Just Great Software

PowerGREP

Version 5.1.5 — 16 May 2018

Published by Just Great Software Co. Ltd.

Table of Contents

Regular Expression Reference......437

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.

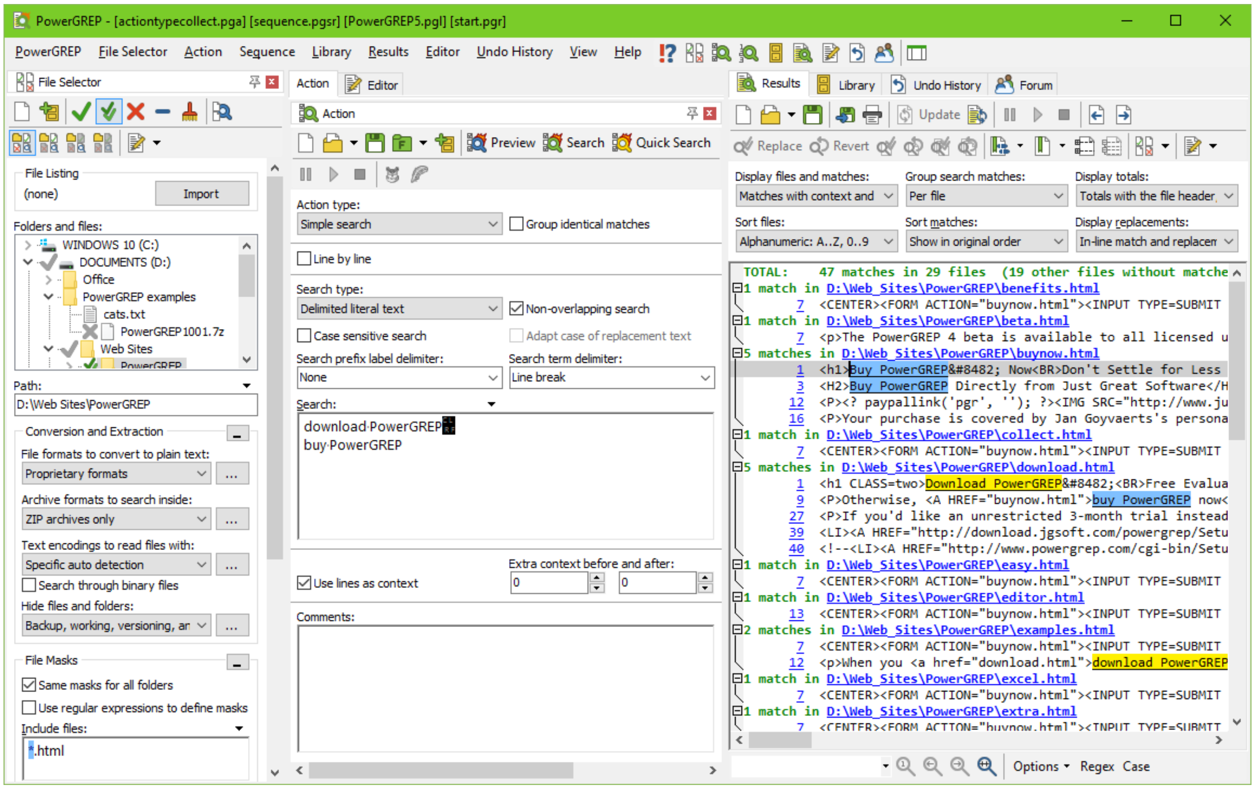
1. You start with telling PowerGREP which files you want to work with. Click on a file or folder in the File Selector. Then select Include File or Folder in the File Selector menu to mark the file or folder to be searched through. Marking a folder is a quick way to work with all the files in that folder. 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.



File Listing
(none)

Folders and files:
- WINDOWS 10 (C:)
- DOCUMENTS (D:)
  - Office
  - PowerGREP examples
    - cats.txt
    - PowerGREP1001.7z
  - Web Sites
    - PowerGREP

Path: D:\Web Sites\PowerGREP

Conversion and Extraction

File formats to convert to plain text: Proprietary formats

Archive formats to search inside: ZIP archives only

Text encodings to read files with: Specific auto detection

☐ Search through binary files

Hide files and folders: Backup, working, versioning, ar

File Masks

☑ Same masks for all folders

☐ Use regular expressions to define masks

Include files: *.html

Action type: Simple search

☐ Group identical matches

☐ Line by line

Search type: Delimited literal text

☑ Non-overlapping search

☐ Case sensitive search

☐ Adapt case of replacement text

Search prefix label delimiter: None

Search term delimiter: Line break

Search:

```
download PowerGREP
buy PowerGREP
```

☑ Use lines as context

Extra context before and after: 0 0

Comments:

Display files and matches: Matches with context and

Group search matches: Per file

Display totals: Totals with the file header

Sort files: Alphanumeric: A..Z, 0..9

Sort matches: Show in original order

Display replacements: In-line match and replacem

```
TOTAL:     47 matches in 29 files  (19 other files without matche
⊟1 match in D:\Web Sites\PowerGREP\benefits.html
        7  <CENTER><FORM ACTION="buynow.html"><INPUT TYPE=SUBMIT
⊟1 match in D:\Web Sites\PowerGREP\beta.html
        7  <p>The PowerGREP 4 beta is available to all licensed u
⊟5 matches in D:\Web Sites\PowerGREP\buynow.html
        1  <h1>Buy PowerGREP&#8482; Now<BR>Don't Settle for Less
        3  <H2>Buy PowerGREP Directly from Just Great Software</H
       12  <P><? paypallink('pgr', ''); ?><IMG SRC="http://www.ju
       16  <P>Your purchase is covered by Jan Goyvaerts's persona
⊟1 match in D:\Web Sites\PowerGREP\collect.html
        7  <CENTER><FORM ACTION="buynow.html"><INPUT TYPE=SUBMIT
⊟5 matches in D:\Web Sites\PowerGREP\download.html
        1  <h1 CLASS=two>Download PowerGREP&#8482;<BR>Free Evalua
        9  <P>Otherwise, <A HREF="buynow.html">buy PowerGREP now<
       27  <P>If you'd like an unrestricted 3-month trial instead
       39  <LI><A HREF="http://download.jgsoft.com/powergrep/Setu
       40  <!--<LI><A HREF="http://www.powergrep.com/cgi-bin/Setu
⊟1 match in D:\Web Sites\PowerGREP\easy.html
        7  <CENTER><FORM ACTION="buynow.html"><INPUT TYPE=SUBMIT
⊟1 match in D:\Web Sites\PowerGREP\editor.html
       13  <CENTER><FORM ACTION="buynow.html"><INPUT TYPE=SUBMIT
⊟2 matches in D:\Web Sites\PowerGREP\examples.html
        7  <CENTER><FORM ACTION="buynow.html"><INPUT TYPE=SUBMIT
       12  <p>When you <a href="download.html">download PowerGREP
⊟1 match in D:\Web Sites\PowerGREP\excel.html
        7  <CENTER><FORM ACTION="buynow.html"><INPUT TYPE=SUBMIT
⊟1 match in D:\Web Sites\PowerGREP\extra.html
        7  <CENTER><FORM ACTION="buynow.html"><INPUT TYPE=SUBMIT
```

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. By default, "hide files and folders" is set to "backup, working, versioning, and hidden". This makes files that look like backup copies, working copies or version control copies as well as hidden files completely invisible to PowerGREP and its File Selector. If you need to search through such files, select a different "hide files and folders" configuration first. Otherwise, leave this option unchanged as it reduces clutter in the folders and files tree, and makes sure you don't accidentally delete PowerGREP's own backup copies that its Undo History depends on.

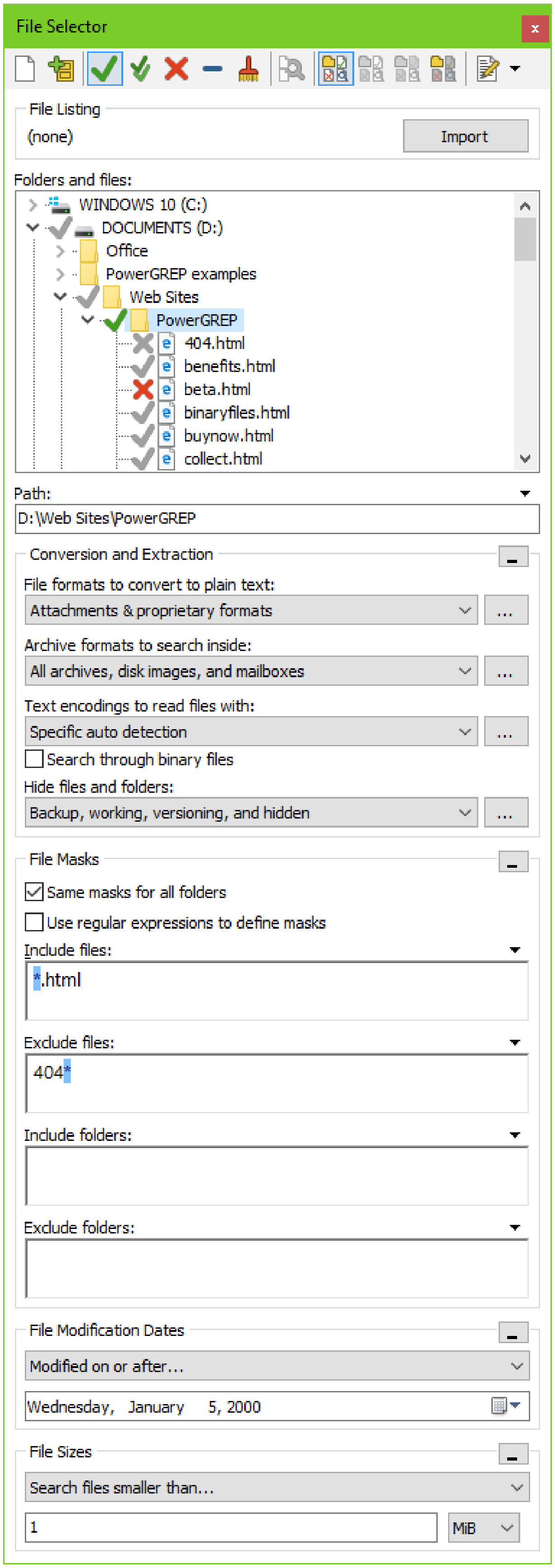
2. If you already have a text file with a list of files and/or folders that you want search through, click the Import button to show the Import File Listings screen. There you can select one or more text files to read file listings from. Then you can skip ahead to step 9.

3. 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.

4. 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.

5. 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.

6. 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.



File Selector

File Listing
(none)
Import

Folders and files:
- WINDOWS 10 (C:)
- DOCUMENTS (D:)
  - Office
  - PowerGREP examples
  - Web Sites
    - PowerGREP
      - 404.html
      - benefits.html
      - beta.html
      - binaryfiles.html
      - buynow.html
      - collect.html

Path: D:\Web Sites\PowerGREP

Conversion and Extraction
File formats to convert to plain text: Attachments & proprietary formats
Archive formats to search inside: All archives, disk images, and mailboxes
Text encodings to read files with: Specific auto detection
☐ Search through binary files
Hide files and folders: Backup, working, versioning, and hidden

File Masks
☑ Same masks for all folders
☐ Use regular expressions to define masks
Include files: *.html
Exclude files: 404*
Include folders:
Exclude folders:

File Modification Dates
Modified on or after…
Wednesday, January 5, 2000

File Sizes
Search files smaller than…
1 MiB

7. 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.

8. 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.

9. Certain file formats may need to be converted to plain text before they can be searched through in a meaningful way, or at all. Select a suitable configuration in “file formats to convert to plain text”. “Proprietary formats” converts formats like PDF or DOC that require conversion. “All formats” also converts formats like HTML or RTF that are more readable when converted. When searching through email, use one of these two choices if you don’t want to search through attachments. Otherwise, choose “attachments & proprietary formats” or “Attachments & all formats” instead. When doing a search-and-replace, you need to disable read-only converters. Do so by selecting “writeable proprietary formats”, “all writable formats”, “attachments & writable proprietary formats”, or “Attachments & all writable formats”. Alternatively, choose “None” to skip all files that can’t be searched through without conversion. The three “compound documents” configurations as well as the “(unused)” choice are for advanced users.

10. If your files may be stored in archives such as ZIP files, mailboxes such as MBOX or PST files, or disk images such as ISO files, set “archive formats to search inside” to a configuration that enables searching inside archives, mailboxes, and/or disk images. PowerGREP then treats files in enabled archive, mailbox, and disk image formats as folders. Files in disabled formats are skipped. Select “None” if you want to skip all such formats.

11. Since computers deal with numbers rather than with text, plain text files use one of various Unicode or legacy encodings or code pages to map those numbers to human-readable text. Set “text encodings to read files with” to “specific auto detection” to have PowerGREP detect the appropriate encoding for each file as much as possible. If you have text files saved by old DOS or mainframe applications or on Windows computers using a different default code page than your own, you will likely need to edit this configuration or create a new one to specify which code pages those files use.

12. Files that look like they don’t contain human-readable text are skipped unless you turn on “search through binary files”. You can have all files treated as binary by turning on that option and setting “text encodings to read files with” to “all files as binary”. If you do that, then you probably want to set “file formats to convert to plain text” to “(unused)”.

13. If you want to search through files of particular types only, enter a semicolon-delimited list of file types in the “include files” box. Enter `*.txt;*.html`, for example, to search through text files and HTML files only. All other files lose their gray tick marks in the folders and files tree. To exclude certain types of files, enter their file types in the “exclude files” box. Those files get gray X marks in the folders and files tree. The File Selector reference explains the file masks you can use in the include files and exclude files boxes in full detail.

14. 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.

15. 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 Based on 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.
- count matches: Count matches per search term. Different matches of the same regex are all counted towards that regex. Search terms without matches are also indicated in the results.
- 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.
- file or folder name search: Search through the names of files and/or folders, rather than through the contents of files. Allows you to copy, move, delete, zip, and unzip the listed files via the target file creation setting.
- file or folder name collect: Collect a piece of text based on a search match found in the file or folder's name using replacement text syntax and path placeholders. Create a new file with all the collected file name parts.
- rename files or folders: 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”, “file or folder name search”, or “file or folder name collect”, PowerGREP can save the file names of the files that are found into a target file. If the action type is “search and replace” or “search and delete”, 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 saves 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 appears on the Results panel. Unless you used the Quick Execute button, the results are 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 (grouping or not) alphabetically, alphanumerically, or by the number of times each unique match was found. Alphanumeric sort puts "match2" before "match10" because 2 < 10. Alphabetic sort puts "match2" after "match10" because 2 sorts after 1.
- 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 higlighted: 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. They 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. F9, for example, 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. Ctrl+P, for example, is associated with Results|Print and Editor|Print. If you press Ctrl+P while inspecting the results, PowerGREP prints the results. If you press Ctrl+P while editing a file in the editor, PowerGREP prints the file you're editing. If you press Ctrl+P while selecting files or editing the action definition, nothing happens.

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. 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.

You can also directly focus a few controls via the keyboard. These keyboard shortcuts focus a control, showing and activating the panel it sits on if necessary. Alt+S focuses the Search box on the Action panel. Alt+I focuses the Include Files box on the File Selector panel.

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. You can focus the path field by pressing Alt+2 followed by Tab.

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 Expressions Quick Start

This quick start gets you up to speed quickly with regular expressions. Obviously, this brief introduction cannot explain everything there is to know about regular expressions. For detailed information, consult the regular expressions 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.

Many applications and programming languages have their own implementation of regular expressions, often with slight and sometimes with significant differences from other implementations. When two applications use a different implementation of regular expressions, we say that they use different "regular expression flavors". This quick start explains the syntax supported by the most popular regular expression flavors.

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 highlighted in red as `regex`. This is actually a perfectly valid regex. It is the most basic pattern, simply matching the literal text `regex`. Matches are highlighted in blue in this book. We use the term "string" to indicate the text that the regular expression is applied to. Strings are highlighted in `green`.

Characters with special meanings in regular expressions are highlighted in various different colors. The regex `(?x)([Rr]egexp?)\?` shows meta tokens in purple, grouping in green, character classes in orange, quantifiers and other special tokens in blue, and escaped characters in gray.

Literal Characters

The most basic regular expression consists of a single literal character, such as `a`. It matches the first occurrence of that character in the string. If the string is `Jack is a boy`, it matches the `a` after the `J`.

This regex can match the second `a` too. It only does 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.

Twelve characters have special meanings in regular expressions: 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 parenthesis `(`, the closing parenthesis `)`, the opening square bracket `[`, and the opening curly brace `{`. These special characters are often called "metacharacters". Most of them are errors when used alone.

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 has 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` does 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 negates the character class. The result is that the character class matches any character that is *not* in the character class. `q[^x]` matches `qu` in `question`. 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. In modern applications, they include non-English letters and numbers.

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`.

If your application supports Unicode, use `\uFFFF` or `\x{FFFF}` to insert a Unicode character. `\u20AC` or `\x{20AC}` matches the euro currency sign.

If your application does not support Unicode, use `\xFF` to match a specific character by its hexadecimal index in the character set. `\xA9` matches the copyright symbol in the Latin-1 character set.

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. Most applications 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` matches `cat` in `About cats and dogs`. If the regex is applied again, it matches `dog`. You can add as many alternatives as you want: `cat|dog|mouse|fish`.

Alternation has the lowest precedence of all regex operators. `cat|dog food` matches `cat` or `dog food`. To create a regex that matches `cat food` or `dog food`, you need to group the alternatives: `(cat|dog) food`.

Repetition

The question mark makes the preceding token in the regular expression optional. `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 expand the match as far as they can, and only give back if they must to satisfy the remainder of the regex. The regex `<.+>` matches `<EM>first</EM>` in `This is a <EM>first</EM> test`.

Place a question mark after the quantifier to make it lazy. `<.+?>` matches `<EM>` 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 Capturing

Place parentheses around multiple tokens to group them together. You can then apply a quantifier to the group. E.g. `Set(Value)?` matches `Set` or `SetValue`.

Parentheses create a capturing group. The above example has one group. After the match, group number one contains nothing if `Set` was matched. It contains `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.

Backreferences

Within the regular expression, you can use the backreference `\1` to match the same text that was matched by the capturing group. `([abc])=\1` matches `a=a`, `b=b`, and `c=c`. It does not match anything else. If your regex has multiple capturing groups, they are numbered counting their opening parentheses from left to right.

Named Groups and Backreferences

If your regex has many groups, keeping track of their numbers can get cumbersome. Make your regexes easier to read by naming your groups. `(?<mygroup>[abc])=\k<mygroup>` is identical to `([abc])=\1`, except that you can refer to the group by its name.

Unicode Properties

`\p{L}` matches a single character that is in the given Unicode category. `L` stands for letter. `\P{L}` matches a single character that is not in the given Unicode category. You can find a complete list of Unicode categories 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 lookahead and lookbehind.

Free-Spacing Syntax

Many application have an option that may be labeled "free-spacing" or "ignore whitespace" or "comments" that makes the regular expression engine ignore unescaped spaces and line breaks and that makes the # character start a comment that runs until the end of the line. This allows you to use whitespace to format your regular expression in a way that makes it easier for humans to read and thus makes it easier to maintain.

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 or folder 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. Set "archive formats to search inside" to "(unused)" if you want to search through the names of archives (treating them as ordinary files). Set it to "None" if you do not want to search through the names of archives. Set it to "All archives" if you want to search through the names of the files inside the archives (treating the archives as folders).
5. Start with a fresh action.
6. Set the action type to "file or folder name search".
7. Set "what to search through" to "file names only".
8. Enter one or more search terms to look for in the file names.
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 7. 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".

This file selection and action are available in the PowerGREP5.pgl library as "Search through file names".

Using The File Selector to Search Through File Names

For complex operations where you want to use an action type other than "file or folder name search" so that you can search through or 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 though 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. Set "archive formats to search inside" to "(unused)" if you want to search through the names of archives (treating them as ordinary files). Set it to "None" if you do not want to search through the names of archives. Set it to "All archives" if you want to search through the names of the files inside the archives (treating the archives as folders).
5. Turn on "use regular expressions to define masks", even if you want to search for a simple word or phrase.
6. Enter your search terms in the "include files" box, delimited by semicolons.
7. Start with a fresh action.
8. Set the action type to "list files".

9. Leave the Search box blank.
10. 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, enter the search terms from step 6 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 7 above to make the file selector search through the file names. You should not select "(unused)" in step 4 though. Treating archives as ordinary files does not produce proper search results when searching through the contents of files. Choose "None" to skip archives, or choose another configuration to search through the contents of the files inside (some) archives. Then proceed as follows:

8. Leave the action type as "simple search".
9. On the Action panel, enter the search terms you want to search for through the contents of the files.
10. 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 6. A file will only be listed in the results if both its name matches the terms of step 6, and its contents match the terms of step 9.

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.

This file selection and action are available in the PowerGREP5.pgl library as "Search through file names and file contents".

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. Select the file format, archive format, and text encoding configurations that are appropriate for the files you want to search through.
5. Start with a fresh action.
6. Set the action type to “list files”.
7. Enter the search terms that the files should not contain.
8. Turn on the “invert search results” option.
9. 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.

This action is available in the PowerGREP5.pgl library as “Find files not containing a search term”.

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.

Depending on the kinds of files you want to find these email addresses in, you may want to check out these examples first. They elaborate what is summarized as step 1 here.

- Search through Microsoft Word documents
- Search through PDF files
- Search through XPS and OXPS files
- Search through OpenOffice Writer documents
- Search through OpenDocument Format files
- Search through spreadsheets
- Search through mailboxes and email messages

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,}\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.

This action is available in the PowerGREP5.pgl library as "Email: Find email addresses".

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".
4. Turn on "group identical matches".
5. Then turn on the newly revealed option "group results for all files".
6. Leave the search type as "regular expression" and make sure to leave "case sensitive search" off.
7. Enter the regular expression `\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}\b` into the Search box.
8. Select "save results into a single file" in the target file creation list.
9. Click the ellipsis (...) button next to "target file location", and select the file you want to save the results into.
10. If you want a comma-delimited list of addresses for use with email software, set "between collected text" to "text between matches and files". Then type in a comma in the box that appears below the "between collected text" drop-down list.
11. 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.

This action is available in the PowerGREP5.pgl library as "Email: Get a list of all email addresses (one per line)". The comma-delimited variant is available as "Email: Get a list of all email addresses delimited with commas". The actions in the sample library use a more complicated and slightly more accurate regex to find the email addresses. But the basic regex shown above will do the job just fine. The section with regular expression examples in this manual has a detailed discussion about matching email addresses with regular expressions with many more examples.

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.

These actions are available in the PowerGREP5.pgl library as "Find word pairs" and "Find 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 PowerGREP5.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 PowerGREP5.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.

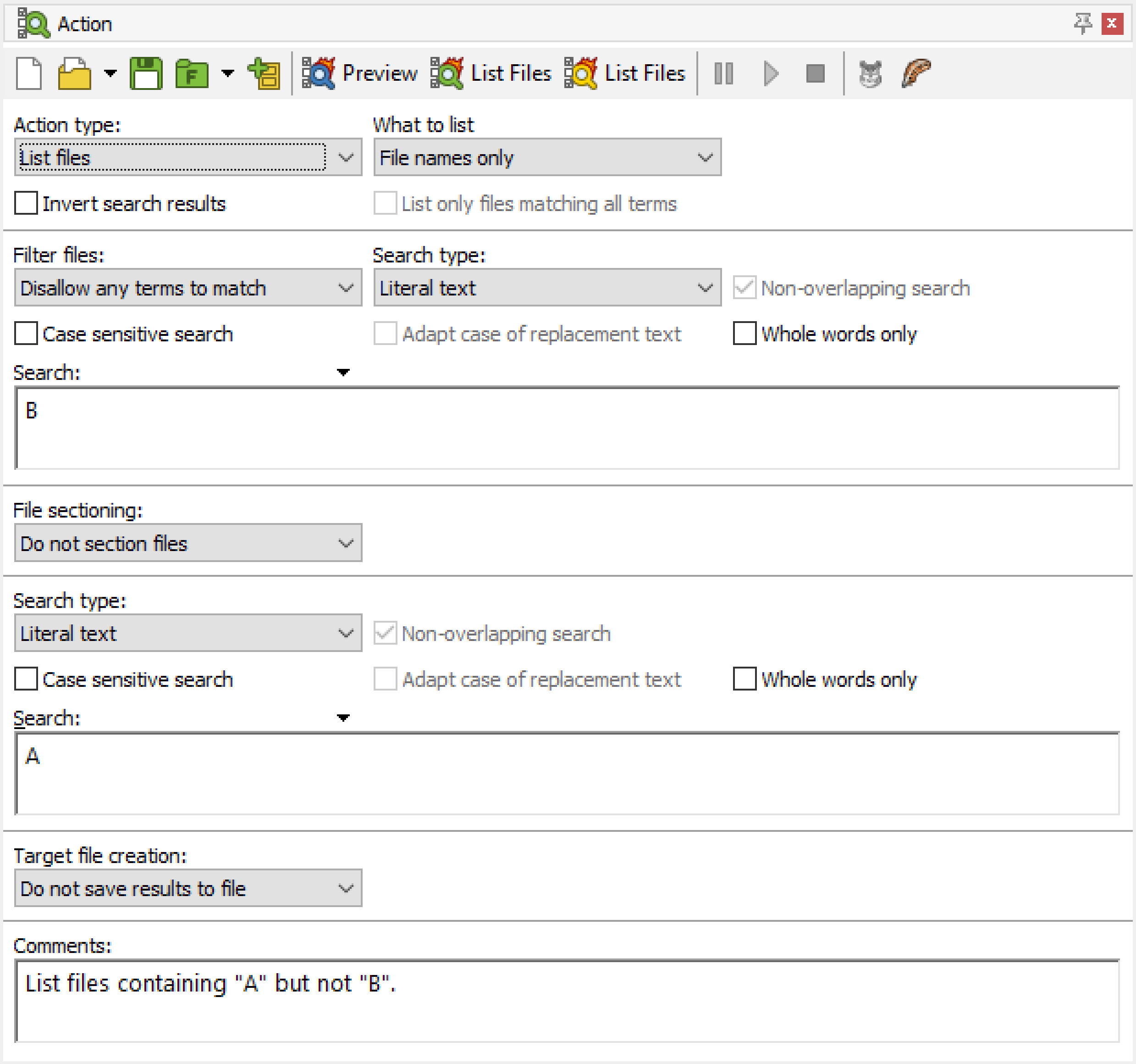
Find Some Terms, Exclude Other Terms

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".

