - grapheme, 286
 - Java, 293
 - normalization, 294
 - Perl, 293
 - properties, 287
 - ranges, 290
 - scripts, 289
- unpad, 66
- Update Results, 161
- upper, 317
- uppercase, 120, 235
- uppercase letter, 288
- use a separate thread for each drive letter, 203
- use a separate thread for each network share, 203
- Use Action, 178
- use ifilter to decode the compound document into plain text, 198
- use ifilter, if available, instead of powergrep's built-in decoder, 193
- Use Item, 156
- Use Item in Sequence, 156
- use lines as context, 131
- UTF-16, 213
- UTF-32, 213
- UTF-8, 213
- vertical bar, 256, 273
- vertical tab, 120, 257
- View Action, 181
- View Assistant, 180
- View Editor, 181
- View File Selector, 180
- View Forum, 182
- View Library, 181
- View Results, 181
- View Sequence, 181
- View Undo History, 181
- visualize line breaks, 202, 219
- visualize spaces and tabs, 201, 219
- VT, 120
- \w. *see* word character
- \W. *see* word character
- web archives, 200
- web logs, 72, 75, 77
- whitespace, 261, 288, 323
 - ignore, 114, 320
 - padding, 66
- whole line, 266, 334
- whole sections, 125
- whole word, 270, 271
- whole words only, 120
- wild cards. *see* file masks *or* regular expressions
- Windows code page, 213
- word, 270, 271, 317

- word boundary, 270
 - Tcl, 271
- word character, 261, 270
- word pairs, 28
- Word Wrap
 - Editor, 174
 - Results, 164
- words
 - keywords, 335
- working copy, 190
- xdigit, 317

- XML declaration, 214
- XML names, 313
- XML Paper Specification files, 36, 198
- XML schema, 247
- XPS files, 36, 198
- \y. *see* word boundary
- zero or more, 276
- zero or once, 275
- zero-length match, 267
- zero-width, 266, 302