The screen shot below shows what the Action panel looks like when you follow the above steps to find "A and not B".



Action type: List files

What to list: File names only

☐ Invert search results

☐ List only files matching all terms

Filter files: Disallow any terms to match

Search type: Literal text

☑ Non-overlapping search

☐ Case sensitive search

☐ Adapt case of replacement text

☐ Whole words only

Search:

```
B
```

File sectioning: Do not section files

Search type: Literal text

☑ Non-overlapping search

☐ Case sensitive search

☐ Adapt case of replacement text

☐ Whole words only

Search:

```
A
```

Target file creation: Do not save results to file

Comments:

List files containing "A" but not "B".

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 PowerGREP5.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 PowerGREP5.pgl library as "Find two or more words on the same line".

8. Search Through Microsoft Word Documents

Microsoft Word 2003 and prior used the DOC file format to save documents. This is a proprietary binary file format. PowerGREP can convert DOC files to plain text so that you can search through them.

Microsoft Word 2007 and later use the DOCX file format. DOCX files are technically ZIP archives that contain XML and assorted files. While DOCX is an open format in principle, the XML it uses is still really complicated. PowerGREP can convert DOCX files to plain text so you can easily search through them, without having to deal with the XML. PowerGREP can also reconvert its plain text conversion back into the original DOCX file so you can easily search-and-replace through DOCX files.

File Format Configurations for Word Documents

To be able to search through Word documents as if they were plain text documents, you need to set the “file formats to convert to plain text” on the File Selector panel to a configuration that converts Word documents to plain text. In the configuration, the option "Use PowerGREP’s built-in decoder to convert files to plain text“ should be turned on for the file formats ”Microsoft Word 95 to 2003 (DOC)“ and ”Microsoft Word 2007 to 2016“. Default configurations that use these options are ”proprietary formats“, ”all formats“, ”attachments & proprietary formats“, and ”attachments & all formats".

If you want to search only through Word documents, enter the file mask `*.do[ct];*.do[ct][xm]` in the “include files” box on the File Selector panel. If you leave the “include files” and “exclude files” boxes blank, then PowerGREP searches through the plain text conversion of all file formats enabled by the configuration, as well as through the raw contents of all files that are not recognized as one of those file formats.

This file selection is available in the PowerGREP5.pgl library as “Office: Search through Word documents”.

To indicate which Word documents to search through, click on the folder that contains them in the “folders and files” tree. Then select Include File or Folder or Include Folder and Subfolders from the File Selector menu.

Finally, prepare and execute your search on the Action panel.

Search Through The Raw XML Inside DOCX Files

When PowerGREP converts Word documents to plain text, you can only search through the body text of the documents. The conversion does not show any metadata, so you can’t search through that. For DOC files, this is the only way.

If you are familiar with the XML format used by DOCX files, you can tell PowerGREP to search through the raw XML instead. This allows you to search for anything in the files, as long as you know how it is represented in the XML. To do so, select a file format configuration on the File Selector panel that uses the option “search through the individual files inside the compound document” for the “Microsoft Word 2007 to 2016” file format. Default configurations that do so are “compound documents”, “compound documents & proprietary formats”, and “compound documents & writable proprietary formats”. Choose “Compound documents & proprietary formats” if you want to search through the plain text conversion of DOC files in addition to searching through the XML inside DOCX files. The other two skip DOC files.

If you want to search only through DOCX files, enter the file mask `*.do[ct][xm]` in the "include files" box on the File Selector panel.

This file selection is available in the PowerGREP5.pgl library as "Office: Search through the raw XML inside DOCX files".

To effectively work with the XML, you will likely want to use file sectioning. This makes it easy to restrict the main part of the action to the contents of specific XML tags. To search through the body text of DOCX files, for example, take these steps on the Action panel:

1. Start with a fresh action.
2. Set the action type to "search".
3. Enter the search terms that you want to find.
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 `<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.
6. In the Section Collect box, enter the backreference `\1` to restrict the main action to the contents of the <w:t> tag.
7. 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 PowerGREP5.pgl library as "Office: Search printable text in the raw XML inside DOCX files".

9. Search and Replace Through Microsoft Word Documents

Microsoft Word 2003 and prior used the DOC file format to save documents. This is a proprietary binary file format. PowerGREP can convert DOC files to plain text so that you can search through them. This converter does not allow you to make replacements in DOC files. There is no way to do so with PowerGREP.

Microsoft Word 2007 and later use the DOCX file format. DOCX files are technically ZIP archives that contain XML and assorted files. While DOCX is an open format in principle, the XML it uses is still really complicated. PowerGREP can convert DOCX files to plain text so you can easily search through them, without having to deal with the XML. PowerGREP can also reconvert its plain text conversion back into the original DOCX file so you can easily search-and-replace through DOCX files.

File Format Configurations for Editing DOCX Files

To be able to search-and-replace through DOCX files if they were plain text documents, you need to set the "file formats to convert to plain text" on the File Selector panel to a configuration that uses the option "Use PowerGREP's built-in decoder to convert files to plain text" for the file format "Microsoft Word 2007 to 2016". The configuration should not enable any read-only converters. PowerGREP will refuse to run a search-and-replace if the selected configuration enables read-only converters. This means the configuration needs to select "always exclude files of this type" for the "Microsoft Word 95 to 2003" file format. Default configurations that satisfy these requirements are "writable proprietary formats", "all writable formats", "attachments & proprietary formats", "attachments & writable proprietary formats", "attachments & all formats", and "attachments & writable proprietary formats".

With one of these configurations selected on the File Selector panel, you can use the Editor|Open menu item to open a DOCX file and edit it in PowerGREP's built-in editor. Though PowerGREP won't show you the file's formatting, it will be preserved when you save the file.

If you want to search-and-replace only through DOCX files, enter the file mask `*.do[ct][xm]` in the "include files" box on the File Selector panel. If you leave the "include files" and "exclude files" boxes blank, then PowerGREP searches through the plain text conversion of all file formats enabled by the configuration, as well as through the raw contents of all files that are not recognized as one of those file formats.

This file selection is available in the PowerGREP5.pgl library as "Office: Search-and-replace through Word documents (DOCX only)".

To indicate which Word documents to search through, click on the folder that contains them in the "folders and files" tree. Then select Include File or Folder or Include Folder and Subfolders from the File Selector menu.

Finally, prepare and execute your search-and-replace on the Action panel.

Search-and-Replace Through The Raw XML Inside DOCX Files

When PowerGREP converts Word documents to plain text, you can only search-and-replace through the body text of the documents. The conversion does not show any metadata such as hyperlinks, so you can't edit that.

If you are familiar with the XML format used by DOCX files, you can edit the raw XML instead. This allows you to edit anything in the files, as long as you make sure not to mess up the XML structure. To do so, select a file format configuration on the File Selector panel that uses the option "search through the individual files inside the compound document" for the "Microsoft Word 2007 to 2016" file format and that does not enable read-only converters. Default configurations that do so are "compound documents" and "compound documents & writable proprietary formats".

With one of these configurations selected on the File Selector panel, you can expand the nodes for DOCX files in the folders and files tree. You'll then see the file and folder structure inside the DOCX file. Right-click on one of the XML files inside it and click the Edit item to edit it in PowerGREP's built-in editor.

If you want to search only through DOCX files, enter the file mask `*.do[ct][xm]` in the "include files" box on the File Selector panel. When DOCX files are treated as compound documents, the file masks still include and exclude the DOCX files themselves rather than the files they contain. Target and backup settings on the Action panel also make PowerGREP save and back up DOCX files as a whole, even though the search results will show the replacements made in the XML files inside the DOCX files.

This file selection is available in the PowerGREP5.pgl library as "Office: Search through the raw XML inside DOCX files".

To effectively work with the XML, you will likely want to use file sectioning. This makes it easy to restrict the main part of the action to the contents of specific XML tags. The XML in DOCX files does not have line breaks. So you need to change "context type" to avoid heaps of XML on the Results Panel. Select "use sections as context" when using file sectioning. Otherwise, set it to "no context" or to "search for context" with `<?+[^<>]++>?+` as the context regex. This regex matches a single XML tag or the text between two tags. If your replacement string contains reserved XML characters, use extra processing to automatically replace those.

To search-and-replace through the body text of DOCX files, for example, take these steps on the Action panel:

1. Start with a fresh action.
2. Set the action type to "search and replace".
3. Enter the text to search for and replace with.
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 `<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.
6. In the Section Collect box, enter the backreference `\1` to restrict the main action to the contents of the <w:t> tag.
7. Turn on "extra processing".

8. Set the extra processing search type to “delimited literal text”, the “extra term delimiter” to a comma, and the “extra pair delimiter” to an equals sign.
9. Enter `<=&lt;,>=&gt;,&=&amp;` into the “extra processing search” box.
10. Make sure “non-overlapping search” is turned on for extra processing.
11. Set “context type” to “use sections as context”.
12. Click the Preview button to run the action.

This example is available in the PowerGREP5.pgl library as “Office: Search-and-replace in printable text in the raw XML inside DOCX files”.

10. Search Through PDF Files

PDF (Portable Document Format) is a format designed to electronically store printed pages. PowerGREP can extract the text form PDF files and arrange it to reconstruct the text on the page.

To enable this, you need to set the “file formats to convert to plain text” on the File Selector panel to a configuration that uses the option "Use PowerGREP’s built-in decoder to convert files to plain text“ for the file format ”Portable Document Format (PDF)“. Default configurations that use this option are ”proprietary formats“, ”all formats“, ”attachments & proprietary formats“, and ”attachments & all formats".

If you click the (...) button in the File Selector to edit the configuration and select “Portable Document Format (PDF)” in the list, you will see a checkbox labeled "Convert the text in PDF files in reading order rather than trying to mimic the page layout". This option is turned off in all the default configurations. So by default PowerGREP’s plain text conversion of PDF files mimic the layout the text would have when you print the PDF or view it in a PDF viewer. Line breaks are added to limit the length of lines. Whitespace is used to preserve indentation. Text in columns is arranged in columns using extra whitespace.

If you turn on this option, then the plain text conversion will have the text in reading order. No line breaks are added to paragraphs. No whitespace is added to preserve indentation. Text that was in columns in the PDF appears as normal text in the plain text conversion, with all text of the first column before all the text of the second column.

Mimicking the page layout generally makes the text easier to read. But you have to take the extra spaces and line breaks into account when searching. Instead of searching for the phrase “two words” as literal text, you should search for the regular expression `two\s+words`. The `\s+` matches any amount of whitespace or line breaks. So `two\s+words` matches `two words` even when there are multiple spaces or line breaks between the two words.

Converting text in reading order means you don’t have to deal with extra spaces or line breaks in your search terms. But it is mainly helpful when dealing with text in columns. If the phrase “two words” appears in a column in the PDF with “two” at the end of the line and “words” at the start of the next line, then a plain text conversion that mimics the page layout will have the entire line of text of the other column between those two words. Compare this plain text conversion mimicking two columns:

```
Converted from a PDF file with text     The second column has the phrase "two
in two columns.                         words" wrapped across two lines.
```

With this plain text conversion in reading order:

```
Converted from a PDF file with text in two columns. The second column has the
phrase "two words" wrapped across two lines.
```

The regex `two.*?words` with the option “dot matches line breaks” turned on matches “two” followed by “words” with any amount of any text between them. In the second conversion, this would match `two words` as you’d expect. But in the first conversion it matches this:

```
two in two columns. words
```

The regex engine has no concept of columns. It just processes the text from left to right and from top to bottom. Converting PDFs in reading order makes sure the regex engine sees the text in the order you would read it.

To see how your own PDF files fare with these conversions, first select the file format configuration that enables PDF conversion the way you want it on the File Selector panel. Then use the Editor|Open menu item to open a PDF file and view it in PowerGREP's built-in editor. The text shown by the editor is the text that PowerGREP searches through when the PDF is included in an action.

If you want to search only through PDF files, enter the file mask `*.pdf` in the "include files" box on the File Selector panel. If you leave the "include files" and "exclude files" boxes blank, then PowerGREP searches through the plain text conversion of all file formats enabled by the configuration, as well as through the raw contents of all files that are not recognized as one of those file formats.

These file selections are available in the PowerGREP5.pgl library as "Office: Search through PDF documents (mimic page layout)" and "Office: Search through PDF documents (text in reading order)".

The PDF format is designed to store final printouts. It is not designed to be editable. PowerGREP cannot make replacements in PDF files.

11. Search Through XPS and OXPS Files

XPS (XML Paper Specification) and OXPS (Open XML Paper Specification) are two variations of the same file format. Windows Vista and later use this format to spool printed pages to the printer driver. Windows Vista and later also include a printer driver that saves your printout as an XPS or OXPS file to disk.

These files are technically ZIP archives that contain XML and assorted files. The actual content is stored as drawing calls for printers. Text is stored as series of glyph indices rather than as readable text. These glyhps are annotated with the text they're supposed to render. PowerGREP can extract these text annotations from XPS and OXPS files and arrange them correctly to reconstruct the text on the page.

To enable this, you need to set the "file formats to convert to plain text" on the File Selector panel to a configuration that uses the option "Use PowerGREP's built-in decoder to convert files to plain text" for the file format "XML Paper Specification (XPS and OXPS)". Default configurations that use this option are "proprietary formats", "all formats", "attachments & proprietary formats", and "attachments & all formats".

With one of these configurations selected on the File Selector panel, you can use the Editor|Open menu item to open an XPS or OXPS file and view it in PowerGREP's built-in editor. You won't be able to edit the file. You can edit its plain text representation if you first use File|Save As to save it as a plain text file.

If you want to search only through XPS and OXPS files, enter the file mask `*.xps;*.oxps` in the "include files" box on the File Selector panel. If you leave the "include files" and "exclude files" boxes blank, then PowerGREP searches through the plain text conversion of all file formats enabled by the configuration, as well as through the raw contents of all files that are not recognized as one of those file formats.

This file selection is available in the PowerGREP5.pgl library as "Office: Search through XPS and OXPS files".

The XPS format is designed to store final printouts. It is not designed to be editable. XPS files only provide text annotations to make them searchable. Modifying those annotations does not change the text shown on a printout or in an XPS viewer. Because of all this, PowerGREP cannot make replacements in XPS files.

If you want to look at or search through the raw XML inside XPS files, set select the "compound documents" file format configuration. But this is unlikely to give you good results, unless you really know your way around the XML format used by XPS files.

12. Search Through OpenOffice and LibreOffice Writer Documents

OpenOffice Writer and LibreOffice Writer save documents in the OpenDocument Text (ODT) format. ODT files are technically ZIP archives that contain XML and assorted files. While ODF is an open format in principle, the XML it uses is still really complicated. PowerGREP can convert ODT files to plain text so you can easily search through them, without having to deal with the XML. PowerGREP can also reconvert its plain text conversion back into the original ODT file so you can easily search-and-replace through ODT files.

StarOffice Writer saved files with an SXW extension. These use a format that was a precursor to the ODT format. PowerGREP's built-in converter for ODT files also supports SWX files.

File Format Configurations for OpenOffice and LibreOffice Writer Documents

To be able to search through OpenOffice Writer and LibreOffice Writer documents as if they were plain text documents, you need to set the "file formats to convert to plain text" on the File Selector panel to a configuration that uses the option "Use PowerGREP's built-in decoder to convert files to plain text" for the file format "OpenDocument Text (ODT)". Default configurations that use this options are "proprietary formats", "writable proprietary formats", "all formats", "all writable formats", "attachments & proprietary formats", "attachments & writable proprietary formats", "attachments & all formats", and "attachments & all writable formats".

To be able to search-and-replace through ODT files, there's an additional requirement. The configuration should not enable any read-only converters. Default configurations that use the built-in converter for ODT and don't use read-only converters for any other formats are "writable proprietary formats", "all writable formats", "attachments & writable proprietary formats", and "attachments & all writable formats".

With one of these configurations selected on the File Selector panel, you can use the Editor|Open menu item to open a DOCX file and edit it in PowerGREP's built-in editor. Though PowerGREP won't show you the file's formatting, it will be preserved when you save the file.

If you want to search only through OpenOffice Writer, LibreOffice Writer, and StarOffice Writer documents, enter the file mask `*.odt;*.sxw` in the "include files" box on the File Selector panel. If you leave the "include files" and "exclude files" boxes blank, then PowerGREP searches through the plain text conversion of all file formats enabled by the configuration, as well as through the raw contents of all files that are not recognized as one of those file formats.

These file selections are available in the PowerGREP5.pgl library as "Office: Search through OpenOffice/LibreOffice/StarOffice Writer documents (ODT/SXW files)" and "Office: Search-and-replace through OpenOffice/LibreOffice/StarOffice Writer documents (ODT/SXW files)".

To indicate which ODT files to search through, click on the folder that contains them in the "folders and files" tree. Then select Include File or Folder or Include Folder and Subfolders from the File Selector menu.

Finally, prepare and execute your search or search-and-replace on the Action panel.

13. Search Through OpenDocument Format Files

PowerGREP has a built-in decoder for OpenDocument Text (ODT) files. This allows you to search through OpenOffice Writer and LibreOffice Writer documents as if they were plain text files, as described in the preceding example.

PowerGREP does not have a decoder for other OpenDocument formats such as database files (*.odb), chart files (*.odc), formula files (*.odf), graphics files (*.odg), image files (*.odi), presentation files (*.odp), and spreadsheets (*.ods). All these files are technically ZIP archives containing one or more XML files and other support files such as image files.

If OpenOffice or LibreOffice is installed on your PC, then your PC will also have an IFilter installed for all the OpenDocument formats. OpenOffice and LibreOffice provide this IFilter so that Windows Search can extract and index the text from these files. PowerGREP can use the same IFilter to search through these files. Since the IFilter system was designed by Microsoft for Windows Search, and Windows Search can only search, the IFilter system is read-only. So PowerGREP can't make replacements in files that are being converted to plain text using an IFilter.

To have PowerGREP use the OpenOffice or LibreOffice IFilter, you need to set the "file formats to convert to plain text" on the File Selector panel to a configuration that uses the option "Use IFilter, if available for this format, to convert files to plain text" for files that have an extension used by OpenDocument Format. The default configurations "proprietary formats", "all formats", "attachments & proprietary formats", and "attachments & all formats" all include a custom file format named "IFilter" that uses the IFilter option for all ODF formats except ODT. For ODT, these configurations use PowerGREP's built-in converter.

If you want to search only through OpenDocument Format files, enter the file mask `*.od[bcfgimpst];*.sxw` in the "include files" box on the File Selector panel. If you leave the "include files" and "exclude files" boxes blank, then PowerGREP searches through the plain text conversion of all file formats enabled by the configuration, as well as through the raw contents of all files that are not recognized as one of those file formats.

This file selection is available in the PowerGREP5.pgl library as "Office: Search through OpenDocument Format files (requires OpenOffice or LibreOffice)".

Search Through the Raw XML Inside OpenDocument Format Files

Using the built-in converter for ODT files and the IFilter for other ODF files to convert these files to plain text is the most practical way to search through ODF files. But there is another way. You can tell PowerGREP to treat these files as compound documents and search through the raw XML inside them. This works even if you don't have OpenOffice or LibreOffice installed. It even allows you to search-and-replace through the XML.

To handle ODF files as compound documents, select a file format configuration on the File Selector panel that uses the option "search through the individual files inside the compound document" for the "OpenDocument Text (ODT)" and the "zipped compound documents" file formats. The file mask for these two file formats need to match all extensions used by OpenOffice and LibreOffice. If you want to search-and-replace, then the file format configuration shouldn't enable any read-only converters. Default configurations that fit these requirements are "compound documents" and "compound documents & writable proprietary formats".

With one of these configurations selected on the File Selector panel, you can expand the nodes for ODF files in the folders and files tree. You'll then see the file and folder structure inside the ODF file. Right-click on one of the XML files inside it and click the Edit item to edit it in PowerGREP's built-in editor.

If you want to search only through OpenDocument Format files, enter the file mask `*.od[bcfgimpst];*.sxw` in the "include files" box on the File Selector panel.

This file selection is available in the PowerGREP5.pgl library as "Office: Search through the raw XML inside OpenDocument Format files".

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. Start with a fresh action.
2. Set the action type to "search".
3. Enter the search terms that you want to find.
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 `<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.
6. 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 angle brackets) in the sections that are found.

This action is available in the PowerGREP5.pgl library as "Office: Search through printable text the raw XML inside OpenDocument Format files".

14. Search Through Spreadsheets

PowerGREP is designed to search through text documents. It can search through spreadsheets by first converting the spreadsheet into a plain text document. This plain text conversion contains the text or numbers shown by all the cells. Cells are delimited with tabs, rows are delimited with line breaks, and sheets are delimited with page breaks. Content not shown by the cells, like formulas, are not included in the plain text conversion and are thus cannot be searched for with PowerGREP.

To be able to search through spreadsheets as if they were plain text documents, you need to set the “file formats to convert to plain text” on the File Selector panel to a configuration that converts spreadsheets to plain text. In the configuration, the option "Use PowerGREP’s built-in decoder to convert files to plain text“ should be turned on for the file formats ”Microsoft Excel 95 to 2003 (DOC)“, ”Microsoft Excel 2007 to 2016“, ”Quattro Pro“, and ”Lotus 1-2-3 (WKS)“. Default configurations that do this are ”proprietary formats“, ”all formats“, ”attachments & proprietary formats“, and ”attachments & all formats".

With one of these configurations selected on the File Selector panel, you can use the Editor|Open menu item to open a spreadsheet and view it in PowerGREP’s built-in editor. The text you see in the editor is the text that PowerGREP would search through when the file is included in an action. You won’t be able to edit the file. You can edit its plain text representation if you first use File|Save As to save it as a plain text file.

If you want to search only through spreadsheets, add the file mask of the spreadsheet formats you want to search through to the “include files” box on the File Selector panel. For Excel, use `*.xls;*.xls[xm]`. For Quattro Pro, use `*.wq[12];*.wb[123];*.qpw;*.wkq`. For Lotus 1-2-3, use `*.wk[s134]`.

This file selection is available in the PowerGREP5.pgl library as “Office: Search through spreadsheets”.

To indicate which spreadsheets to search through, click on the folder that contains them in the “folders and files” tree. Then select Include File or Folder or Include Folder and Subfolders from the File Selector menu.

Finally, prepare and execute your search on the Action panel.

15. Update Hyperlinks in Microsoft Office Files

To be able to search-and-replace hyperlinks in DOCX, XLSX, and PPTX files saved by Office 2007 and later, you need to set "file formats to convert to plain text" on the File Selector panel to "writable compound documents". This tells PowerGREP to search through the raw XML inside these files. The destination paths and URLs of hyperlinks in Office files appear as plain text inside the XML. This means you can easily replace one path or URL with another. Characters that have special meanings in XML aren't allowed in file paths, so we don't need to take any special care for those.

1. Clear the file selection.
2. Set "file formats to convert to plain text" to "writable compound documents".
3. Enter the file mask `*.do[ct][xm];*.xls[xm];*.pptx` in the "include files" box to restrict the action to Microsoft Office files.
4. 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.
5. Repeat the previous step if you want to search through the files in multiple folders.
6. Start with a fresh action.
7. Set the action type to "search and replace".
8. Set the search type to "literal text".
9. Enter the old path into the Search box and the new path into the Replace box. You could enter \\OLDSERVER\OLDSHARE\ into the Search box and \\NEWSERVER\NEWSHARE\ into the Replace box to change all hyperlinks from the old share on the old server to the new share on the new server.
10. Set "context type" to "search for context". Enter the regular expression `<?+[^<>]++>?+` to get one XML tag of context for hyperlinks in XML attributes, or the text between two XML tags for hyperlinks in body text. Or set "context type" to "no context" if you don't care to see any XML.
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 update the hyperlinks.

An example file selection that does the first 3 steps is available in the PowerGREP5.pgl library as "Office: Search-and-replace through the raw XML inside MS Office files".

If you have multiple sets of links that need to be updated in the same files, you can use the "list of literal text" or "delimited literal text" search type to replace all of them in a single action.

16. Search and Edit Audio File Meta Data

PowerGREP is designed to search through text documents. It can search through audio files or music files by creating a plain text conversion of the meta data inside these files. This includes all the information usually displayed by music players, like the title of the song and the name of the artist.

To be able to search through the meta data in audio files, you need to set the “file formats to convert to plain text” on the File Selector panel to a configuration that converts audio files to plain text. In the configuration, the option "Use PowerGREP’s built-in decoder to convert files to plain text“ should be turned on for the file formats ”audio file meta tags (ID3, APE, FLAC, Vorbis, MP4, WAV)“ and ”Windows Media Audio (WMA) meta tags“. All default configurations do this, except for ”(unused)“, ”none“, and ”compound documents".

This file selection is available in the PowerGREP5.pgl library as “Media: Search and edit audio file meta data”.

With a proper configuration selected on the File Selector panel, you can use the Editor|Open menu item to open an audio file and view it in PowerGREP’s built-in editor. The text you see in the editor is the text that PowerGREP would search through when the file is included in an action. You will see one line of text for each meta tag. It consists of the the name of the tag or a label for the tag indented with spaces, a colon and a space, followed by the contents of the tag. Tags that PowerGREP recognizes are indicated with a label in mixed case. PowerGREP uses the same label across all tag formats. Tags that PowerGREP does not recognized are indicated with the tag’s name in all caps.

```
       Title: Water Prelude (feat. Angel City Chorale)
      Artist: Christopher Tin
       Genre: Classical
       Album: The Drop That Contained the Sea
       Track: 1/10
        Year: 2014
Album Artist: Christopher Tin
    Composer: Christopher Tin
   Tag Types: FLAC
     ENCODER: xACT 2.26
```

PowerGREP uses the following labels for tags that it recognizes: Title, Artist, Genre, Album, Track, Disc, Year, Album Artist, Composer, Publisher, Rating, Tag Types, Comment, and Lyrics. The labels always appear in the same order. Labels may be omitted for files that don’t have a complete set of tags. Labels may be duplicated for files that have duplicate tags with different values. The Title, Artist, and Tag Types labels are always present. For audio files that have no meta data at all their values will be completely blank.

You can freely edit the values of the tags in PowerGREP’s editor. PowerGREP will update the meta data in the audio file when you save it. You can also modify the tag values in a search-and-replace action.

You can add new tags by adding a new line to the file with a label that PowerGREP recognizes followed by a colon, space, and the value you want for the tag. You don’t need to indent the label to make the values line up. Supported tag types are APEv2, FLAC, ID3v1, ID3v2, MP4, Vorbis, WAV, and WMA. The Comment and Lyrics labels allow multiple lines of text. The others only allow a single line.

17. Rename Audio Files Using Meta Data

Once you have the File Selector set up to search through audio files, you can use PowerGREP to rename your audio files using some of the meta data such as the name of the artist and the title of the song.

1. Start with a fresh action.
2. Set the action type to "rename files or folders".
3. Set "what to rename" to "file names only".
4. Set "filter files" to "require one terms to match". Leave the search type as "regular expression".
5. For filtering files, enter the regex `\A *Title: (?'title'\V+)\R *Artist: (?'artist'\V+)` to capture the Title and Artist tags, which PowerGREP always lists first in that order. The named capturing groups carry over to the remainder of the action.
6. In the main part of the action, enter the regex `.*` to match the file's whole name, whatever it is.
7. As the replacement, enter `${artist} - ${title}.%FILEEXT%` to replace the file's name with the artist and title captured from the file's contents along with the file's original extension.
8. Tick the extra processing checkbox. An additional set of controls for entering search terms appears.
9. 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.
10. Leave the extra processing replacement blank so invalid characters are deleted.
11. Set the backup file options as you like them.

Should a file not have a song title or artist name in its meta data, then the regex we're filtering files with will fail to match. That file is thus filtered out from the action and won't be renamed.

This action is available in the PowerGREP5.pgl library as "Media: Rename audio files using meta data".

18. Search and Edit EXIF and IPTC Image Meta Data

PowerGREP is designed to search through text documents. It can search through image files or photos by creating a plain text conversion of the EXIF and/or IPTC meta data inside these files. This includes all the information usually displayed by photo viewers. PowerGREP can extract this meta data from JPEG, TIFF, and PSD (PhotoShop) files.

To be able to search through the meta data in audio files, you need to set the "file formats to convert to plain text" on the File Selector panel to a configuration that converts audio files to plain text. In the configuration, the option "Use PowerGREP's built-in decoder to convert files to plain text" should be turned on for the file format ”image file meta tags (EXIF)". All default configurations do this, except for ”(unused)", ”none", and ”compound documents".

This file selection is available in the PowerGREP5.pgl library as "Media: Search and edit image file meta data".

With a proper configurations selected on the File Selector panel, you can use the Editor|Open menu item to open an image file and view it in PowerGREP's built-in editor. The text you see in the editor is the text that PowerGREP would search through when the file is included in an action. You will see one line of text for each meta tag. It consists of PowerGREP's label for the tag indented with spaces, a colon and a space, followed by the contents of the tag.

```
           Title: Lightroom title
         Subject: Lightroom caption
          Rating: ****
            Tags: Place|Home|Garden;View|Buildings|Home
        Comments: Lightroom user comment
         Authors: Jan Goyvaerts
       Copyright: © 2002 Jan Goyvaerts
      Date Taken: 2002-01-03 12:18:15
      Date Saved: 2014-09-23 7:52:24
ISO Speed Rating: 69
        F/Number: 2.7
   Exposure Time: 1/160 sec
   Exposure Bias: 0 EV
   Metering Mode: Pattern
    Focal Length: 5.41
           Flash: Did not fire
     File Source: Digital camera
     Camera Make: Canon
    Camera Model: Canon DIGITAL IXUS 300
      Lens Model: 5.4-16.2 mm
        Software: Adobe Photoshop Lightroom 5.4 (Windows)
```

PowerGREP uses the following labels for tags that it recognizes: Description, Title, Subject, Rating, Tags, Comments, Authors, Copyright, Date Taken, Date Saved, Date Digitized, Orientation, Image Size, GPS Latitude, GPS Longitude, GPS Altitude, GPS Speed, GPS Tracking, GPS Time, ISO Speed Rating, F/Number, Exposure Time, Exposure Bias, Exposure Program, Exposure Mode, Metering Mode, Exposure Index, Brightness Value, Focal Length, Focal Length (35 mm), Digital Zoom Ratio, Flash, White Balance, White Balance Mode, Gain Control, Contrast, Sharpness, Saturation, File Source, Image Unique ID, Related Sound File, Camera Make, Camera Model, Camera Serial, Camera Owner, Lens Make, Lens Model, Lens Serial and Software. The labels always appear in the same order. Labels may be omitted for files that don't have a complete set of EXIF data.

You can freely edit the values of the Description, Title, Subject, Rating, Tags, Comments, Authors, Copyright, Date Taken, Date Saved, Date Digitized, Orientation, GPS Latitude, GPS Longitude, GPS Altitude, GPS Speed, GPS Tracking, GPS Time, and Camera Owner tags in PowerGREP's editor. You can add missing tags. PowerGREP will update the EXIF and IPTC meta data in the image file when you save it. You can also modify the tag values in a search-and-replace action.

Some tags require specific values. The value for the Rating tag should be between zero and five asterisks. Orientation should be set to Upright, Flip horizontally, Rotate 180°, Flip vertically, Transpose, Rotate 90° clockwise, Transverse, or Rotate 90° counter-clockwise. GPS latitude and longitude need to be in degrees, minutes, and seconds. GPS altitude needs to be a decimal number followed by "m above sea level". Inaccurate GPS readings may show up as "m below sea level".

The Rating, Tags, Comments, Authors, and Copyright tags are always included in PowerGREP's plain text conversion, even if there are no values for them in the file's meta data. This makes it easy to add those values. At least one of the Description, Title, and Subject tags is also always included. Some may be omitted because many applications treat them as redundant, but in different ways. Windows Explorer, for example, writes its Title metadata to both the Description tag and the Title tag. Lightroom writes its Caption metadata to both the Description tag and the Subject tag. PowerGREP does not display redundant tags. It does maintain the redundancy if you edit the tag's value.

19. Search (and Replace) through RTF and HTML as Plain Text

RTF and HTML are two common file formats used for word processing documents and web pages. These are text based formats, so you often can get search results even when searching through their raw contents. If you are familiar with the RTF and HTML formats, you may even prefer to work with their raw contents as that allows you to search for and even manipulate the RTF and HTML tags.

But in many cases, those tags just get in the way. The French word élève, for example, may appear as `\'e9l\'e8ve` in an RTF file or as something much more complicated. In an HTML file it could appear literally as `élève`, but also with character entities like `élève` or `élève` or `élève` or a mixture of those. If you just want to search for some text, it's much easier to do so if all the text is rendered like a word processor or web browser does.

File Format Configurations for RTF and HTML Files

To search through a plain text conversion of RTF and HTML files that eliminates all tags and mimics the page layout, you need to set the "file formats to convert to plain text" on the File Selector panel to a configuration that uses the option "Use PowerGREP's built-in decoder to convert files to plain text" for the file formats "Rich Text Format (RTF)" and "HyperText Markup Language (HTML)". Default configurations that do this are "all formats", "all writable formats", "attachments & all formats", and "attachments & all writable formats".

If you want to search-and-replace through the plain text conversion of RTF and HTML files, then the file format configuration should not use any read-only converters, in addition to using the built-in converters for RTF and HTML. Default configurations that satisfy all this are "all writable formats" and "attachments & all writable formats".

With one of these configurations selected on the File Selector panel, you can use the Editor|Open menu item to open an RTF and HTML file and edit it in PowerGREP's built-in editor. Though PowerGREP won't show you the file's formatting, it will be preserved when you save the file.

If you want to search through only RTF files, enter the file mask `*.rtf` in the "include files" box on the File Selector panel. If you want to search through only HTML files, use the file mask `*.html;*.htm;*.shtml;*.hta`. Enter both marks delimited with a semicolon or line break to search through RTF and HTML files. Leave the "include files" and "exclude files" boxes blank if you want to search through all files. The four configurations mentioned above do not exclude any file formats.

These file selections are available in the PowerGREP5.pgl library as "Office: Search through RTF and HTML as plain text" and "Office: Search-and-replace through RTF and HTML as plain text".

If you'd rather deal with the RTF and HTML code directly, then you need to select a file format configuration that uses the option "search through the file's raw (unconverted) contents" for the RTF and HTML file formats. All default configurations except the four mentioned above and "(unused)" do this. The "(unused)" configuration does not assign any file masks to any file formats, so no files are recognized as being convertible. So "(unused)" too tells PowerGREP to search through the raw contents of RTF and HTML files.

20. Search Through Mailboxes And Email Messages

To search through email with PowerGREP you need to configure how PowerGREP handles email messages and how PowerGREP handles email folders or mailboxes.

Search Through Email Messages

The configuration you select for "file formats to convert to plain text" on the File Selector panel determines how PowerGREP handles email messages. This affects both email messages stored in separate files as well as email messages stored in mailboxes. Within the configuration, which you can view or edit by clicking the (...) button, the relevant settings are those for the "Email message (MIME or UUencode)" and the "Microsoft Outlook message" file formats. The MIME format is the most common format for storing email messages. It is used for emails saved separately in `.eml` files and for emails inside MBOX mailboxes and Outlook Express `.dbx` folders. The Outlook format is used for emails saved separately in `.msg` files and for emails in Outlook `.pst` folders.

If you select "Use PowerGREP's built-in decoder to convert files to plain text" for these formats, then PowerGREP converts email messages to plain text. The plain text conversion consists of the basic email headers Subject, From, Date, and To followed by the body text of the email. PowerGREP converts the body text to plain text even for emails sent purely as HTML or RTF. Email messages appear as files in the folders and files tree on the File Selector panel. Attachments are not accessible and are not searched through. This option is recommended if you know that the text you're searching for is in the body text of an email rather than in an attachment. Skipping attachments speeds up the search considerably. Default configurations that use this option are "proprietary formats" and "all formats".

If you select "Search through the individual files inside the compound document" for these formats, then email messages appear as expandable nodes in the folders and files tree. The files inside that node depend on the format of the message and its contents. Default configurations that use this option are "attachments & proprietary formats" and "attachments & all formats".

Inside single-part MIME messages you'll see a file body.txt or body.html with the body text of the email and a file headers.txt with the basic email headers. Inside multi-part MIME messages, you'll see numbered files with extensions corresponding to their content type. There will always be a file 0.txt with the basic headers of the email. The other files hold the various parts of the message, numbered in the order they have in the message. An email sent with the body in both text and html formats along with one attached image, for example, is shown as containing the files 1.txt, 2.html, and 3.png. If the attachment headers indicate file names, those file names are used instead of the numbers.

For UUencode messages, you'll see a file body.txt with the body text and a file headers.txt with the basic email headers. Attachments appear as additional files with their file names.

For Outlook messages, you'll usually see two files body.txt and body.rtf with the body text of the email in plain text and rich text formats as saved by Outlook. Attachments appear as additional files with their file names. There will also be a file headers.txt with the basic email headers.

To decide whether "attachments & proprietary formats" or "attachments & all formats" is the better choice, check out the example about searching through RTF and HTML as plain text. You'll likely prefer "attachments & all formats" when dealing with HTML email so the HTML tags don't get in the way. Since

RTF is only used by Outlook emails and Outlook normally saves its own plain text conversion, you may even want to edit this configuration and select “always exclude files of this type” for “Rich Text Format”. Then the body.rtf inside Outlook messages is always skipped, but body.txt is still searched through.

Search Through Mailboxes

The configuration you select for “archive formats to search inside” on the File Selector panel determines how PowerGREP handles mailboxes. Within the configuration, which you can view or edit by clicking the (...) button, the relevant settings are those for the "MBOX mailboxes“, ”Outlook folders (PST and OST)“, and ”Outlook Express folders (DBX)" archive formats. The MBOX format is used by most email software on UNIX/Linux. It is also used by many Windows email clients. Microsoft’s Outlook and Outlook Express have their own mailbox formats. MBOX and DBX files store messages in MIME format. PST files store messages in Outlook’s own format. OST files are used by Outlook 2013 and later.

Since the MBOX format has its roots in the UNIX world, MBOX files are often saved without an extension. PowerGREP’s default archive configurations treat files named “INBOX” or “Sent” without an extension as MBOX files. You may need to edit the file masks for the MBOX format to make sure PowerGREP correctly recognizes your mailboxes.

If you clear the checkbox “search through files inside archives of this format” then all files matching that format’s file masks are excluded from the action. If you tick the checkbox, those files appear as folders in the folders and files tree in the File Selector.

For MBOX and DBX files, those folders contain numbered files starting with `1.eml` where each file is one email message, numbered in the order the messages have in the mailbox. These `.eml` files hold the plain text conversion of each email or act as compound documents with the email body and attachments inside them depending on the file format configuration as described in the previous section.

For PST files, those folder nodes contain Outlook folders. Inside the Outlook folders you’ll see numbered files starting with `1.txt` when the file format configuration is set to convert Outlook messages to plain text. But you’ll see numbered folders starting with `1` when the file format configuration is set to treat Outlook messages as compound documents. The reason messages are shown as folders is that PST files are an email database rather than a collection of separate MSG files.

Default configurations that search inside all mailbox formats are “mailboxes only”, “mailboxes and zip archives”, and “mailboxes and all archives”. If the file format configuration is set to search through email attachments, then the archive configuration determines whether PowerGREP searches inside attachments that are zip archives or other archives.

By default Outlook saves its PST files under the AppData or Application Data folder under your Windows user profile. This folder is normally a hidden folder. Hidden folders are hidden from PowerGREP’s view by default. If you want to search through PST files inside the AppData folder or inside another folder, then you need to set “hide files and folders” on the File Selector panel to a configuration that does not hide hidden files and folders. Of the predefined configurations, you can select any configuration that does not have “hidden” in its name.

If you want to restrict the search to certain mailboxes based on their file names, you need to use the “include folders” and “exclude folders” boxes on the File Selector panel. PowerGREP treats mailbox files as folders when searching inside them.

File selections for searching through email are available in the PowerGREP5.pgl library as “Email: Search through email body text” and “Email: Search through email body text and attachments”.

21. Search Through ZIP Files And Other Archives

You can easily search through files inside ZIP files and other archives with PowerGREP. All you need to do is to set “archive formats to search inside” on the File Selector panel to a configuration that enables the archive formats you want to search inside. PowerGREP comes with a number of predefined configurations that enable none, some, or all of the archive formats supported by PowerGREP. You can edit these configurations or create your own to tailor them to your needs. It’s best to enable only those archive formats that you really want to search inside, so PowerGREP doesn’t waste its time decompressing other archives.

Select “ZIP archives only” to search through files inside ZIP archives and skip all other archives formats. Select “all archives” to search inside all archive formats supported by PowerGREP, but not through mailboxes or disk images.

Selecting an archive format configuration does not affect files that are not inside archives. Those files are still searched through if you mark the folder that contains them.

When an archive format is enabled, PowerGREP treats files that match that archive format’s file mask as (compressed) folders. If you want to search through all files in an archive, including files in subfolders inside the archive, make sure to use the command Include Folder and Subfolders in the File Selector menu to mark that archive or one of its parent folders.

If you want to search only through files inside ZIP archives, set “include folders” on the File Selector panel add the file mask `*.zip`. You cannot use “include files” for this. That is for the files inside the archives. To search only through TXT files inside ZIP archives, set “include files” to `*.txt` and “include folders” to `*.zip`.

This file selection is available in the PowerGREP5.pgl library as “Archives: Search only through files inside ZIP archives”.

22. Search Through UOT Files

The UOT (United Office Text) is part of the UOF (United Office Format) group of file formats used by Chinese office suites. OpenOffice Writer can also save UOT files.

PowerGREP does not have built-in support for UOT. It does include an external converter as an example. You can find it as uot2txt.exe in the Examples subfolder of PowerGREP's installation folder. That is C:\Program Files\Just Great Software\PowerGREP 5\Examples by default. C# source code is also included. This executable requires version 4.5 of the .NET framework to run. Windows 8, 8.1, and 10 include .NET 4.5. For Windows Vista SP2 and Windows 7 SP1, you can download .NET 4.5 (48 MB) if you don't have it installed already. Windows XP is not supported.

To use the converter, click the (...) button next to "file formats to convert to plain text" on the File Selector panel. Select the configuration you want to edit. Or, create a new configuration by selecting the configuration you want to clone and then clicking the New button in the left hand side of the dialog box. Which configuration is best as a starting point depends on whether you want to search through any other files at the same time as searching through UOT files. For this example it doesn't matter.

Click the New button in the right hand corner of the dialog box to add a custom file format to the configuration. Name the format "United Office Text (UOT)" and enter `*.uot` as its file mask. Untick all checkboxes except "use an external application to convert files to plain text". Enter this command line:

```
"%APPPATH%\Examples\uot2txt.exe" "%INFILE%" "%OUTFILE%"
```

Set the encoding to "Unicode, UTF-16 little endian".

All of this tells PowerGREP to convert files with an `.uot` extension using uot2txt.exe which can be found in the Examples folder of the folder where you installed PowerGREP. PowerGREP should pass the full paths to both the input file (the .uot file) and the output file (a plain text) file to the external converter. PowerGREP should use the UTF16-LE encoding to read the plain text conversion.

Click OK to save and select the new configuration. You can now use the Editor|Open menu item to open an UOT file and view it in PowerGREP's built-in editor. You won't be able to edit the file. You can edit its plain text representation if you first use File|Save As to save it as a plain text file.

If you want to search only through UOT files, enter the file mask `*.uot` in the "include files" box on the File Selector panel. If you leave the "include files" and "exclude files" boxes blank, then PowerGREP searches through the plain text conversion of all file formats enabled by the configuration, as well as through the raw contents of all files that are not recognized as one of those file formats.

This file selection is available in the PowerGREP5.pgl library as "Office: Search through UOT files".

23. 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 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.

24. 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.

25. 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. Select a file format configuration such as "none" or "writable proprietary formats" that does not enable and read-only converters.
3. Start with a fresh action.
4. Set the action type to "search and replace". Set the search type to "list of regular expressions".
5. In the Search box, enter the regular expression `\A`. This regular expression matches the position at the very start of the file.
6. In the Replacement box, enter the header text you want to insert.
7. Click the green plus button to the left of the Search box to add a second step to the action.
8. Enter `\z` in the Search box to make the second step match at the very end of the file.
9. Enter the footer text in the Replacement box.
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 Replace button to actually add the header and footer.

26. 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 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 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. Select a file format configuration such as "none" or "writable proprietary formats" that does not enable and read-only converters.
3. Start with a fresh action.
4. Set the action type to "search and replace". Set the search type to "regular expression".
5. Enter `^` as the search term. This regular expression matches the start of any line in the file, including blank lines.
6. 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.
7. Set the target and backup file options as you like them.
8. Click the Preview button to run a test.
9. 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 lookaround 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 delmiters” 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.

27. 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, also known as the form feed control character. 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 decoders that convert PDF files and XPS files into plain text (so PowerGREP can search through them) also insert page breaks that match the page transitions in the original PDF and XPS files. 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 processes the PDF or XPS 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. Select a file format configuration such as "proprietary formats" that converts PDF and XPS files to plain text.
3. Start with a fresh action.
4. Set the action type to "collect data".
5. Set "file sectioning" to "split along delimiters".
6. 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.
   - 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.
   - Set the "search type" to "regular expression" and type in the regex `\f` into the "section search" box. This regular expression matches the "form feed" control character that indicates page breaks.
7. Specify your search term(s) in the main part of the action.
8. In the collect box, use the match placeholder `%SECTIONNN%` as a placeholder for the page number. E.g. `%MATCH% on page %SECTIONNN%` collects `found me on page 7` when the main part of the action finds "found me" in the 7th section (page).
9. Click the Search button to run the search.

You can find this action in the PowerGREP5.pgl standard library as "Collect page numbers".

28. Update Copyright Years

At the start of every year, you have to update the year in the copyrightstatements 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 &copy; 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. Select a file format configuration such as "none" or "writable proprietary formats" that does not enable and read-only converters.
3. Open the PowerGREP5.pgl library file included with PowerGREP. You can find it in the folder where PowerGREP is installed, c:\Program Files\Just Great Software\PowerGREP 5 by default.
4. Select "Update copyright statements" in the library, and click the Use Action button.
5. Set the replacement to `\1-` (backslash, one, dash) followed by the current year.
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 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 +(&copy;|\(c\)|&copy;) +\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 +(&copy;|\(c\)|&copy;) +\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 &copy;. The symbol must be followed by one or more spaces, and 4 digits.

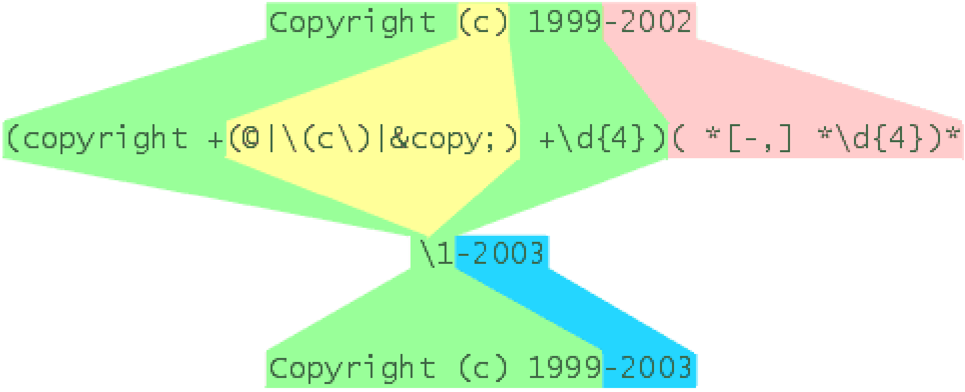
The first part of the regular expression will successfully match a copyright statement in the form of "Copyright (c) 1999".

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 `\1-2016` 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) 1999-2016". 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.



29. 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. Select a file format configuration such as "none" or "writable proprietary formats" that does not enable and read-only converters.
3. Start with a fresh action.
4. Set the action type to "search and replace" or "collect data". Leave the search type as "regular expression".
5. 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:

6. 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.
7. Set the target and backup options as you want.
8. 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:

6. As the replacement text or text to be collected, without regard to padding.
7. Turn on the "extra processing" option.
8. Leave the extra processing search type as "regular expression", and turn on the option "dot matches newlines".
9. In the "extra processing search" box, enter the regular expression `.++`. This regular expression matches the entire replacement text from step 5.
10. 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.
11. Set the target and backup options as you want.
12. Click the Preview button to check the results.

30. 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. Select a file format configuration such as "none" or "writable proprietary formats" that does not enable and read-only converters.
3. Set the action type to "search and replace". Leave the search type as "regular expression".
4. Enter the regular expression `\w++`. This regular expression matches a single word.
5. 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.
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 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 six steps are the same as in the previous action.

7. Select "search for sections" in the "file sectioning" drop-down list.
8. 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.
9. Click the Preview button to run a test.
10. 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.

31. Convert Text File Encoding And Line Break Style

The reference topic about text encoding configurations provides some background information on the various encodings used for plain text files, and how to set up PowerGREP to make sure it correctly displays and searches through the text in your files. This example assumes you've already selected a text encoding configuration that allows all your files to be displayed correctly when opening them in PowerGREP's built-in editor.

While PowerGREP supports all the text encodings and legacy code pages that still have any relevance and offers a lot of flexibility in using different encodings when reading different files via its text encoding configurations, many other applications are not nearly as flexible. Windows applications, for example, typically only support the PC's default code page (such as Windows 1252) and a few forms of Unicode (UTF-8 and UTF-16LE with a byte order marker).

Line break styles can also be an issue. Windows text files normally terminate lines with a CRLF pair, while UNIX/Linux and OS X text files use a single LF. Classic Mac text files used a single CR. PowerGREP automatically handles all these as well as all other Unicode line breaks. So you don't need to tell PowerGREP which line break style your files use. But again, other applications are usually not so flexible. Windows applications like Notepad show UNIX text files as if they had no line breaks. Linux applications often show the CR that they're not expecting at the end of a line as a Ctrl+M control character.

With PowerGREP you can easily convert your text files to a specific encoding and a specific line break style.

1. Select the files you want to convert in the File Selector.
2. Make sure "text encodings to read files with" is set to a configuration that allows PowerGREP to correctly read the text in all those files.
3. Start with a fresh action.
4. Set "action type" to "list files".
5. Leave the Search box blank to convert all files you selected in step 1.
6. Set "target file creation" to "convert matched files to text" if you want to overwrite each file with its conversion. Set it to "convert copies of matched files to text" if you want to keep the original files and save the conversions in another folder. If you choose the latter, you'll get two additional settings to specify which folder.
7. Set "target file text encoding" to the encoding that the converted files should use. Converting to Unicode always works as Unicode supports all characters. Converting to other encodings may cause characters to be lost if the new encoding does not support all characters used in some files. Lost characters appear as question marks after the conversion. Choose ”same as original file" if you want to change the line break style without changing the encoding.
8. Set "target file line break style" to the line break style that the converted files should use. Unicode line breaks are unaffected. Only CR, LF, and CRLF line breaks are converted to the style you choose. Choose ”same as original file" if you want to change the encoding without changing the line break style.
9. Set the backup file options as you like them, so you can undo the conversion if it doesn't work out the way you expected.
10. Click the Convert Files to execute the conversion.

32. Convert Files in Proprietary Formats to Plain Text

PowerGREP can convert files in many proprietary file formats to plain text in order to search through them. Many examples earlier in this section explain how to set this up. This example assumes you've already done so.

If you need to process these files with other applications that can't read their proprietary formats, use PowerGREP to convert the files to plain text, and then use the other applications on the converted files.

1. Select the files you want to convert in the File Selector.
2. Make sure "file formats to convert to plain text" is set to a configuration that allows PowerGREP to correctly read the text in all those files.
3. Start with a fresh action.
4. Set "action type" to "list files".
5. Leave the Search box blank to convert all files you selected in step 1.
6. Set "target file creation" to "convert copies of matched files to text".
7. Set "target file destination type" to "single folder" or "folder tree".
8. Click the (...) button next to "target file location" to choose the target folder. Or enter the path to a new folder which PowerGREP will create when you execute the action.
9. Set "target file text encoding" to an encoding supported by the application you want to process these files with. Choose Unicode if supported by the target application as Unicode supports all characters. In this situation, "same as original file" means the encoding used by PowerGREP's plain text conversion (which is UTF-16LE for most formats) rather than the encoding used by the original file's proprietary format.
10. Set "target file line break style" to the line break style that the converted files should use. Choose CRLF if you'll process the files on Windows. Choose LF for UNIX, Linux, or OS X.
11. Set the backup file options as you like them.
12. Click the Convert Files to execute the conversion.

Doing this conversion is also useful if you will be repeatedly searching through the same set of files and the files are too large to fit in PowerGREP's conversion cache. It's even more useful if many people on your network are searching through the same set of files, as the conversion cache is not shared between PCs. This way, the conversion needs to be done only once. Converting files from their proprietary formats to plain text usually takes much longer than searching through that plain text.

When PowerGREP is the target application, set "target file text encoding" to UTF-16LE. Also make sure that "text encodings to read files with" has the option "write a byte order marker at the start of Unicode files" turned on for the default settings and all format specific settings. The predefined "generic auto detection" configuration does this. This ensures that all the converted files use UTF-16LE with a BOM. These files support all characters and PowerGREP will always read them correctly due to the BOM, regardless of the text encoding configuration used when searching through these files in the future. Set "target file line break style" to "same as original file" as PowerGREP handles all line break styles automatically.

When there are new or modified files to be converted, you can repeat the action to convert just the modified files. To do this, set "file modification dates" in the File Selector to "modified on or after". Set the date to the day you last ran the conversion.

33. Find Bytes That Are Not Part of Valid UTF-8 Sequences

With its default settings, PowerGREP does a very good job of automatically handling all Unicode text files. When you have files in a variety of legacy encodings that cannot be auto-detected, you can use a text encoding configuration to make sure PowerGREP always shows you the correct text. PowerGREP is also very flexible at handling files that contain bytes that aren't strictly valid for their encoding. Even when searching and replacing through such files, PowerGREP preserves any invalid bytes in the files.

But many other applications aren't as flexible. Many scripting languages, for example, simply let your scripts crash when they read a file as UTF-8 and the file contains even one byte that is not part of a valid UTF-8 sequence. This example shows how you can disable PowerGREP's smart handling of text file encodings and instead look at the raw bytes in UTF-8 files. Then you can search for bytes that aren't valid UTF-8 sequences.

1. Open the PowerGREP.pgl library file included with PowerGREP. You can find it in the folder where PowerGREP is installed, c:\Program Files\JGsoft\PowerGREP3 by default.
2. Select the action "Encodings: Find bytes that are not part of valid UTF-8 sequences" in the library, and click the Use button. This loads a somewhat complicated regular expression onto the Action panel. It matches any byte that is not part of a valid UTF-8 sequence.
3. There is a file selection labeled "Encodings: All files as binary" in the library. Loading this does steps 5 through 7 below, but clears any other settings. So load the file selection from the library only if you haven't already marked the files or folders you want to search through.
4. Select the UTF-8 files you want to inspect in the File Selector. This action only produces meaningful results on files that are mostly UTF-8, but have invalid bytes here and there. If you use it on files that aren't UTF-8, the action will find pretty much all bytes 0x80 through 0xFF.
5. Set "file formats to convert to plain text" to "None". PowerGREP's converters for proprietary formats produce UTF-16 for most formats. They never produce invalid UTF-8. So there's no point in including files in proprietary formats in this action.
6. Set "text encodings to read files with" to "all files as binary". This predefined configuration tells PowerGREP to treat all files as binary files.
7. Turn on "search through binary files" to make sure PowerGREP actually searches through any files.
8. Click the Preview button to run the action.

After running this example, the Results panel will show you all the bytes that PowerGREP found that aren't part of valid UTF-8 sequences. This allows you to manually fix those files, or determine the cause of these files not being valid UTF-8.

If you just want to get a list of files that aren't valid UTF-8 without seeing the individual bytes, load the action "Encodings: Find files with bytes that are not part of valid UTF-8 sequences" from the library instead. This action uses the "list files" rather than the "search" action type. This is faster as PowerGREP will continue with the next file as soon as one invalid byte is found.

If you want to remove the offending bytes, load the action "Encodings: Delete bytes that are not part of valid UTF-8 sequences" from the library. This uses the "search and delete" action type to delete all bytes matched by the regular expression.

34. Replace in File Names and Contents

You can search and replace through file names with the “rename files or folders” 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 or folders”.
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. 2016).
6. In the Replace box, enter the new year (e.g. 2017).
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 2016 in their paths into 2017. When that’s done, the second step searches through the contents of the copied files, replacing 2016 with 2017.

This sequence is available in the PowerGREP5.pgl library as “Replace in file names and contents”.

35. 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 replacde 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. Make sure the file format configuration searches through the raw (unconverted) contents of HTML files. The predefined "None" configuration is one that does this.
3. Start with a fresh action.
4. Set the action type to "search and replace".
5. Set "filter files" to "require all search terms to match". Leave the search type as "regular expression".
6. 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".
7. 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.
8. 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.
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 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 PowerGREP5.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.

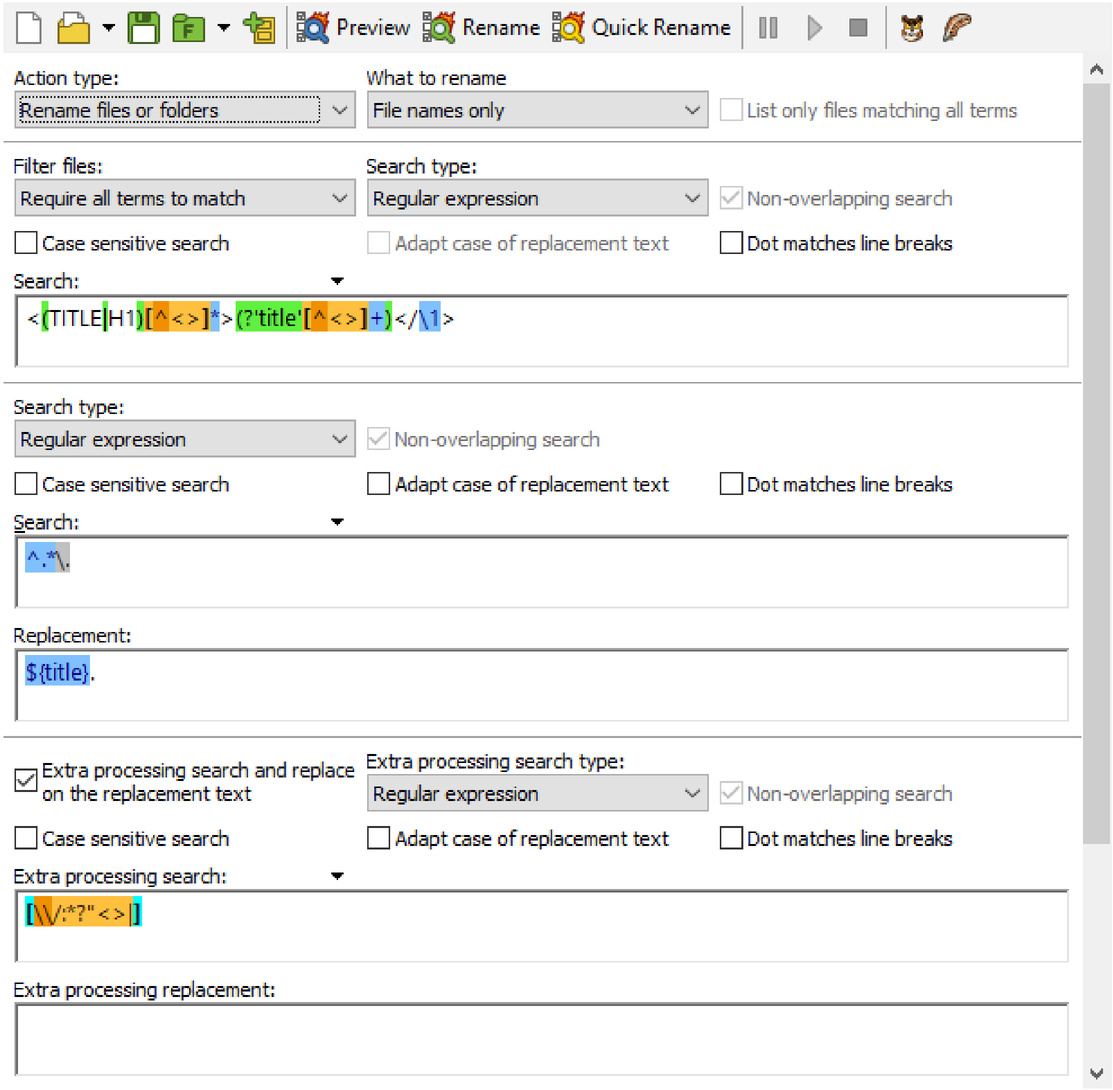
The replacement text becomes `<head>\1<title>${h1}</title>`. In addition to inserting the new title tag and the named capturing group, this replacement text also re-inserts the <head> tag that we matched and the text between the head and title tags that we may have matched. If there was no title tag 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 PowerGREP5.pgl standard library as "Update or insert HTML title tags".

36. Rename Files Based on HTML Title Tags

The “rename files or folders” 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.



Action type: Rename files or folders

What to rename: File names only

□ List only files matching all terms

Filter files: Require all terms to match

Search type: Regular expression

☑ Non-overlapping search

□ Case sensitive search

□ Adapt case of replacement text

□ Dot matches line breaks

Search:

```
<(TITLE|H1)[^<>]*>(?'title'[^<>]+)</\1>
```

Search type: Regular expression

☑ Non-overlapping search

□ Case sensitive search

□ Adapt case of replacement text

□ Dot matches line breaks

Search:

```
^.*\.
```

Replacement:

```
${title}.
```

☑ Extra processing search and replace on the replacement text

Extra processing search type: Regular expression

☑ Non-overlapping search

□ Case sensitive search

□ Adapt case of replacement text

□ Dot matches line breaks

Extra processing search:

```
[\\/:*?"<>|]
```

Extra processing replacement:

1. Select the HTML files you want to rename in the File Selector.
2. Make sure the file format configuration searches through the raw (unconverted) contents of HTML files. The predefined “None” configuration is one that does this.
3. Start with a fresh action.
4. Set the action type to “rename files or folders”.

5. Leave “what to rename” set to “file name only”. Our search-and-replace should only change the file’s name.
6. Set “filter files” to “require all search terms to match”. Leave the search type as “regular expression”.
7. 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”.
8. 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.
9. 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.
10. Tick the extra processing checkbox. An additional set of controls for entering search terms appears.
11. 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.
12. Leave the extra processing replacement blank so invalid characters are deleted.
13. Set the backup file options as you like them.
14. Click the Preview button to run a test.
15. 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 PowerGREP5.pgl library as “Rename files based on HTML title tags”.

37. 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. Make sure the file format configuration searches through the raw (unconverted) contents of HTML files. The predefined "None" configuration is one that does this.
3. Start with a fresh action.
4. Set the action type to "search and replace". Leave the search type as "regular expression".
5. 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.
6. Type the tag you want to replace all body tags with in the Replacement box. E.g.: `<BODY BGCOLOR=white TEXT=black>`
7. Set the target and backup file options as you like them.
8. Click the Preview button to run a test.
9. 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 PowerGREP5.pgl standard library as "Replace HTML tags".

38. 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. Make sure the file format configuration searches through the raw (unconverted) contents of HTML files. The predefined "None" configuration is one that does this.
3. Start with a fresh action.
4. Set the action type to "search and replace". Leave the search type as "regular expression".
5. Select "search for sections" from the File Sectioning drop-down list. Leave the search type as "regular expression".
6. In the Section Search box, enter the regular expression `<table[^>]*>` and make sure to leave "case sensitive search" off.
7. 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.
8. Enter `bgcolor=blue` in the Replacement box. Each bgcolor attribute will be replaced with whatever you enter 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 replace "bgcolor" attributes in "table" tags.

You can find this action in the PowerGREP5.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.

39. 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. Make sure the file format configuration searches through the raw (unconverted) contents of HTML files. The predefined "None" configuration is one that does this.
3. Start with a fresh action.
4. Set the action type to "search and replace". Leave the search type as "regular expression".
5. Select "split along delimiters" from the File Sectioning drop-down list.
6. Set the "section search type" to "list of regular expressions".
7. Add `<a\b[^<>]*>.*?</a>` as the first file sectioning regular expression. It matches any <a> tag and its contents.
8. 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 `&lt;`.
9. Make sure "non-overlapping search" is turned on. The file sectioning should make one pass of the file using both regular expressions.
10. 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.
11. Enter `<a href="\0">\0</a>` in the Replacement box. This replaces each URL with itself wrapped inside an anchor using itself as the destination.
12. Set the target and backup file options as you like them.
13. Click the Preview button to run a test.
14. 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 PowerGREP5.pgl standard library as "Put anchors around URLs that are not already inside a tag".

40. 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:

```
&=&amp;
<=&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 PowerGREP5.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.

41. 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 &lt; and &amp;. 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 `&lt;`.
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 `&amp;`.
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.<=&lt;
12.>=&gt;
   &=&amp;
```

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 PowerGREP5.pgl standard library as “Replace reserved characters in XML files”.

42. 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 newlines" 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.

43. 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 PowerGREP5.pgl standard library as "Convert Windows paths into UNIX paths".

44. 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 PowerGREP5.pgl standard library as "CSV: Extract data into a CSV file or spreadsheet".

45. 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 PowerGREP5.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 PowerGREP5.pgl standard library as "CSV: Unpad fields".

46. 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 PowerGREP5.pgl standard library as “Collect a numbered list”.

47. 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 PowerGREP5.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.

48. 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 PowerGREP5.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.

49. Process Files in a Batch File or Script

PowerGREP does not have the ability to execute a command or external program on each file that it finds. But with PowerGREP you can easily collect files into a batch file or script that executes those commands.

For this example, assume you purchased some music that you downloaded in FLAC format, for best quality. Now you want to put this music onto a pocket player in MP3 format. You want to use FFmpeg to perform the conversion. This example shows how you can easily create a batch file with PowerGREP that runs FFmpeg on each of your FLAC files to convert it into an MP3 file.

You can find this action in the PowerGREP5.pgl library as "Batch file to convert audio files to MP3 using FFmpeg".

1. Mark the folder(s) containing the FLAC files in the File Selector.
2. Enter `*.flac` into the "include files" box.
3. Select a file format configuration such as "proprietary formats" that searches through audio files. Though we don't use this ability in this example, PowerGREP recognizes the FLAC file format and can search through metadata in FLAC files. The "audio file meta tags" file format needs to be enabled. Otherwise PowerGREP would skip all FLAC files.
4. Set "archive formats to search inside" to "None". FFmpeg can't use paths that point to files inside archives.
5. Start with a fresh action.
6. Set the action type to "file or folder name collect".
7. Leave the Search box blank to convert all FLAC files. If you enter a search term, only files containing that search term in their file name will be collected into the batch file.
8. In the Collect box, enter a command line to FFmpeg. `ffmpeg -i "%FILE%" "E:\%FILENAMENOEXT%.mp3"` puts all MP3 files into the root of the E:\ drive. If your FLAC files are in a folder structure you can preserve that with path placeholders. If each album has its own folder, use `"E:\%FOLDER<1\FILENAMENOEXT%.mp3"`. If each artist has a folder and each album a folder inside that, use `"E:\%PATH<2\FILENAMENOEXT%.mp3"`.
9. Set "target file creation" to "save results into a single file" and set "target file location" to the full path to a .bat file for PowerGREP to create.
10. Set "target file text encoding" to your computer's default code page, which is probably Windows 1252.
11. Set "target line break style" to "Windows (CR LF)".
12. Leave "order of matches from different files" set to "no particular order". The order in which the files are converted is irrelevant.
13. Set the backup options for the .bat file as you like them.
14. Click the Collect button to create the batch file.
15. Double-click one of the highlighted matches on the Results panel to open the batch file in PowerGREP's built-in editor to check that everything looks correct.
16. Open a command prompt from the Windows Start menu and execute the batch file. Or just double-click the batch file in Windows Explorer.

In many situations, you may need to collect some code before and after the block of statements that process all the files. Your script may need a shebang or import statements. Or it may need to do some extra work before and after all the files are processed. You can do this with the "collect headers and footers" checkbox. After turning this on, select "target file header" and enter the block of code that should go before the file statements. Then select "target file footer" to enter the code that should go after the file statements.

50. 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.

51. 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:

`bdsl.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++`
3. Date, time and time zone stamp: `\[[^]]++\]`
4. HTTP request, consisting of request method (GET), file (/) and protocol (HTTP/1.1): `"(?:GET|POST|HEAD) [^\s"]++ 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 ` `. The result is:

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

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

This action is available in the PowerGREP5.pgl library as “Logs: Inspect Apache web logs”.

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 [^\s?"]+?\.html?+[^\s"]*+ HTTP/[0-9.]++" [0-9]++ [-0-9] "([^"]*+)"` 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 fore were also removed. We're capturing the HTTP request with `GET [^\s?"]+?\.html?+[^\s"]*+ 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 PowerGREP5.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 [^\s?"]+?\.html?+[^\s"]*+ HTTP/[0-9.]++" [0-9]++ [-0-9]++ "(?:-|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 PowerGREP5.pgl library as "Logs: Inspect Apache web logs - Referring domains".

52. 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\?.*?\bq=\+*+([^&"\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 `\bq=\+*+([^&"\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 PowerGREP5.pgl library file included with PowerGREP. You can find it in the folder where PowerGREP is installed, c:\Program Files\Just Great Software\PowerGREP 5 by default.
3. Select "Logs: 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 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 "Logs: 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 `\l2` (backslash ell two) to collect the search terms converted to lowercase, and `\1` to collect the landing page.

53. 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 "split 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'[0-9]{1,2}+)/(?'month'Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)/(?'year'[0-9]{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 PowerGREP5.pgl library as "Logs: 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”.

54. 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'[0-9]{1,2}+)/(?'month'Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)/(?'year'[0-9]{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 PowerGREP5.pgl library as "Logs: Merge Apache web logs".

55. 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 "split 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 PowerGREP5.pgl library as "Logs: 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”.

56. Split Database Dumps

Database dumps often produce extremely large files that are difficult to deal with outside your database software. PowerGREP can easily split the dump for an entire server into separate dumps for each database, or the dump for an entire database into separate dumps for each table. This gives you smaller files that you can work with in applications like text editors.

1. Mark the SQL files with your database dumps in the File Selector.
2. Start with a fresh action.
3. Set the action type to "split files".
4. Enter `` ^CREATE DATABASE[^`'";]*[`'"](\w+)[`'"].*?(?=^CREATE DATABASE|\z) `` into the Search box.
5. Enter a path like `c:\\output\\$1.sql` into the Target File box.
6. Turn on "dot matches newlines" to allow the regex matches to span across lines.
7. Click the Quick Split button to execute the action.

You want to use the Quick Split button rather than the Split Files button so that PowerGREP will not waste time and memory to try to display your entire database dump on the Results panel. When the action completes, you'll have one file database_name.sql for each database that was in your dump files.

The regular expression needs to match the entire block of SQL for one database in your dump. The above regex assume the block begins with a CREATE DATABASE statement at the start of a line. This is matched with `` ^CREATE DATABASE[^`'";]*[`'"](\w+)[`'"] ``, which also captures the name of the database. The regex also assumes that the block ends just before the next CREATE DATABASE statement or at the end of the file. This position is matched with `(?=^CREATE DATABASE|\z)`. Between that `.*?` matches anything.

If your dump contains the tables for a single database, you can use the regular expression `` ^CREATE TABLE[^`'";]*[`'"](\w+)[`'"].*?(?=^CREATE TABLE|\z) `` to split it into separate files for each table.

If your dumps don't start with CREATE DATABASE or CREATE TABLE, then you may need to use different regexes entirely. For example, mysqldump begins each database dump with a comment:

```
--
-- Current Database: `database_name`
--
```

If you split these dumps with the regex from the steps above then the split files will have the comment for the next database in the log at the bottom of the file. To make sure you get the correct comment at the top of the file you can use this regex:

```
^--
-- Current Database: `(\w+)`.*?(?=^--
-- Current Database: `|\z)
```

This regex contains two literal line breaks. In PowerGREP, when "search type" is set to "regular expression" (as opposed to "free-spacing regular expression"), a literal line break in a regex matches a line break of any style. So by using literal line breaks in this regex you can make sure that it will work correctly regardless of whether your database dump uses UNIX or Windows line breaks.

The `$1` in the target file path is replaced with the contents of the first capturing group. In this case that's the name of the database or table. If you're splitting multiple dump files that may contain dumps for databases with the same name, then the above Target File setting will create only one file for each unique name that contains the dump for all the databases with the same name. If you want to separate those too you can use a target path like `c:\\output\\%FILENAMENOEXT%_$1.sql` to add the names of the original dump files to those of the split files.

57. 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><A HREF="%FILENAME%">\1</A></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:

12.
    ```
    <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 PowerGREP5.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 PowerGREP5.pgl library as "Indexing HTML files with first paragraph".

58. 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 PowerGREP5.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 PowerGREP5.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 "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 "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.

59. 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 '<B>\1</B><BR>';`
9. `  } else {`
10. `    print '<A HREF="%FILENAME%">\1</A><BR>';`
    `  }`

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 PowerGREP5.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.

60. 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 PowerGREP5.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

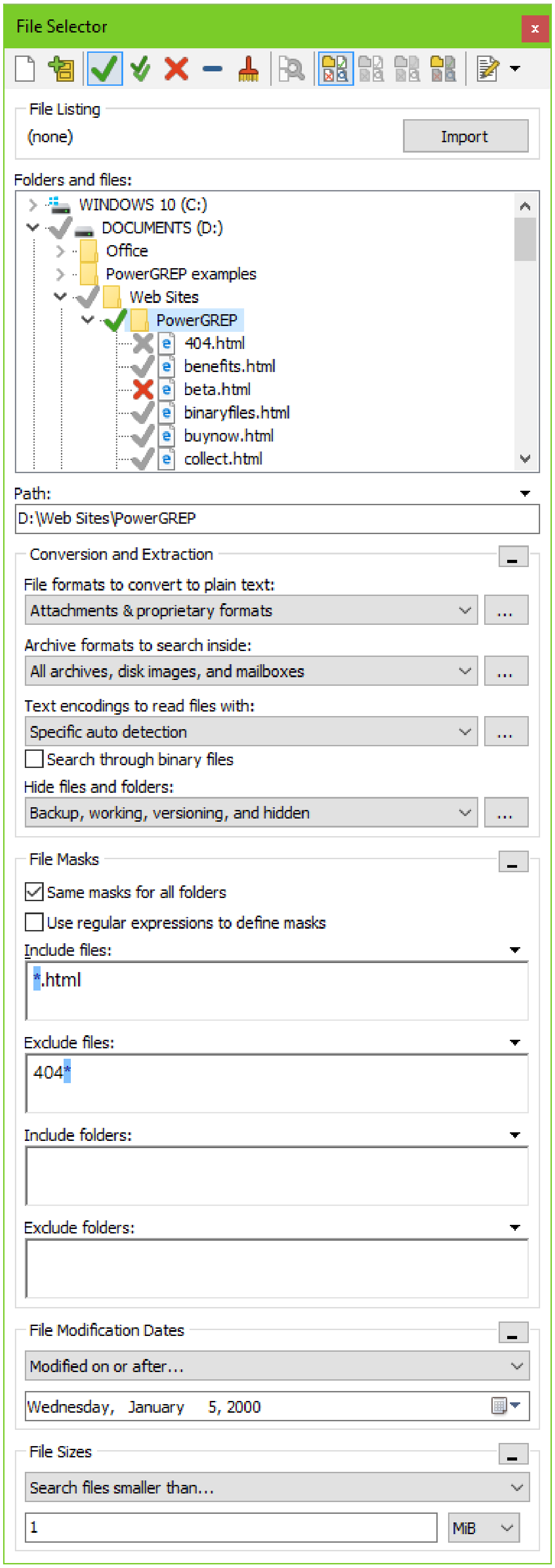
If you already have a text file with the list of files or folders that you want PowerGREP to search through, click the Import button to show the Import File Listings screen. This screen provides a variety of options to tell PowerGREP how to extract paths from the text file. You can tell PowerGREP to 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, except those hidden by the "hide files and folders" configuration that you can select further down on the File Selector panel. If the tree is too small to work with, click the minimize buttons on the sections below the tree on the File Selector panel. Minimizing those sections allows the tree to take up their space. You may need to minimize all of them if you're working on a small screen.

The **Network** node provides 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 entered a UNC path.

The **Clipboard** node provides access to the contents of the Windows clipboard. There may be multiple files under this node. When applications copy text to the



File Selector

File Listing

(none)

Import

Folders and files:

WINDOWS 10 (C:)
DOCUMENTS (D:)
Office
PowerGREP examples
Web Sites
PowerGREP
404.html
benefits.html
beta.html
binaryfiles.html
buynow.html
collect.html

Path:

D:\Web Sites\PowerGREP

Conversion and Extraction

File formats to convert to plain text:

Attachments & proprietary formats

Archive formats to search inside:

All archives, disk images, and mailboxes

Text encodings to read files with:

Specific auto detection

□ Search through binary files

Hide files and folders:

Backup, working, versioning, and hidden

File Masks

☑ Same masks for all folders

□ Use regular expressions to define masks

Include files:

*.html

Exclude files:

404*

Include folders:

Exclude folders:

File Modification Dates

Modified on or after…

Wednesday, January 5, 2000

File Sizes

Search files smaller than…

1 MiB

Windows clipboard, they can make the same text available in different formats. PowerGREP recognizes the plain text, rich text, and HTML clipboard formats. You should mark only one file under the Clipboard node in your action to make PowerGREP search through the contents of the clipboard in one specific format. Which format is best depends on the application that copied the text to the clipboard.

The **Editor** node provides access to the contents of the Editor panel. There is always one node under this file with the name of the file open in PowerGREP's built-in editor or untitled.txt if no file is open. Mark this file to make PowerGREP search through the contents of the Editor panel.

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 button with the green tick mark 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 appears next to the file, indicating it is marked for inclusion in the next action. The toolbar button stays down to indicate that the currently selected file or folder is included.

To include all the files in a particular folder, invoke Include File or Folder on the folder. A green tick mark appears next to the folder. Gray tick marks appear next to the 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 are not included and do not get tick marks.

To include all files in a particular folder and all of its subfolders, select the folder and invoke the Include Folder and Subfolders command by selecting it from the File Selector menu, clicking the button with the double green tick mark, or pressing Shift+Ctrl+I. A double green tick mark appears next to the folder you marked. The toolbar button stays down to indicate that the currently selected folder is included along with its subfolders. Double gray tick marks appear next to those subfolders. All the files in the marked folder and its subfolders get gray tick marks.

Some files inside an included folder may not get gray tick marks or may get gray X marks. That means they are not included in the action because of the settings below the folders and files tree on the File Selector panel, as described below.

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.

If a file is included (gray tick) because you marked the folder it is in, you can exclude it from the action by selecting Exclude File or Folder in the File Selector menu, clicking the toolbar button with the red X, or pressing Ctrl+Y. A red X replaces the gray tick next to the file. The toolbar button stays down to indicate that the currently selected file or folder is excluded.

If a folder is is included (double gray tick) because you marked its parent folder, you can exclude it with Exclude File or Folder. This also indirectly excludes all files and subfolders of the excluded folder, removing their gray tick marks. You can still explicitly mark files and subfolders of an excluded folder with a green tick mark to include them.

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 in the File Selector menu or by clicking the toolbar icon with the blue strikeout line or by pressing Ctrl+L. The toolbar button stays down to indicate that the selected files and folders are not directly included or excluded. 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 appears after you clear the green tick or red X.

When you select files or folders in the tree the toolbar buttons with the green tick, double green tick, red X, and blue strikeout change state to indicate the state of the selected files and folders. If all of them are marked with a green tick, double green tick, or red X, then that button is shown in its "down" state. If none of them are marked with any of these 3 marks then the blue strikeout button is shown in its "down" state, regardless of whether any of the files or folders are indirectly included or excluded as indicated with gray ticks or gray X marks. If the selected items have a mixture of marks, then all four toolbar buttons will be in their "up" state. So basically, these four buttons are used to both change the marks on files and folders and to indicate the marks they have.

To remove the marks from a folder and all the files and subfolders it contains, use the Clear Folder and its Files and Subfolders menu item or click the button with the broom on the toolbar. This removes all green and double green tick marks and all red X marks. Gray tick marks will remain if a parent folder is still marked for inclusion.

To start from scratch, select Clear in the File Selector menu or click the button with the empty document. This clears all marks in the folders and files tree, selects all the default configurations, and removes all file masks, file dates, and file sizes.

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. To access the network share "share" on the server "server", for example, enter \\server\share\ including the final backslash. That share then appears 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, enter the first path and press (Shift+)Ctrl+I. The text in the Path field will become selected, so you can immediately enter 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.

Conversion and Extraction

You can select four different configurations that control how PowerGREP deals with certain files.

- File format configuration: Specify which files, if any, should be treated as files (in proprietary file formats) that need to be converted into plain text before they can be searched through. If you select none of the options for a particular file format, then files matching that format's file mask are excluded from the action. They get gray X marks if they are inside a folder that is included.
- Archive format configuration: Specify which files, if any, should be treated as compressed archives (such as ZIP files), disk images (such as ISO files), or mailboxes (such as PST files) that contain other files or email messages that need to be searched through. Archive formats that you enable appear as folders in the folders and files tree. Archive formats that you disable appear as files. They are excluded from the action. They get gray X marks if they are inside a folder that is included.
- Text encodings to read files with: Specify which text encodings should be used to interpret plain text files.
- Hide files and folders: Specify which files and folders, if any, should be completely invisible in the files and folder tree. Those files and folders cannot be searched through.

Search Through Binary Files

Turn on to search through files that have been configured or detected as binary files by the text encoding configuration. Turn off to skip those files.

Skipping binary files does not exclude them from the search as the automatic detection of binary files is done during the search. Binary files retain their gray tick marks in the files and folders tree. The Results panel indicates skipped binary files in its bottom half, if there are any.

File Masks

With file masks you can include or exclude files by their names or extensions. You can use traditional file masks, or regular expressions. Simply clear or tick "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 `.` in a regular expression. 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. To search through all web logs from September 2003, for example, 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. 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. The regular expression `.*\.txt`, however, matches 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.

Include Files And Exclude Files

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. The other files lose their gray tick marks. 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 you specify an exclusion mask, all files matching the exclusion mask are excluded form the next action. Those files get gray X marks if they are inside a folder that is included. The exclusion mask takes precedence. If you specify both masks, files matching both are not searched through.

Masks only apply to files that were included because you marked the folder containing them. If you directly mark a file with a green tick mark then file masks do not apply to that file and it is always searched through.

In the screen shot above, the folder “PowerGREP” was marked for inclusion with Include File or Folder, as evidenced by the green tick next to it. The file “404.html” is excluded because it matches the exclusion mask, indicated by the gray X mark. The file “beta.html” is excluded with the Exclude File or Folder command, indicated by the red X mark.

By default, the same file masks are used for all folders that you marked for inclusion. If you want to use different marks 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

Include Folders and Exclude Folders

You can also use file masks to include or exclude folders based on their names. If you use the double tick mark to include a folder with its subfolders, and you specify a file mask for “include folders”, then files within the marked folder or its subfolders are only included if the name of their immediate parent folder matches the “include folders” mask. Those files retain their gray tick marks, while the other files lose their gray tick marks.

This also applies to files directly inside the folder that you marked with the double tick mark. Other settings such as “include files” can further reduce the list of files that are searched through, but never add files to the search if their immediate parent folder failed to match “include folders”.

If a folder is included along with its subfolders (double tick mark), and you specify a file mask for “exclude folders”, then the folder that you marked and any subfolders of it are excluded from the search if their name matches the “exclude folders” mask. Files and folders inside those excluded folders are all excluded, even if they match “include folders” or “include files” masks. The excluded folders get gray X marks, and the files and folders inside them lose their gray tick marks. File and folders inside folders excluded by “exclude folders” can still be included by marking them for inclusion with a green tick mark.

If you mark a folder to be included without its subfolders (single tick mark), then the files in that folder are included in the search, regardless of whether or not that folder matches the “include folders” mask or the “exclude folders” mask.

Folder names normally do not have extensions. Do not include a dot in file masks for folder names unless you specifically want to require a dot in the folder’s name.

File Masks on Relative Paths

Normally, file masks do not include any backslashes. They are applied to the name of the or folder only. But PowerGREP allows you to use backslashes in file masks. When you do, the file mask is compared with the file’s path relative to the marked folder. If you marked "C:\My Documents\My Stuff“, for example, 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 that are enabled in the archive format configuration as (compressed) folders. File masks such as `*.zip` are applied to files rather than to folders. That means you cannot set “include files” or “exclude files” to `*.zip` to control whether PowerGREP searches through .zip files. Instead, select or edit an archive format configuration to enable only those archive formats that you want to search through and which file extensions PowerGREP should recognize as archive formats.

If you want to search through specific archives based on their names, select an archive format configuration that enables the format of those archives. Then enter the masks that match the names of the archives you do or do not want to search through in “include folders” or “exclude folders”.

This rule does not apply to disabled archive formats when you execute a list files action without a search text, a file name search action, or a rename files action. In those situations, PowerGREP treats archives as ordinary files if they use an archive format that you configured not to be searched inside of. File masks such as `*.zip` do work in that situation, and the “list files” action will list the zip files themselves in the results.

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. The date filter is only applied to files that are indirectly included because you marked their folder with a single or double green tick mark. Files that you marked directly with a green tick mark ignore the date filter. Files that fail the date filter are indicated with gray X marks.

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. If at half past three, for example, you tell PowerGREP to search though files modified during the last two hours, then PowerGREP searches 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

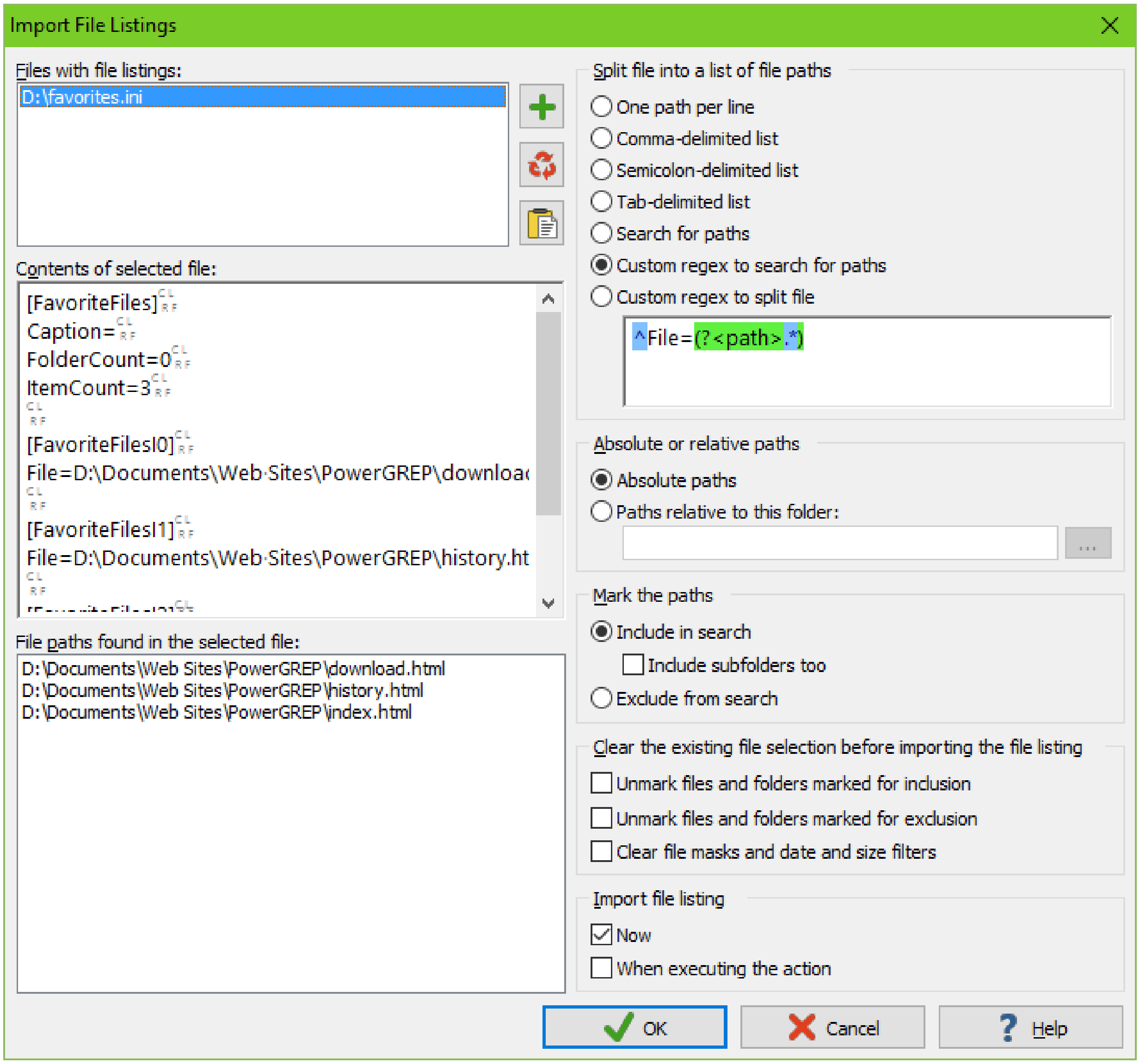
You can also further reduce the files that will be searched through by filtering them by their sizes. The size filter is only applied to files that are indirectly included because you marked their folder with a single or double green tick mark. Files that you marked directly with a green tick mark ignore the size filter. Files that fail the size filter are indicated with gray X marks.

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 file sizes in seven different units:

- bytes
- kB: kilobytes (1,000 bytes)
- MB: megabytes (1,000,000 bytes)
- GB: gigabytes (1,000,000,000 bytes)
- kiB: kibibytes (1,024 bytes)
- MiB: mebibytes (1,048,576 bytes)
- GiB: gibibytes (1,073,741,824 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.



Import File Listings

Files with file listings:

D:\favorites.ini

Contents of selected file:

```
[FavoriteFiles]
Caption=
FolderCount=0
ItemCount=3

[FavoriteFiles\0]
File=D:\Documents\Web Sites\PowerGREP\downloa

[FavoriteFiles\1]
File=D:\Documents\Web Sites\PowerGREP\history.ht
```

File paths found in the selected file:

```
D:\Documents\Web Sites\PowerGREP\download.html
D:\Documents\Web Sites\PowerGREP\history.html
D:\Documents\Web Sites\PowerGREP\index.html
```

Split file into a list of file paths

○ One path per line
○ Comma-delimited list
○ Semicolon-delimited list
○ Tab-delimited list
○ Search for paths
● Custom regex to search for paths
○ Custom regex to split file

```
^File=(?<path>.*)
```

Absolute or relative paths

● Absolute paths
○ Paths relative to this folder:

Mark the paths

● Include in search
□ Include subfolders too
○ Exclude from search

Clear the existing file selection before importing the file listing

□ Unmark files and folders marked for inclusion
□ Unmark files and folders marked for exclusion
□ Clear file masks and date and size filters

Import file listing

☑ Now
□ When executing the action

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 recycle button deletes the selected file from the list. You can click the Paste button to add a reference to the clipboard as a file with file listings.

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 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 folders and files 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're importing from the clipboard, PowerGREP uses the text that's on the clipboard at the time you execute the action.

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 folders and files tree as well as make sure that PowerGREP always uses the latest file listings.

4. File Format Configuration

The file format configuration tells PowerGREP which files it should convert to plain text prior to searching through them. It can also tell PowerGREP to exclude files. If you select a configuration that exclude files, those files immediately lose their gray tick marks in the folders and files tree.

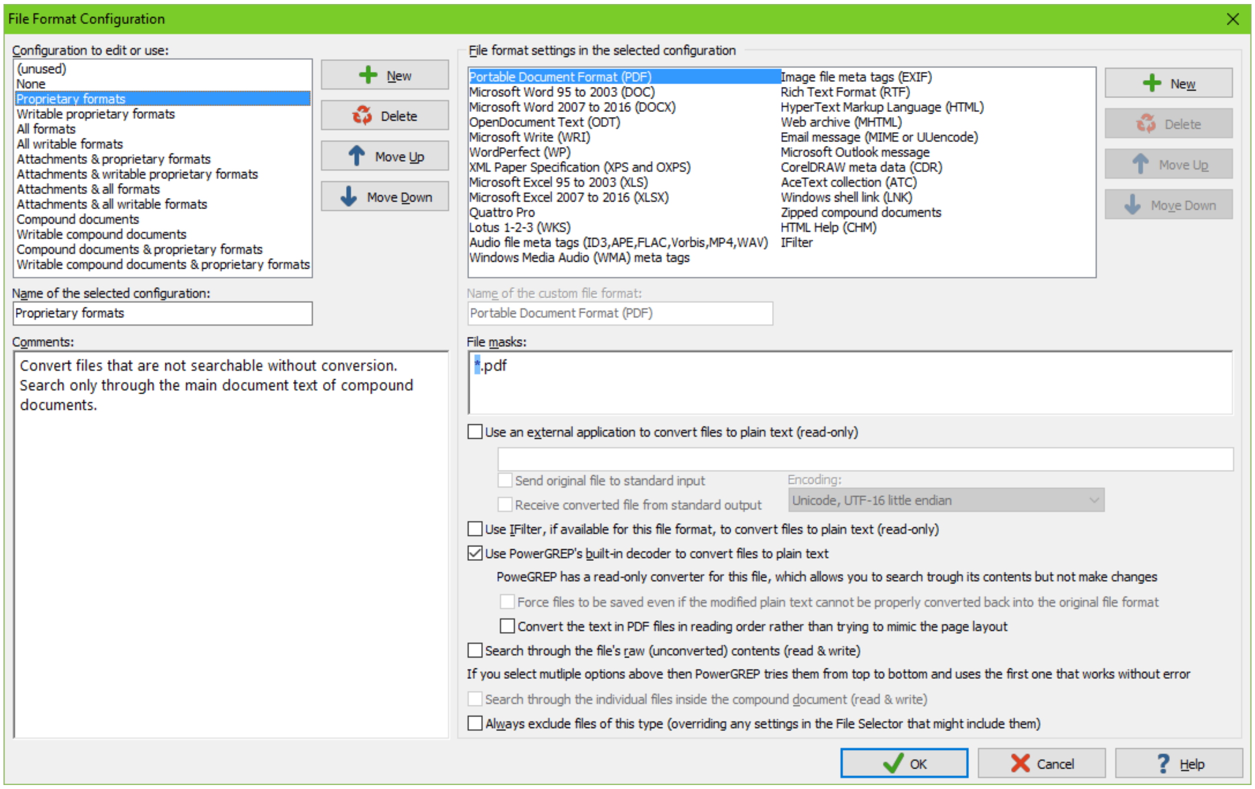
- **(unused):** Do not even check whether files can be converted to plain text. This configuration particularly useful for actions that do not search through the contents of files. It makes other actions search through the raw, undecoded contents of all files.
- **None:** Exclude all files in proprietary formats and all compound documents from the search. Do search through files that are human-readable without conversion.
- **Proprietary formats:** Convert and search through files that do not contain any human-readable text when viewing the raw, undecoded contents of the file, like PDF and DOC files. Convert compound documents like DOCX to plain text too. Process compound documents like CHM for which PowerGREP does not have a converter as archives.
- **Writable proprietary formats:** Convert and search through files in proprietary file formats like DOCX for which PowerGREP has a two-way converter that can write changes you make by searching and replacing through the plain text conversion to be written back to the file in its proprietary file format. Exclude files in proprietary formats like PDF and DOC for which PowerGREP has a read-only converter.
- **All formats:** In addition to files in proprietary formats, convert files in formats that are human-readable (to an extent) like HTML and RTF without conversion, but that are much easier to read when converted to plain text. (Except the “all formats” configurations, all other configurations search through the raw contents of files that are human-readable without conversion.)
- **All writable formats:** Convert all formats for which PowerGREP has a two-way converter. Exclude proprietary files for which PowerGREP has a read-only converter. Search through the raw contents of non-proprietary formats for which PowerGREP has a read-only converter.
- **Attachments & proprietary formats:** Process file formats such as email messages that are conceptually documents with attachments as archives, searching through both the document or message itself and its attachments, instead of searching through just the message text. Convert other compound documents and files in other proprietary formats to plain text. Do not convert human-readable formats such as HTML.
- **Attachments & writable proprietary formats:** Process documents with attachments as archives. Convert proprietary formats for which PowerGREP has a two-way converter. Exclude other proprietary formats.
- **Attachments & all formats:** Process documents with attachments as archives. Convert all other files that PowerGREP can convert to plain text, including human-readable formats such as HTML.
- **Attachments & all writable formats:** Process documents with attachments as archives. Convert all other files for which PowerGREP has a two-way converter to plain text, including human-readable formats such as HTML.
- **Compound documents:** Process all compound documents (files that are conceptually documents but technically ZIP or other archives) like DOCX as archives, searching through the files inside the archives, even if PowerGREP could convert the files to plain text. Exclude files in proprietary formats that are not compound documents like PDF and DOC.
- **Writable compound documents:** Process compound documents like DOCX that PowerGREP can write to. Exclude other compound documents and proprietary formats.
- **Compound documents & proprietary formats:** Process all compound documents like DOCX as archives, even if PowerGREP could convert the files to plain text. Convert files in other proprietary formats like PDF and DOC to plain text.

- **Writable compound documents & proprietary formats:** Process compound documents like DOCX that PowerGREP can write to. Convert proprietary formats for which PowerGREP has a two-way converter. Exclude other compound documents and proprietary formats.

These options obviously only affect hose proprietary formats and compound formats that PowerGREP knows about. Files in proprietary formats that PowerGREP cannot convert to plain text are treated as raw binary files.

Examples: Search through Microsoft Word documents, Search and replace through Microsoft Word documents, Search through PDF files, Search through XPS and OXPS files, Search through OpenOffice Writer documents, Search through OpenDocument Format files, Search through spreadsheets, Search and Edit Audio File Meta Data and Search and Edit EXIF and IPTC Image Meta Data

Editing File Format Configurations



Configuration to edit or use:

(unused)
None
Proprietary formats
Writable proprietary formats
All formats
All writable formats
Attachments & proprietary formats
Attachments & writable proprietary formats
Attachments & all formats
Attachments & all writable formats
Compound documents
Writable compound documents
Compound documents & proprietary formats
Writable compound documents & proprietary formats

Name of the selected configuration: Proprietary formats

Comments: Convert files that are not searchable without conversion. Search only through the main document text of compound documents.

File format settings in the selected configuration

Portable Document Format (PDF)
Microsoft Word 95 to 2003 (DOC)
Microsoft Word 2007 to 2016 (DOCX)
OpenDocument Text (ODT)
Microsoft Write (WRI)
WordPerfect (WP)
XML Paper Specification (XPS and OXPS)
Microsoft Excel 95 to 2003 (XLS)
Microsoft Excel 2007 to 2016 (XLSX)
Quattro Pro
Lotus 1-2-3 (WKS)
Audio file meta tags (ID3,APE,FLAC,Vorbis,MP4,WAV)
Windows Media Audio (WMA) meta tags
Image file meta tags (EXIF)
Rich Text Format (RTF)
HyperText Markup Language (HTML)
Web archive (MHTML)
Email message (MIME or UUencode)
Microsoft Outlook message
CorelDRAW meta data (CDR)
AceText collection (ATC)
Windows shell link (LNK)
Zipped compound documents
HTML Help (CHM)
IFilter

Name of the custom file format: Portable Document Format (PDF)

File masks: *.pdf

☐ Use an external application to convert files to plain text (read-only)

☐ Send original file to standard input
☐ Receive converted file from standard output

Encoding: Unicode, UTF-16 little endian

☐ Use IFilter, if available for this file format, to convert files to plain text (read-only)

☑ Use PowerGREP's built-in decoder to convert files to plain text

PoweGREP has a read-only converter for this file, which allows you to search trough its contents but not make changes

☐ Force files to be saved even if the modified plain text cannot be properly converted back into the original file format

☐ Convert the text in PDF files in reading order rather than trying to mimic the page layout

☐ Search through the file's raw (unconverted) contents (read & write)

If you select mutliple options above then PowerGREP tries them from top to bottom and uses the first one that works without error

☐ Search through the individual files inside the compound document (read & write)

☐ Always exclude files of this type (overriding any settings in the File Selector that might include them)

Click the (...) button next to the "file formats to convert to plain text" drop-down list on the File Selector panel to edit the file format configurations, or just to see their details.

The list on the left shows the available file format configurations. Select one to see its settings or edit it. You can edit all configurations. You can even delete all the configurations. If you delete them all and do not add your own, PowerGREP restores the configurations that were predefined when you first installed PowerGREP.

If you edit a configuration presently selected on the File Selector panel, those changes take effect immediately. But editing configurations does not change the behavior of previously saved file selections.

When you save a file selection, it stores the full details of the selected configurations. When you load a file selection, it continues to use the configuration you saved it with. If you edited that configuration between the time you saved and loaded the file selection, then the configuration loaded with the file selection is indicated with a number such as (2) to indicate its details are different from the configuration with the same name in the Preferences. If you want the loaded file selection to use the edited configuration, then you need need to select the edited configuration (without the number in parenthesis) on the File Selector panel after loading the exiting file selection. If you click the (...) button, both the edited configuration and the loaded configuration are shown in the dialog.

Each configuration has a name that identifies it on the File Selector panel. You can also add comments to explain in which situations you want to use this configuration.

The list on the right shows all the file formats that are configured in the selected configuration. File formats that PowerGREP has built-in support for cannot be moved or deleted. You can completely disable them though, as explained below. The predefined “(unused)” configuration disables them all.

You can add new file formats to the list. This is only useful if you have an external converter or an IFilter that can convert the new file format to plain text. New file formats should be given a name so you can easily identify their settings when editing the file configuration. Adding a file format only adds it to the configuration you’re editing.

To enable a file format, you need to specify one or more **file masks** that match the files in this format. You can use the full syntax for traditional file masks as explained in the help topic about the File Selector panel. File masks are applied to the full file name, not just the extension. You can even use backslashes in file masks if you want the file mask to be applied to the full path of the files instead of just their names.

PowerGREP can use **external applications** to convert files to plain text. These need to be command line applications so that no windows pop up when PowerGREP invokes them. Tick the checkbox to use any external application. In the edit box below, enter the full path to the application’s executable. Enclose the path with double quotes if it contains spaces. You can use %APPPATH% as a placeholder to PowerGREP’s installation folder if you put the external application is in the same folder. Enter any command line arguments the application need after the path to its executable. You can use "%INFILE%“ as a placeholder to the full path of the file to be converted. You can use ”%OUTFILE%“ as a placeholder to the full path of the plain text file that the application should create. Include the quotes around ”%INFILE%“ and ”%OUTFILE%" on the command line. The paths may contain spaces and most command line applications need quotes around paths with spaces.

If the application reads the file to be converted from standard input, tick the option “send original file to standard input”. If it writes the plain text conversion to standard output, tick the option “receive converted file from standard output”.

You will also need to specify the encoding that the application uses when writing its plain text conversion. This makes sure PowerGREP uses the correct encoding when reading the plain text conversion.

If you are developing this external converter yourself, you should make it terminate with a non-zero exit code to signal error conditions. Terminate with exit code 1 to indicate the file could not be opened. Terminate with exit code 2 to indicate the file is not in the file format that your converter supports. Terminate with exit code 3 if the file is protected with a password. Terminate with exit code 4 to indicate failure without a specific reason. PowerGREP then adds an error for the file that could not be converted to the search results. Without an exit code, PowerGREP cannot distinguish between a file that contains no text and a file that could not be converted. It will accept the empty conversion as a proper conversion.

Example: Search through UOT files

The **IFilter** system is a DLL-based system used by Windows Search. Developers of software that uses proprietary file formats can include an IFilter DLL with their software to enable Windows Search to search through files in that format. Because Windows Search can only search, this system is read-only. Because Windows Search only displays a list of matching files, rather than a list of search matches with context like PowerGREP does, the plain text conversion produced by an IFilter have all text strung together rather than mimicking a page layout. You should only enable the IFilter option if you know you have a reliable IFilter installed for the selected file format. IFilter DLLs are registered by file extension, so the option will only work correctly for files with extensions for which an IFilter is actually registered.

PowerGREP's **built-in** decoders are truly built-in. They do not require any software to be installed, other than PowerGREP itself. Most of these converters are read-only, allowing you to search through the files, but not make replacements. Some converters are read-write. These allow you to search and replace through the plain text conversions and have the replacements written back to the file in its original format.

Examples: Search through Microsoft Word documents, Search and replace through Microsoft Word documents, Search through PDF files, Search through XPS and OXPS files, Search through OpenOffice Writer documents, Search through spreadsheets and Search through mailboxes and email messages

Depending on the replacements you've made, it may not always be possible to correctly write them back to the original format. This mostly happens when you're doing something that's not sensible for the file format you're working with. For example, the plain text conversion of audio and image files consists of labels and values like "Subject: My First Photo". Replacements you make in the value, such as replacing "First" with "1st" in this example, will always be written back correctly. But you can't make arbitrary replacements in the labels. If you were to replace "Subject" with "Topic", PowerGREP doesn't know how to deal with that, because "Topic" is not one of the labels it uses for image metadata.

PowerGREP can deal with replacements it can't convert back into the original format in two ways. The default way is to not save any replacements to the file, adding an error message for that file to the results instead. The other way is to force the file to be saved, which you can choose by ticking the "force files to be saved" checkbox. PowerGREP then makes a best effort to save the replacements, ignoring any replacements that it can't save.

Examples: Search and Edit Audio File Meta Data and Search and Edit EXIF and IPTC Image Meta Data

For PDF files, there is an extra option. The plain text conversion of PDF files can mimic the page layout, or it can show the text in reading order. The difference is most obvious with PDF files that have text in columns. When mimicing the page layout, the plain text conversion also has the text in two columns. In reading order, the plain text conversion puts all the text of the first column before all the text of the second column. Reading order makes it easier to search for text that may span across multiple lines in one column.

Example: Search through PDF files

If the file contains (some) human-readable content even without conversion, then you can enable the option to search through the file's **raw contents**. If you turn on this option and turn off all others, then you're effectively telling PowerGREP to treat files in this format like plain text files or raw binary files that do not need conversion. This can be useful for file formats like HTML or RTF that PowerGREP can convert to plain text, but that are also searchable in their original form if you're familiar with their structure.

Examples: Search and Edit Audio File Meta Data and Search and Edit EXIF and IPTC Image Meta Data

You can turn on several of the above options at the same time. Any combination is permitted. PowerGREP will try the conversions from top to bottom and will use the first one that succeeds. The “raw contents” option always succeeds. If you don’t select that one and all the other selected ones fail, PowerGREP skips the file and adds an error message to the results.

Some file formats can be treated as **compound documents**. This includes file formats that are technically ZIP files as well as file formats for email messages that may contain attachments. This option cannot be used in combination with the other four options. Compound documents are treated as files when using the settings on the File Selector panel to determine whether the file should be searched through or not. If the file needs to be searched through, PowerGREP searches through all its constituent files. For email formats, it searches through both the email body and all attachments. Compound documents can be expanded in the folders and files tree on the File Selector panel to show and open the constituent files.

Examples: Search through Microsoft Word documents, Search and replace through Microsoft Word documents, Search through OpenDocument Format files and Search through mailboxes and email messages

Finally, the option to **always exclude** files overrides any other settings on the File Selector panel that may try to include files in this format. Such files never get gray tick marks in the File Selector. Directly marking such a file with a green tick results in an error when attempting to execute the action. This is useful for file formats for which PowerGREP only has read-only converters in configurations that you intend to use with search-and-replace actions. This makes PowerGREP skip files that it can’t make replacements in.

Predefined File Format Configurations

The table below shows the details of the file format configurations that are predefined in the default preferences when you first install PowerGREP. The "(unused)“ configuration is not shown. This configuration has no file masks for any of the file formats. This means it does not recognize any files as being in any of the file formats that PowerGREP can convert. The ”(unused)“ configuration is useful for actions like ”file and folder name search“ that do not search through the contents of files. Since those actions do not convert files anyway, you can select ”(unused)" to treat files equally, regardless of their file format. All the other configurations use the same file masks. They only differ in the chosen conversion options.

“Skip”, with a red background, means that files matching these file masks are never searched through when using that file format configuration. This overrides any other settings on the File Selector panel that may try to include those files. Directly marking such a file with a green tick results in an error when attempting to execute the action.

“Compound” means that the files will be treated as a compound document. A green background means that you can search and replace through the files inside the compound document. A white background means that you can search but not replace.

“Built-in” means that the files will be converted to plain text using PowerGREP’s built-in converter for their format. A green background means that you can search and replace through the plain text conversion and have the replacements written back to the file in its original file format. A white background means that you can only search through the plain text conversion.

“Raw”, with a green background, means that the files will be searched through without conversion.

"IFilter", with a white background, means that the files will be converted to plain text using an IFilter. The IFilter system is read-only, so you can only search through the plain text conversion.

Actions that modify files, like search-and-replace, only work with file format configurations that use options with red or green backgrounds for all file formats. If you try to run a search-and-replace with a configuration that uses a read-only converter (indicated with a white background) for at least one file format, then you will get an error telling you to select a different file format configuration. You will get this error even if the search does not actually include any files in the format for which you enabled a read-only converter. The error is triggered before PowerGREP gathers the list of files you want to search-and-replace through.

IncludeDocs(convertconfig.txt)

5. Archive Format Configuration

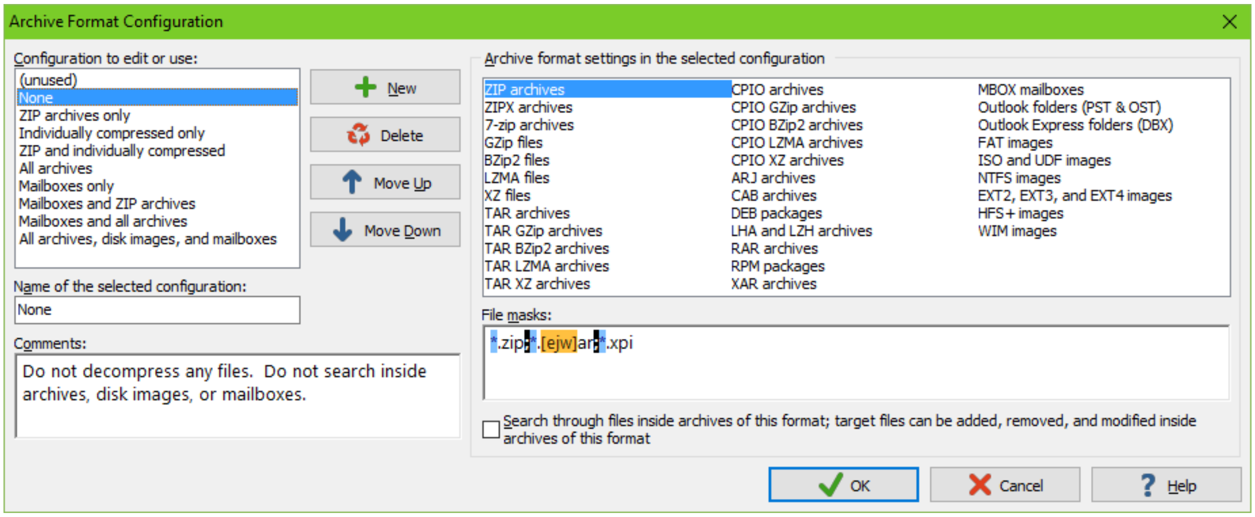
The archive format configuration tells PowerGREP which archives it should search through by treating them as compressed folders. All other archives are excluded from the search. If you select a configuration that includes archives, you will be able to see their contents in the files and folders tree. Files inside archives will get gray tick marks when the archive is included. To include all files in all archives in a particular folder, use the button with the double tick mark to include the folder. If you select a configuration that excludes archives, those archives immediately lose their gray tick marks in the folders and files tree. You will not be able to expand them to see their contents.

Some compression formats like GZ compress individual files. Those can never be expanded in the files and folders tree. Select a configuration that includes such files to search through their decompressed contents.

The target file and backup file settings on the Action panel can write files to archive formats that are excluded and will use the correct compression format, as long as the archive format configuration has the file masks to recognize the formats. That includes all predefined configurations except (unused).

- **(unused):** Do not even check whether files are archives. This configuration particularly useful for actions that do not search through the contents of files. It makes other actions search through the raw, undecoded contents of all archives.
- **None:** Exclude all archives, compressed files, mailboxes, and disk images.
- **ZIP archives:** Search through ZIP archives only. Exclude all other archives, compressed files, mailboxes, and disk images.
- **Individually compressed:** Search through files that are individually compressed using formats like GZ, BZIP2, or XZ. Exclude archives using formats that can contain multiple files.
- **ZIP and individually compressed:** Search through ZIP archives and files that are individually compressed using formats like GZ, BZIP2, or XZ. Exclude archives using formats other than ZIP that can contain multiple files.
- **All archives:** Search through all archives and all individually compressed files. Exclude mailboxes and disk images.
- **Mailboxes:** Search through mailboxes only. Exclude all other archives and compressed email attachments.
- **Mailboxes and ZIP archives:** Search through mailboxes and ZIP archives only.
- **Mailboxes and all archives:** Search through mailboxes, individually compressed files, and archive formats that are not disk images.
- **All archives and disk images:** Search through all archives, all individually compressed files, all mailboxes, and all disk images like ISO and WIM.

Editing Archive Format Configurations



Configuration to edit or use:

(unused)
None
ZIP archives only
Individually compressed only
ZIP and individually compressed
All archives
Mailboxes only
Mailboxes and ZIP archives
Mailboxes and all archives
All archives, disk images, and mailboxes

Name of the selected configuration: None

Comments: Do not decompress any files. Do not search inside archives, disk images, or mailboxes.

Archive format settings in the selected configuration

ZIP archives
ZIPX archives
7-zip archives
GZip files
BZip2 files
LZMA files
XZ files
TAR archives
TAR GZip archives
TAR BZip2 archives
TAR LZMA archives
TAR XZ archives
CPIO archives
CPIO GZip archives
CPIO BZip2 archives
CPIO LZMA archives
CPIO XZ archives
ARJ archives
CAB archives
DEB packages
LHA and LZH archives
RAR archives
RPM packages
XAR archives
MBOX mailboxes
Outlook folders (PST & OST)
Outlook Express folders (DBX)
FAT images
ISO and UDF images
NTFS images
EXT2, EXT3, and EXT4 images
HFS+ images
WIM images

File masks: *.zip;*.[ejw]ar;*.xpi

☐ Search through files inside archives of this format; target files can be added, removed, and modified inside archives of this format

Click the (...) button next to the "archive formats to search inside" drop-down list on the File Selector panel to edit the archive format configurations, or just to see their details.

The list on the left shows the available archive format configurations. Select one to see its settings or edit it. You can edit all configurations. You can even delete all the configurations. If you delete them all and do not add your own, PowerGREP restores the configurations that were predefined when you first installed PowerGREP.

If you edit a configuration presently selected on the File Selector panel, those changes take effect immediately. But editing configurations does not change the behavior of previously saved file selections. When you save a file selection, it stores the full details of the selected configurations. When you load a file selection, it continues to use the configuration you saved it with. If you edited that configuration between the time you saved and loaded the file selection, then the configuration loaded with the file selection is indicated with a number such as (2) to indicate its details are different from the configuration with the same name in the Preferences. If you want the loaded file selection to use the edited configuration, then you need need to select the edited configuration (without the number in parenthesis) on the File Selector panel after loading the exiting file selection. If you click the (...) button, both the edited configuration and the loaded configuration are shown in the dialog.

Each configuration has a name that identifies it on the File Selector panel. You can also add comments to explain in which situations you want to use this configuration.

The list on the right shows all the archive formats that PowerGREP supports. This list cannot be changed. Select a format in the list to change its settings.

To enable an archive format, you need to specify one or more **file masks** that match the files that should be treated as archives in this format. You can use the full syntax for traditional file masks as explained in the help topic about the File Selector panel. File masks are applied to the full file name, not just the extension. You can even use backslashes in file masks if you want the file mask to be applied to the full path of the files instead of just their names.

If you don’t specify a file mask for an archive format, then no files are ever recognized as being in that format. If you remove `*.zip` from the ZIP archives format, for example, then ZIP files are treated as ordinary files that may or may not be included in the action based on your other settings on the File Selector panel.

The label of the checkbox changes depending on the kind of archive format you’ve selected. For archives formats such as ZIP that can contain multiple files, the checkbox is labeled “search through files inside archives of this format”. It also indicates whether PowerGREP can modify files inside those archives or not. For formats such as GZip that compress a single file, the checkbox is labeled “decompress files of this format”.

But the actual meaning of the checkbox is always the same. If you tick the checkbox, PowerGREP searches inside archives or compressed files that match the format’s file mask. Archives that can contain multiple files are then treated as *folders* when using the settings on the File Selector panel to determine whether the archive should be searched through or not. You can expand such archives in the folders and files tree to see the files they contain. Formats that compress an individual file are always treated as files.

If you clear the checkbox, then files matching the archive format’s file masks are never searched through when using that archive format configuration. This overrides any other settings on the File Selector panel that may try to include those archives. They never get gray tick marks. Directly marking such a archive with a green tick results in an error when attempting to execute the action.

Examples: Search through mailboxes and email messages and Search through ZIP files and other archives

Predefined Archive Format Configurations

The table below shows the details of the archive format configurations that are predefined in the default preferences when you first install PowerGREP. The "(unused)“ configuration is not shown. This configuration has no file masks for any of the archive formats. This means it does not recognize any files as being in any of the archive formats that PowerGREP can decompress. The ”(unused)“ configuration is useful for actions like ”file and folder name search" that do not search through the contents of files. It allows you to treat all archives as ordinary files. All the other configurations use the same file masks. They only differ in the chosen conversion options.

“Search” indicates archive formats that are searched through. A green background indicates PowerGREP can also create such archives and add, delete, or replace files inside such archives. A white background indicates that PowerGREP cannot create or modify such archives. “Skip”, with a red background, indicates the archive format is disabled.

IncludeDocs(archiveconfig.txt)

Document Files That Use Archive File Formats

Some modern document formats are technically ZIP archives, though people see them as documents. The best known examples are the formats like DOCX and XLSX used by Microsoft Office 2007 and later. In PowerGREP, such file formats are configured in the File Format Configuration rather than in the Archive format configuration. In the File Format Configuration, you can select whether these files should be converted to plain text or whether they should be treated as compound documents. When treated as compound documents, PowerGREP searches inside the files inside the compound document as it would with a ZIP archive. But the files are still treated as files when determining whether they should be included in the action or not. Actions that copy or move files with results copy or move compound documents as a whole whereas they individually copy or move files inside archives.

If you want to search inside document format that are technically ZIP archives for which PowerGREP does not have built-in support, you should add those to the File Format Configuration. If you want to treat that document format as a compound document, just add its file mask to the "zipped documents" file format. If you want to convert the document format to plain text, add a new file format where you specify its file mask and the external converter or IFilter to convert it with.

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. If you used the WinZIP self-extractor, for example, 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. So it's best to create a new archive format configuration for this. Then you can select this configuration when executables should be treated as archives, and another configuration when they should not be.

There is no option to make PowerGREP attempt all the archive formats that support self-extracting archives on each .exe file. 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, archive format configurations 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 `*zips\*.exe` to differentiate archives from regular .exe files, then 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 can create two archive format configurations where one treats .war files as ZIP archives and the other treats them as compressed tarballs. You'll need to run two separate searches, one with each archive format configuration, in order to search through all your .war files.

The only way to search through both types of .war files at the same time is to come up with file masks that differentiate the two. The file masks that you can use in the archive format configuration 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, then you can use the file masks `*java\*.war` and `*linux\*.war` to separate them.

6. Text Encoding Configuration

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 a 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.

In order for PowerGREP to correctly search through and display the text in your files, PowerGREP needs to know which encoding to use to map the numbers stored in the file to characters. For files converted to plain text according to a file format configuration the encoding is determined as part of the conversion. For all other files, the encoding is determined by the Text Encoding Configuration that you select under "text encodings to read files with" on the File Selector panel. That includes files that need to be treated as raw files according to the File Format Configuration, as well as files not matched by any file masks in the File Format Configuration.

Predefined Text Encoding Configurations

PowerGREP ships with three predefined text encoding configurations. The "specific auto detection" configuration performs quick automatic Unicode detection on all files. It performs slower automatic detection of XML declarations and HTML charset tags only on files with extensions used for XML and HTML files. It also knows that certain file formats need special treatment. It tells PowerGREP not to add a byte order marker to XML files as many XML parsers choke on the BOM. It disables detection of NULL characters on RTF files, because RTF files sometimes end with a NULL.

The "generic auto detection" configuration performs all automatic text encoding detection methods on all files. It does not have specific settings for any kind of file.

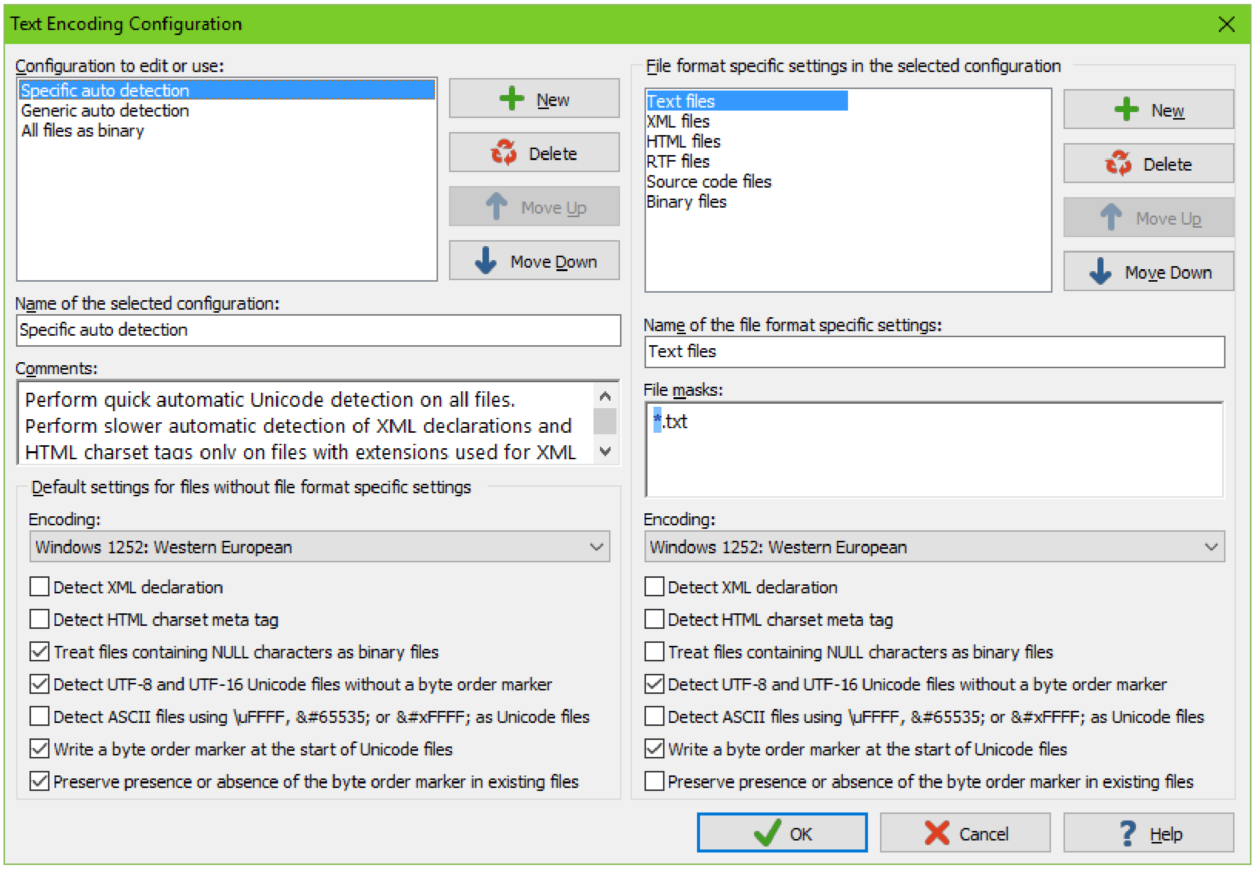
The "all files as binary“ configuration treats all files as binary files. This is useful for actions that use the ”binary data" search type.

Testing Text Encoding Configurations

Use the Open item in the Editor menu to open a text file. The text shown in the editor is the text that PowerGREP searches through when the file is included in an action. Make sure the text appears correctly. Pay particular attention to letters with diacritics or letters from non-Latin scripts if your file is supposed to contain those. Many legacy code pages are extensions of ASCII. Since ASCII covers the English alphabet, English text often comes out correctly even when using the wrong code page.

If the text is not correct, use the Editor|New menu to close the file. Select a different text encoding configuration on the File Selector panel or edit the configuration you were using. Then open the file again in the editor. The editor uses the text encoding settings only when opening a file. So you need to close and reopen the file for the editor to use the new text encoding configuration.

Editing Text Encoding Configurations



Configuration to edit or use:
Specific auto detection
Generic auto detection
All files as binary

Name of the selected configuration: Specific auto detection

Comments:
Perform quick automatic Unicode detection on all files.
Perform slower automatic detection of XML declarations and
HTML charset tags only on files with extensions used for XML

Default settings for files without file format specific settings

Encoding: Windows 1252: Western European

☐ Detect XML declaration
☐ Detect HTML charset meta tag
☑ Treat files containing NULL characters as binary files
☑ Detect UTF-8 and UTF-16 Unicode files without a byte order marker
☐ Detect ASCII files using \uFFFF, &#65535; or &#xFFFF; as Unicode files
☑ Write a byte order marker at the start of Unicode files
☑ Preserve presence or absence of the byte order marker in existing files

File format specific settings in the selected configuration
Text files
XML files
HTML files
RTF files
Source code files
Binary files

Name of the file format specific settings: Text files

File masks: *.txt

Encoding: Windows 1252: Western European

☐ Detect XML declaration
☐ Detect HTML charset meta tag
☐ Treat files containing NULL characters as binary files
☑ Detect UTF-8 and UTF-16 Unicode files without a byte order marker
☐ Detect ASCII files using \uFFFF, &#65535; or &#xFFFF; as Unicode files
☑ Write a byte order marker at the start of Unicode files
☐ Preserve presence or absence of the byte order marker in existing files

Click the (...) button next to the "text encodings to read files with" drop-down list on the File Selector panel to edit the text encoding configurations, or just to see their details.

The list on the left shows the available text encoding configurations. Select one to see its settings or edit it. You can edit all configurations. You can even delete all the configurations. If you delete them all and do not add your own, PowerGREP restores the configurations that were predefined when you first installed PowerGREP.

If you edit a configuration presently selected on the File Selector panel, those changes take effect immediately. But editing configurations does not change the behavior of previously saved file selections. When you save a file selection, it stores the full details of the selected configurations. When you load a file selection, it continues to use the configuration you saved it with. If you edited that configuration between the time you saved and loaded the file selection, then the configuration loaded with the file selection is indicated with a number such as (2) to indicate its details are different from the configuration with the same name in the Preferences. If you want the loaded file selection to use the edited configuration, then you need need to select the edited configuration (without the number in parenthesis) on the File Selector panel after loading the exiting file selection. If you click the (...) button, both the edited configuration and the loaded configuration are shown in the dialog.

Each configuration has a name that identifies it on the File Selector panel. You can also add comments to explain in which situations you want to use this configuration.

The list on the right shows all the file format specific settings in the selected configuration. You can add as many file formats to this list as you like. You can also leave this list blank if you want to use the configuration's default settings for all files. Each configuration has its own list of file formats. Adding or deleting file formats only affects the selected configuration.

Each file format has a name that identifies it in the list of file formats. To actually enable the format, you need to specify one or more **file masks** that match the files that should use the text encoding settings specific to that file format. You can use the full syntax for traditional file masks as explained in the help topic about the File Selector panel. File masks are applied to the full file name, not just the extension. You can even use backslashes in file masks if you want the file mask to be applied to the full path of the files instead of just their names.

Text Encoding Settings

The settings in the left hand section "default settings for files without file format specific settings" are used for all files that do not match any of the file masks that you added. The settings in the right hand section are used for the file format you selected in the right hand list. These two sets of settings are exactly the same. Only one set is used for each file, depending on whether it matches one of the file masks for specific settings, or not.

Encoding

The "encoding" is the character mapping or code page to interpret the file. In most cases you'll want to 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. The encoding you select in the drop-down list is used as long as the automatic detection options are turned off or don't detect anything.

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.

NULL Characters

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 ”search through binary files" checkbox. When you do search through binary files, PowerGREP displays 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.

Byte Order Marker

On the Windows platform, Unicode files should start with a byte order marker which is more accurately called a Unicode signature. 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.

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 Ã©. 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.

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.

Detect ASCII files using \uFFFF, &#65535; or &#xFFFF; as Unicode files

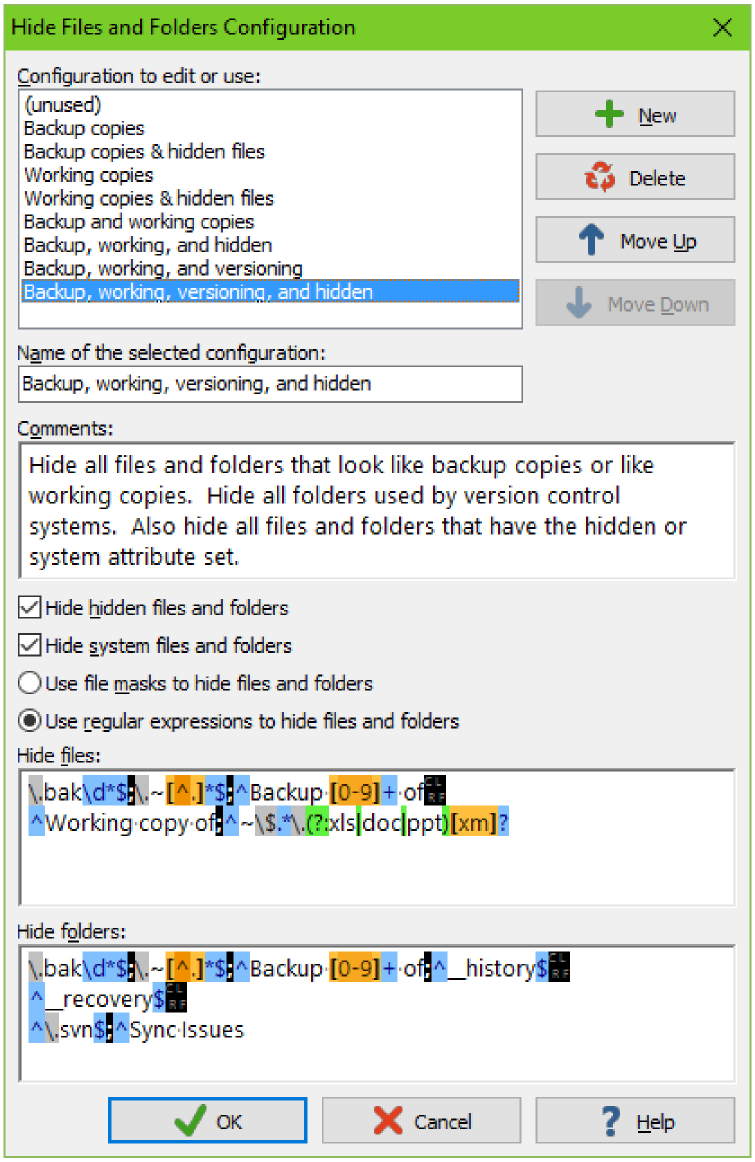
Some files consist only of ASCII characters and specify Unicode characters using numeric Unicode character escapes in the form of \uFFFF or using XML entities in the form of &#65535; and/or &#xFFFF;. Turn on this option to make PowerGREP treat these files as Unicode and transparently convert the Unicode 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.

7. Hide Files and Folders

Hiding files and folders makes them completely invisible to the File Selector and to PowerGREP itself. Such files and folders cannot be searched through unless and until you select a different configuration for hiding files and folders. Hiding files and folders that are never of interest is useful for reducing clutter in the File Selector and in Search Results. It differs from excluding files and folders in that excluded files and folders remain visible in the File Selector, so you can see what you've excluded.

The default configurations hide various sets of files that should generally be ignored when grepping:

- **Backup copies:** Hide all files and folders that match any of the backup naming styles supported by PowerGREP. Hiding these is recommended to make sure you don't accidentally mess up PowerGREP's backups, which would prevent you from undoing your actions.
- **Working copies:** Hide files that look like copies of files that you have presently open in applications such as Microsoft Office or EditPad Pro.
- **Version control:** Hide folders that store metadata for version control software like Subversion.
- **Hidden files:** Hide files and folders that have the hidden or system attribute set. These are hidden in Windows Explorer by default too.



Hide Files and Folders Configuration

Configuration to edit or use:

(unused)
Backup copies
Backup copies & hidden files
Working copies
Working copies & hidden files
Backup and working copies
Backup, working, and hidden
Backup, working, and versioning
Backup, working, versioning, and hidden

Name of the selected configuration: Backup, working, versioning, and hidden

Comments: Hide all files and folders that look like backup copies or like working copies. Hide all folders used by version control systems. Also hide all files and folders that have the hidden or system attribute set.

☑ Hide hidden files and folders
☑ Hide system files and folders
○ Use file masks to hide files and folders
● Use regular expressions to hide files and folders

Hide files:

```
\.bak\d*$;\.~[^.]*$;^Backup·[0-9]+·of
^Working·copy·of;^~\$.*\.(?:xls|doc|ppt)[xm]?
```

Hide folders:

```
\.bak\d*$;\.~[^.]*$;^Backup·[0-9]+·of;^__history$
^__recovery$
^\.svn$;^Sync·Issues
```

Editing Configurations for Hiding Files and Folders

Click the (...) button next to the "hide files and folders" drop-down list on the File Selector panel to edit the configurations for hiding files and folders, or just to see their details.

The list shows the available configurations. Select one to see its settings or edit it. You can edit all configurations. You can even delete all the configurations. If you delete them all and do not add your own, PowerGREP restores the configurations that were predefined when you first installed PowerGREP.

If you edit a configuration presently selected on the File Selector panel, those changes take effect immediately. But editing configurations does not change the behavior of previously saved file selections. When you save a file selection, it stores the full details of the selected configurations. When you load a file selection, it continues to use the configuration you saved it with. If you edited that configuration between the time you saved and loaded the file selection, then the configuration loaded with the file selection is indicated with a number such as (2) to indicate its details are different from the configuration with the same name in the Preferences. If you want the loaded file selection to use the edited configuration, then you need need to select the edited configuration (without the number in parenthesis) on the File Selector panel after loading the exiting file selection. If you click the (...) button, both the edited configuration and the loaded configuration are shown in the dialog.

Each configuration has a name that identifies it on the File Selector panel. You can also add comments to explain in which situations you want to use this configuration.

Each configuration has one group of settings. The two checkboxes, when ticked, make the configuration hide all files and folders that have the hidden attribute or the system attribute set.

The radio buttons let you choose whether you want to use traditional file masks or regular expressions to match the names of files and folders that you want to hide. The rules for these file masks or regular expressions are exactly the same as those for including and excluding files and folders on the File Selector panel.

One set of file masks is applied to the names of files. If one of the file masks in the list matches a file's name, that file becomes invisible to PowerGREP. The other set of file masks does the same for the names of folders.

Files and folders are hidden without taking any of the other settings on the File Selector into account. If a file or folder is hidden, the File Selector acts as if it does not exist at all. Since the archive format configuration is not taken into account, archives are affected by the "hide files" file masks, even though otherwise the File Selector treats them as folders.

8. 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. Restores all conversion and extraction configurations to their defaults. Removes all file masks. Changes the file size and file date settings to allow all sizes and dates.

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, use View|Lock Toolbars to unlock the toolbars if you haven't already. Then 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. When you're done, you can lock the toolbars again to prevent accidental changes.

Include File or Folder

Marks the file or folder you selected in the folders and files tree for inclusion in the next action. A green tick mark appears next to the file or folder, indicating it is marked for inclusion in the next action. The toolbar button stays down to indicate that the currently selected file or folder is included. 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 folders and files tree for inclusion in the next action. A double green tick mark appears next to the file or folder, indicating it is marked for inclusion in the next action. The toolbar button stays down to indicate that the currently selected folder is included along with its subfolders. Gray tick marks appear next to all files in the folder and its subfolders. Double gray tick marks appear next to the subfolders.

Exclude File or Folder

Excludes the file or folder you selected in the folders and files tree for inclusion from the next action, indicated by a red X. The toolbar button stays down to indicate that the currently selected file or folder is excluded. If you exclude a folder, all files and subfolders of the excluded folder are 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 folders and files 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 Based on Search Results

This command is only available after you have previewed or executed an action. There are three choices in the submenu. All three remove all inclusion and exclusion marks.

Mark Matched Files then individually marks for inclusion all files in which search matches were found during the previous action. Mark unmatched Files instead individually marks for inclusion all files that were searched through but in which no search matches were found during the previous action. Mark Target Files instead individually marks for inclusion all files that were created or overwritten during the previous action.

Exclude Files Based on Search Results

This command is only available after you have previewed or executed an action. There are three choices in the submenu.

Exclude Matched Files individually excludes all files in which search matches were found during the previous action. Exclude unmatched Files individually excludes all files that were searched through but in which no search matches were found during the previous action. Exclude Target Files individually excludes all files that were created or overwritten during the previous action.

All excluded files get red X marks next to them. All other files and all folders retain the inclusion or exclusion marks that they had before. Essentially, this command allows you to search again through the same set of files, minus those files in which you just found or did not find some search matches.

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. Unlike the Mark Files Based on Search Results command, this option does not change the file selection.

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 folders and files 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 folders and files 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 folders and files 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 folders and files tree will be blank.

Show Folders

Show only folders in the folders and files 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 folders and files 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.

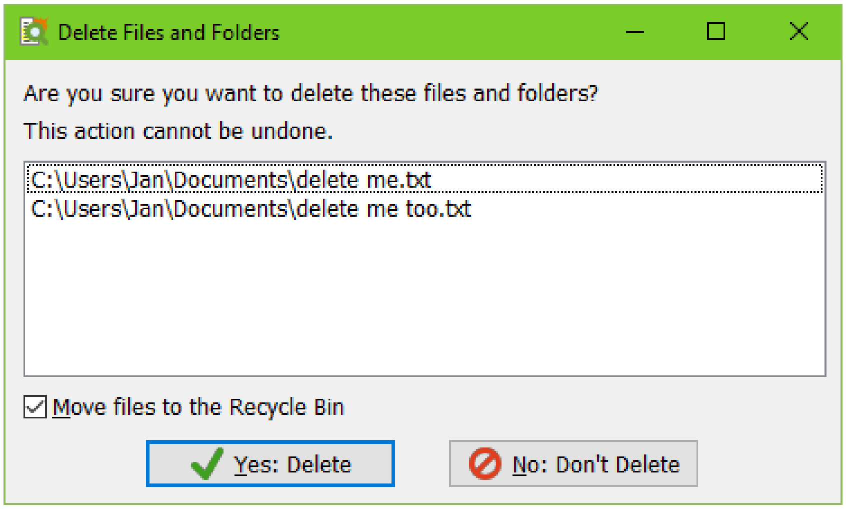
If you prefer to use an external editor or application to view or edit the file, first configure the editor or application in the external editors preferences. You can then click on the downward pointing arrow next to the Edit button on the toolbar, or the right-pointing arrow next to the Edit item in 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



Are you sure you want to delete these files and folders?

This action cannot be undone.

C:\Users\Jan\Documents\delete me.txt
C:\Users\Jan\Documents\delete me too.txt

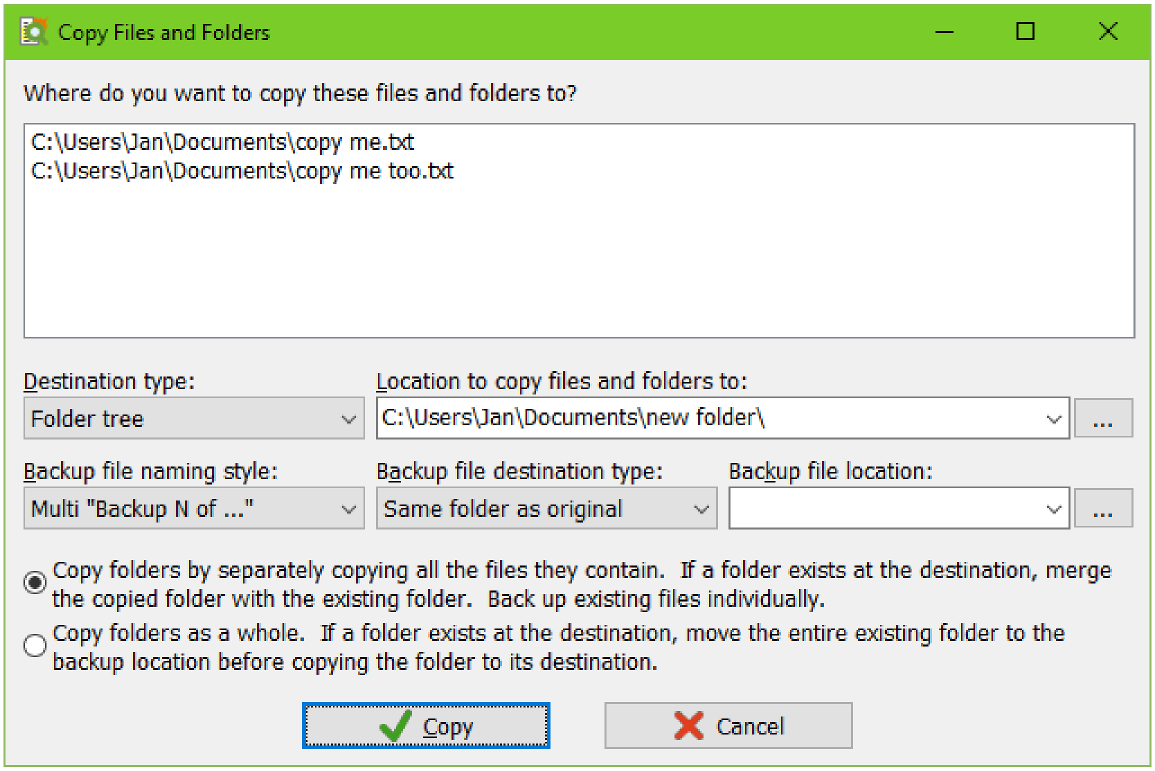
☑ Move files to the Recycle Bin

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



Where do you want to copy these files and folders to?

C:\Users\Jan\Documents\copy me.txt
C:\Users\Jan\Documents\copy me too.txt

Destination type: Folder tree

Location to copy files and folders to: C:\Users\Jan\Documents\new folder\

Backup file naming style: Multi "Backup N of ..."

Backup file destination type: Same folder as original

Backup file location:

● Copy folders by separately copying all the files they contain. If a folder exists at the destination, merge the copied folder with the existing folder. Back up existing files individually.

○ Copy folders as a whole. If a folder exists at the destination, move the entire existing folder to the backup location before copying the folder to its destination.

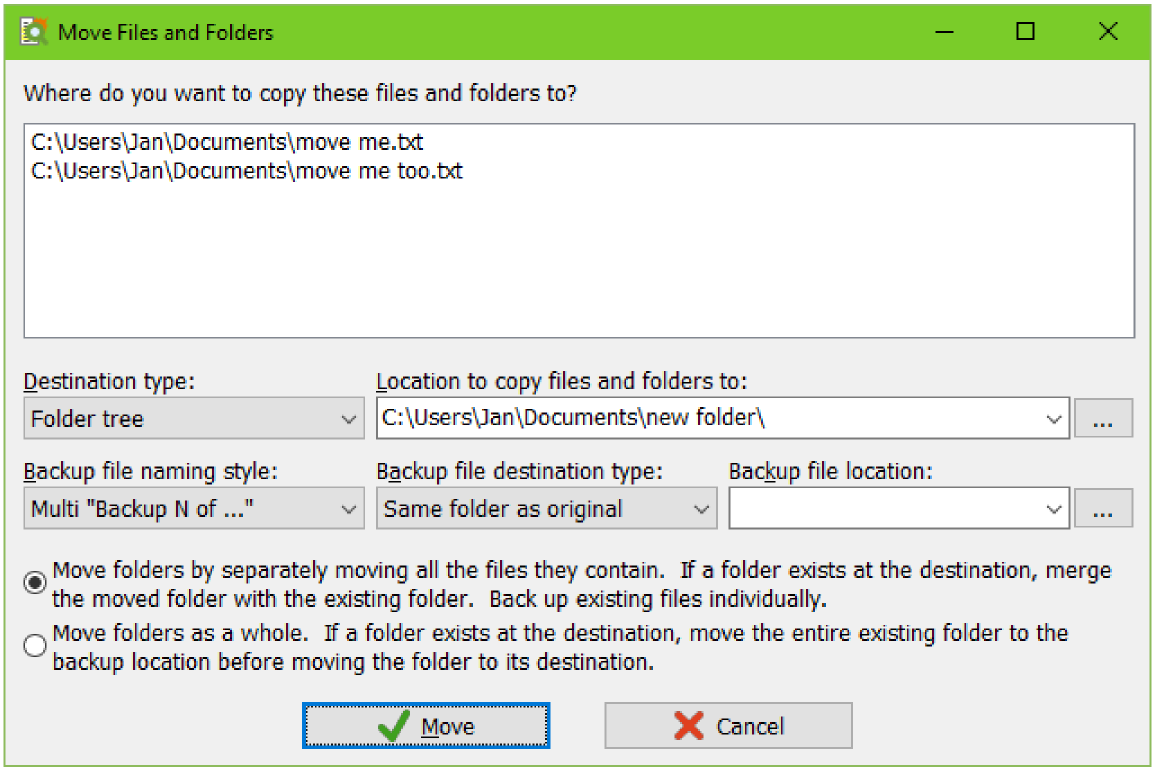
The Copy Files submenu of the File Selector menu allows you to copy four sets of files:

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



Move Files and Folders

Where do you want to copy these files and folders to?

C:\Users\Jan\Documents\move me.txt
C:\Users\Jan\Documents\move me too.txt

Destination type: Folder tree

Location to copy files and folders to: C:\Users\Jan\Documents\new folder\

Backup file naming style: Multi "Backup N of ..."

Backup file destination type: Same folder as original

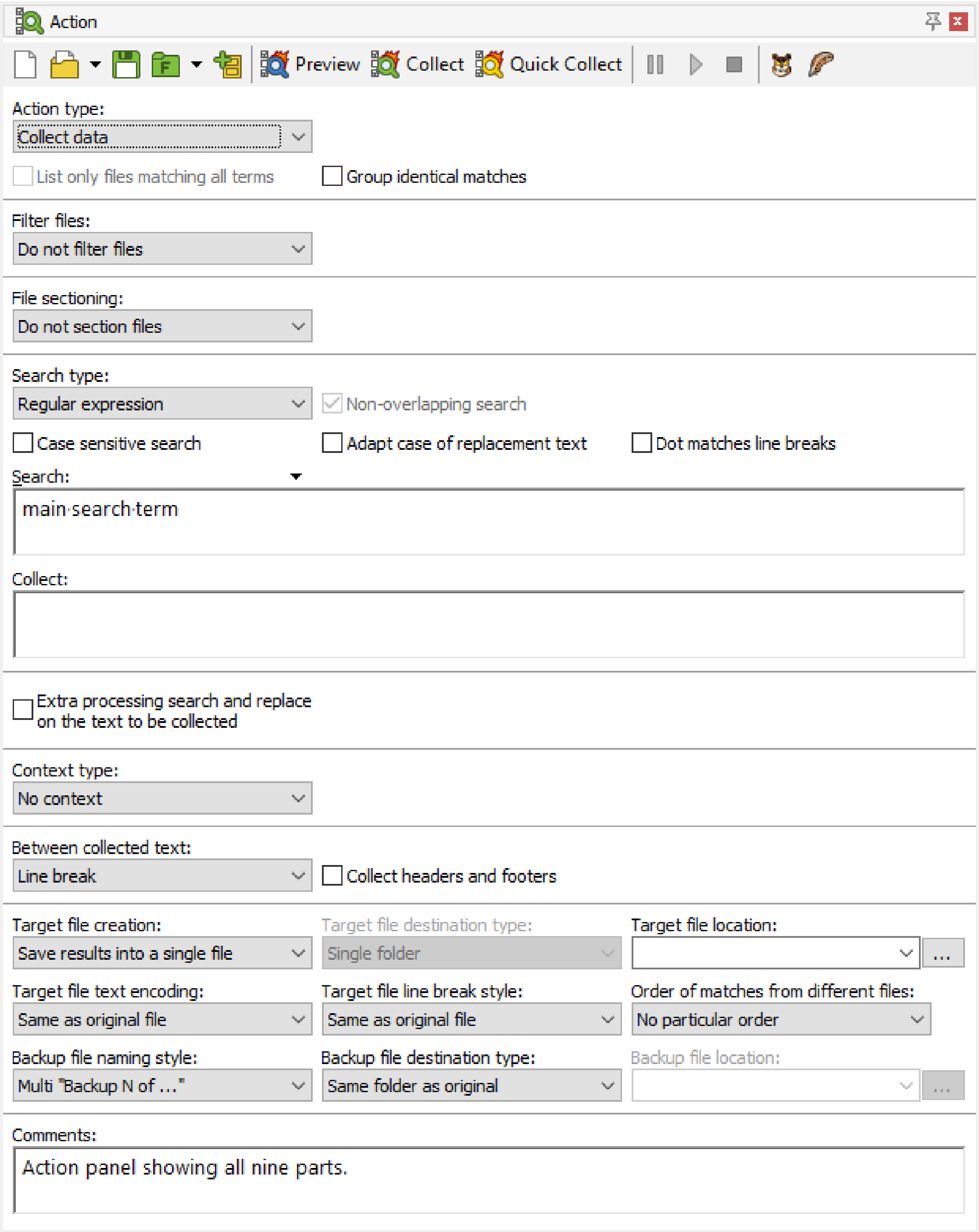
Backup file location:

● Move folders by separately moving all the files they contain. If a folder exists at the destination, merge the moved folder with the existing folder. Back up existing files individually.

○ Move folders as a whole. If a folder exists at the destination, move the entire existing folder to the backup location before moving the folder to its destination.

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.

9. Action Reference



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 line breaks

Search:

```
main·search·term
```

Collect:

☐ Extra processing search and replace on the text to be collected

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 matches from different files: 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.

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”, “count matches”, “list files”, “file or folder name search”, “file or folder name collect”, “rename files or folders”, “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. The “list files” and “rename files and folders” action types, for example, 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 or folders”.
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”, “file or folder name collect”, 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 “collect data”, “file or folder name collect”, “search and replace”, “rename files or folders”, and 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. Only used for actions types that display search matches found in the contents of files.
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”, “file or folder name collect”, 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. Selecting the “delete matching files” target type in combination with the “list files” action type, for example, 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 never modifies any files, or do anything else you might regret.

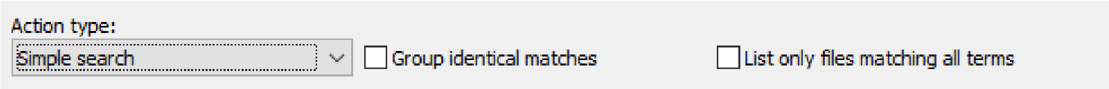
The buttons on the Action toolbar that correspond with the Execute and Quick Execute menu items change their labels to indicate exactly what will happen. The labels change whenever you change the action type or target type. When the action type is “rename files”, for example, the word “Execute” changes into “Rename” or “Copy” depending on whether the target type is “rename files” or “copy files”.

10. 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 twelve action types:

1. Simple search
2. Search
3. Collect data
4. Count matches
5. List files
6. File or folder name search
7. File or folder name collect
8. Rename files or folders
9. Search and replace
10. Search and delete
11. Merge files
12. Split files

Simple Search


Action type: Simple search

□ Group identical matches

□ List only files matching all terms

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”.

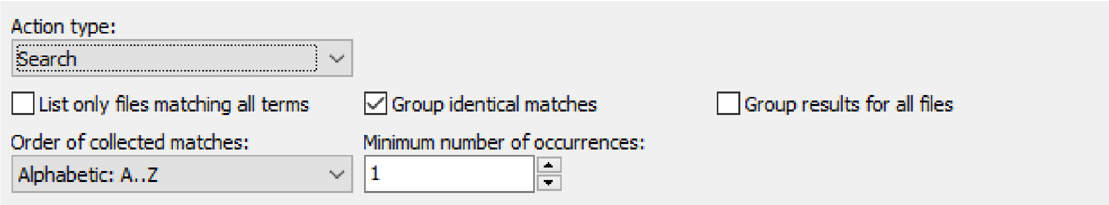
Group Identical Matches

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

Order of collected matches: Alphabetic: A..Z

Minimum number of occurrences: 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.

Order of Collected Matches

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

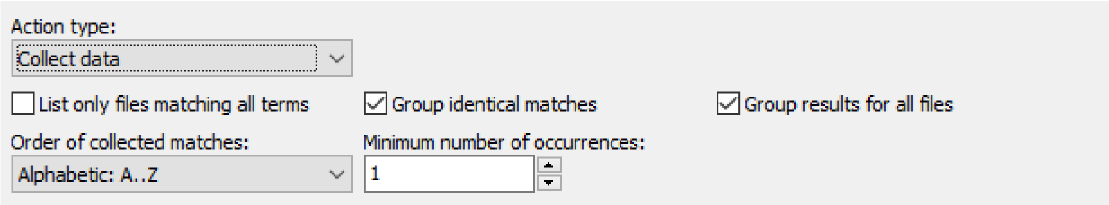
- **Alphabetic: A..Z**: Sort matches alphabetically, from A to Z.
- **Alphabetic: Z..A**: Sort matches alphabetically, from Z to A.
- **Alphanumeric: A..Z, 0..9**: Sort matches alphanumerically, from A to Z and from 0 to 9.
- **Alphanumeric: Z..A, 9..0**: Sort matches alphanumerically, from Z to A and from 9 to 0.
- **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

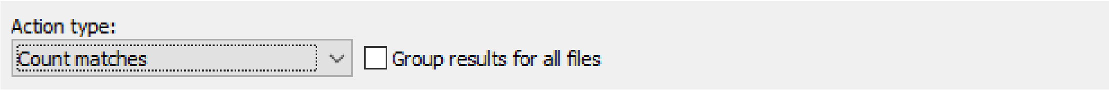

Action type:
Collect data
☐ List only files matching all terms ☑ Group identical matches ☑ Group results for all files
Order of collected matches:
Alphabetic: A..Z
Minimum number of occurrences:
1

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.

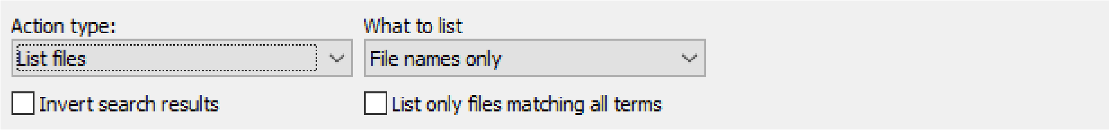
Examples: Find email addresses, Collect page numbers, Collect XML Data with entities replaced, Inspect web logs and Extract Google search terms from web logs

Count Maches


Action type:
Count matches
☐ Group results for all files

The “count matches” action type runs a search just like the “search” action type. But instead of displaying the search matches that were found in the files as the results, “count matches” displays the search terms that you entered in the main part of the action along with the number of times that a match was found for that search term. Search terms that have no matches are also listed in the results. When searching for regular expressions, all the matches for each regex are counted towards that regex, even when the matches are not the same.

List Files


Action type:
List files
What to list
File names only
☐ Invert search results ☐ List only files matching all terms

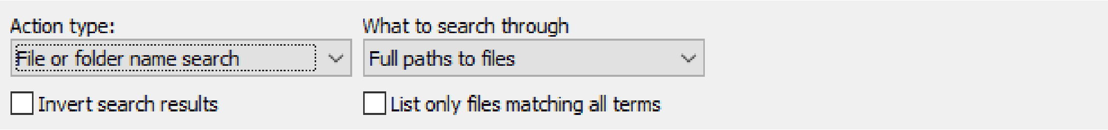
The “list files” action type does not require a search term in the main part of the action. If you don’t provide a search term, PowerGREP simply lists all files that you included in the File Selector. If you do provide a search term, PowerGREP searches through the contents of the files to find it. Files 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 or Folder Name Search



Action type: File or folder name search

What to search through: Full paths to files

□ Invert search results

□ List only files matching all terms

Set the action type to “file or folder 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 names of files, only the names of folders, or the names of both files and folders are searched through. That setting also 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 though” setting also determines the part of the file’s path that is saved into the target file.

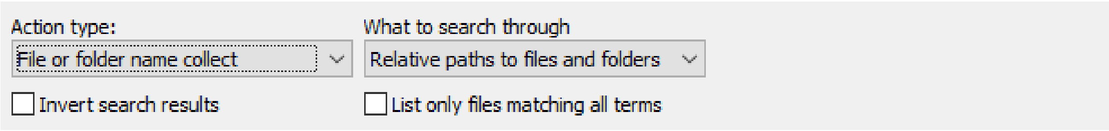
If you turn on “invert results”, then the results show the file or folder paths in which none of the search terms can be found. If you turn on “list only files matching all terms”, you get the file or folder paths in which all the search terms can be found. If you turn on both these options, you get a list of all paths in which none of the search terms 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.

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

File or Folder Name Collect



Action type: File or folder name collect

What to search through: Relative paths to files and folders

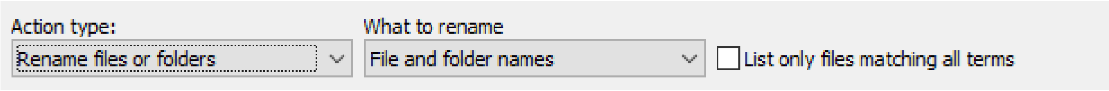
□ Invert search results

□ List only files matching all terms

The “file or folder name collect” action type corresponds with the “file or folder name search” action type just like the “collect data” action type corresponds with the “search” action type. It searches through the names of files and folders but instead of collecting the search matches you can use the replacement string syntax to specify a different text to be collected, which can be further manipulated with “extra processing”. Path placeholders are also very useful with this action type.

Example: Process files in a batch file or script

Rename Files or Folders



Action type: Rename files or folders

What to rename: File and folder names

□ List only files matching all terms

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.

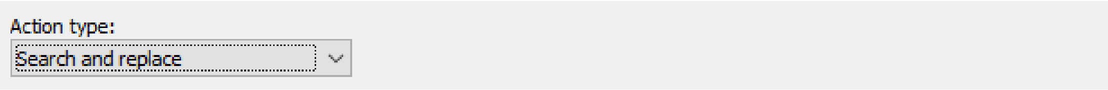
The "what to rename" setting also includes choices to rename folders. If a folder is renamed or moved and files or folders inside that folder are also included in the action, then those files and folders are still processed and can thus be renamed or moved on their own after their parent folder is renamed or moved. So if you rename A to B in folder names only (not folder paths), then the folder C:\A is renamed into C:\B. Its subfolder that was originally C:\A\A is renamed into C:\B\B.

If you set the target type to copy folders then files and folders inside copied folders are copied along with the parent folder. But the names of files and folders inside copied folders are not searched through even if they were marked as included in the File Selector. The action never makes two copies of the same file or folder and never makes copies of copies. So if you copy folders by replacing A with B in folder names only (not folder paths), then the folder C:\A is copied to C:\B. Its subfolder that was originally C:\A\A is copied along as C:\B\A. There will be no additional copy C:\A\B nor will their be a copy of a copy as C:\B\B.

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



Action type:

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.

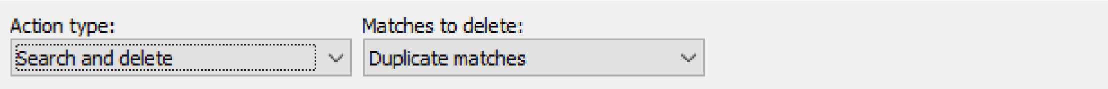
But PowerGREP provides a lot more options on the Action panel. The "filter files" settings allow you to restrict the search-and-replace to files in which a separate set of search terms can or cannot be matched. With the "file sectioning" options you can split the files into lines or other sections and 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



Action type: Search and delete

Matches to delete: 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.

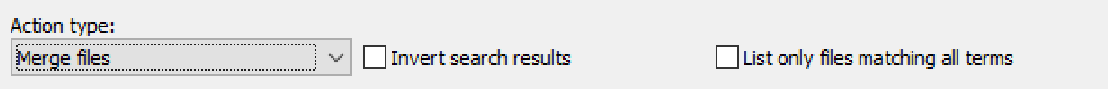
Matches to Delete

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



Action type: Merge files

□ Invert search results

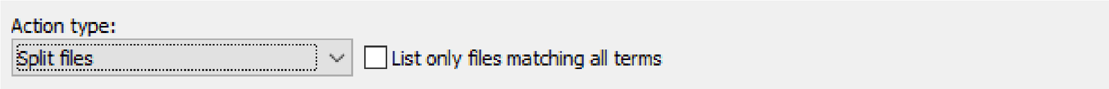
□ List only files matching all terms

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


Action type:
Split files
□ List only files matching all terms

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.

Examples: Split web logs by date, Split logs into files with a certain number of entries and Split database dumps

11. 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. If you set “file sectioning” to “line by line”, for example, 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.

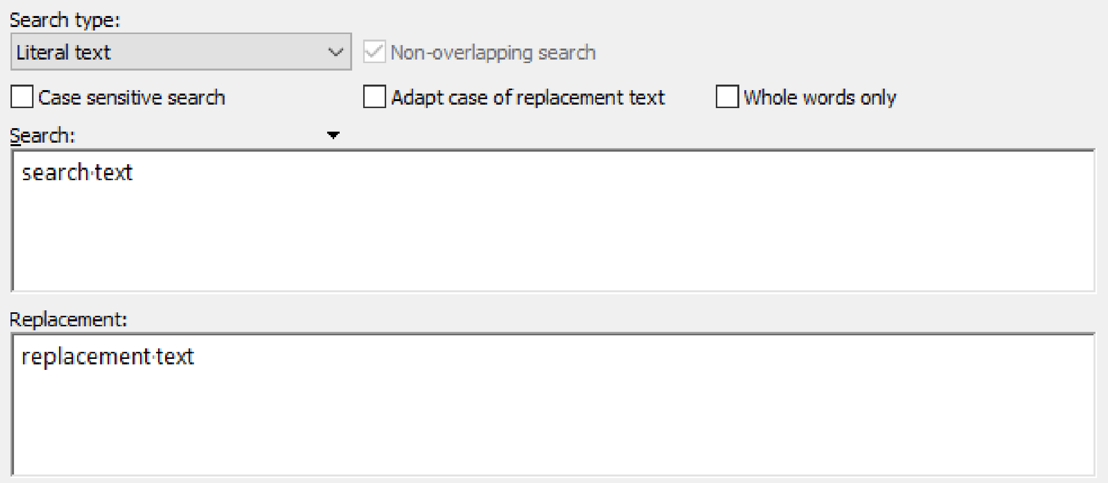
You probably also want to have a look at RegexBuddy and RegexMagic. Both products are available separately. RegexBuddy makes it much easier to work with the regular expression syntax to create and edit regular expressions for use with PowerGREP and a variety of other tools and programming languages. 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.



Search type: Literal text

☑ Non-overlapping search

□ Case sensitive search

□ Adapt case of replacement text

□ Whole words only

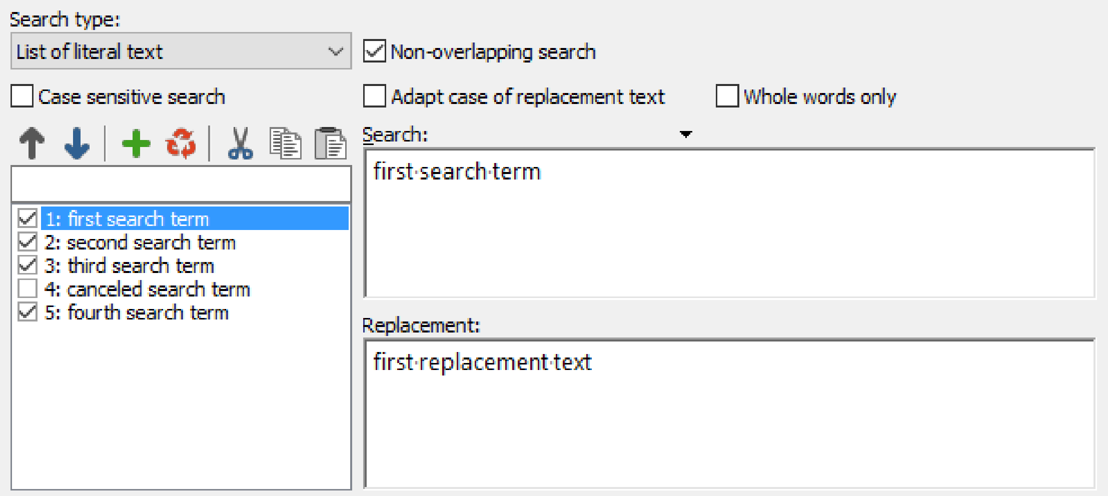
Search:

```
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).



Search type: List of literal text

☑ Non-overlapping search

□ Case sensitive search

□ Adapt case of replacement text

□ Whole words only

- ☑ 1: first search term
- ☑ 2: second search term
- ☑ 3: third search term
- □ 4: canceled search term
- ☑ 5: fourth search term

Search:

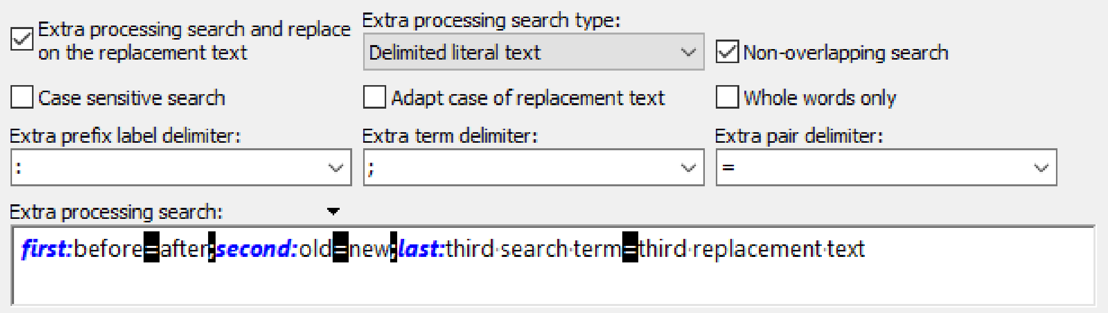
```
first·search·term
```

Replacement:

```
first·replacement·text
```

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"



☑ Extra processing search and replace on the replacement text

Extra processing search type: Delimited literal text

☑ 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 Filein 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

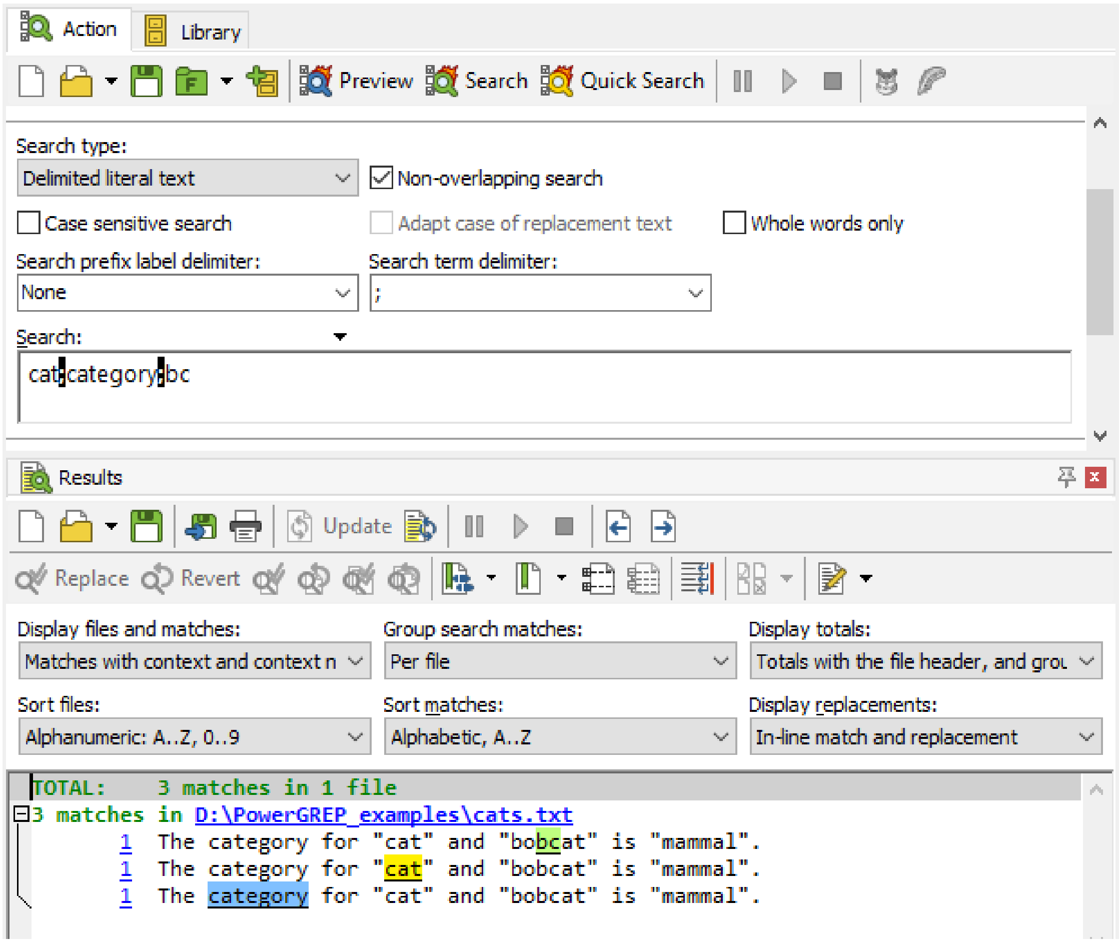
Several action parts such as "extra processing" or the main part of the action expect a replacement text or a text to be collected for each search term. When the search type is a regular expression, you can use backreferences to capturing groups to reinsert part of the regex match into the replacement 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.



Search type: Delimited literal text

☑ Non-overlapping search

□ Case sensitive search

□ Adapt case of replacement text

□ Whole words only

Search prefix label delimiter: None

Search term delimiter: ;

Search: cat;category;bc

Display files and matches: Matches with context and context n

Group search matches: Per file

Display totals: Totals with the file header, and grou

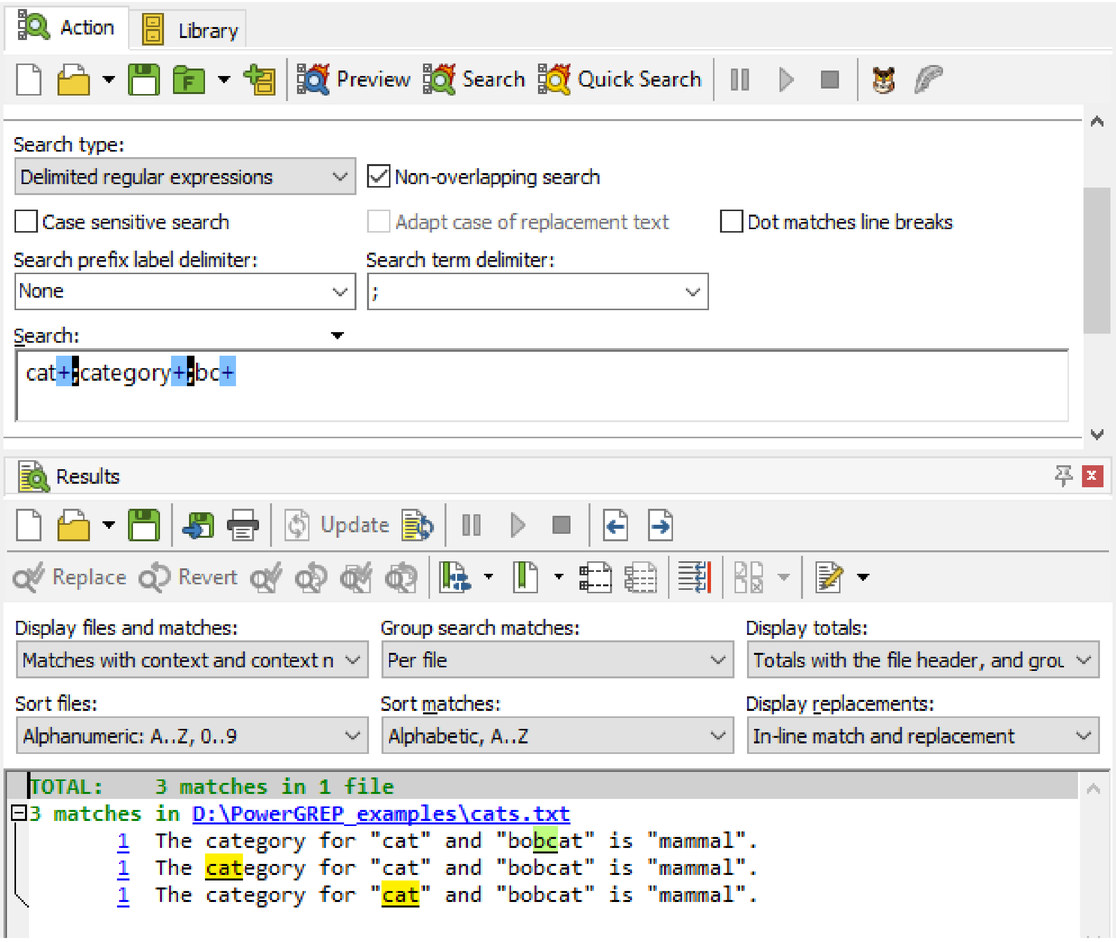
Sort files: Alphanumeric: A..Z, 0..9

Sort matches: Alphabetic, A..Z

Display replacements: In-line match and replacement

```
TOTAL:    3 matches in 1 file
⊟3 matches in D:\PowerGREP_examples\cats.txt
        1  The category for "cat" and "bobcat" is "mammal".
        1  The category for "cat" and "bobcat" is "mammal".
        1  The category for "cat" and "bobcat" is "mammal".
```

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.



Search type: Delimited regular expressions

☑ Non-overlapping search

□ Case sensitive search

□ Adapt case of replacement text

□ Dot matches line breaks

Search prefix label delimiter: None

Search term delimiter: ;

Search:

```
cat+;category+;bc+
```

Display files and matches: Matches with context and context n

Group search matches: Per file

Display totals: Totals with the file header, and grou

Sort files: Alphanumeric: A..Z, 0..9

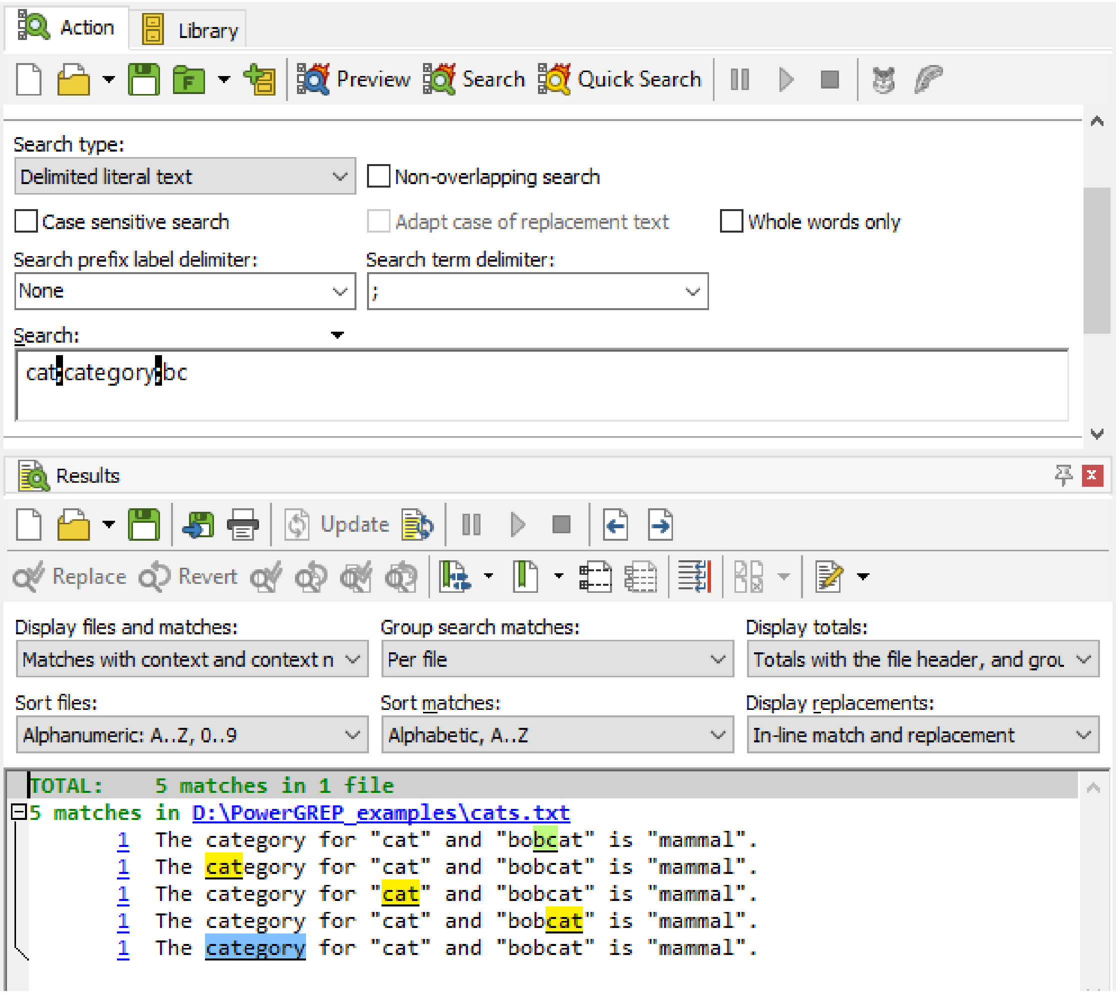
Sort matches: Alphabetic, A..Z

Display replacements: In-line match and replacement

```
TOTAL:    3 matches in 1 file
3 matches in D:\PowerGREP examples\cats.txt
        1  The category for "cat" and "bobcat" is "mammal".
        1  The category for "cat" and "bobcat" is "mammal".
        1  The category for "cat" and "bobcat" is "mammal".
```

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.



Search type: Delimited literal text

☐ Non-overlapping search

☐ Case sensitive search

☐ Adapt case of replacement text

☐ Whole words only

Search prefix label delimiter: None

Search term delimiter: ;

Search:

```
cat;category;bc
```

Display files and matches: Matches with context and context n

Group search matches: Per file

Display totals: Totals with the file header, and grou

Sort files: Alphanumeric: A..Z, 0..9

Sort matches: Alphabetic, A..Z

Display replacements: In-line match and replacement

```
TOTAL:    5 matches in 1 file
⊟5 matches in D:\PowerGREP examples\cats.txt
        1  The category for "cat" and "bobcat" is "mammal".
        1  The category for "cat" and "bobcat" is "mammal".
        1  The category for "cat" and "bobcat" is "mammal".
        1  The category for "cat" and "bobcat" is "mammal".
        1  The category for "cat" and "bobcat" is "mammal".
```

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`.

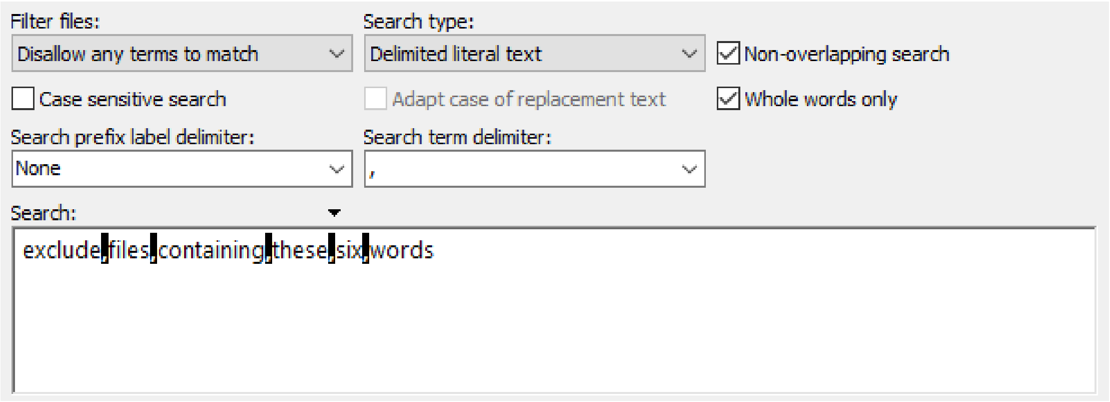
What "whole words only" really does is check if the match is not immediately preceded and not immediately followed by a character that could be part of a word. `cat` fails `category` because of the `e` immediately after the potential match. Note that your search term must be a word or phrase. If your 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.

12. Action Part: Filter Files



Filter files: Disallow any terms to match

Search type: Delimited literal text

☑ Non-overlapping search

□ Case sensitive search

□ Adapt case of replacement text

☑ Whole words only

Search prefix label delimiter: None

Search term delimiter: ,

Search:

```
exclude,files,containing,these,six,words
```

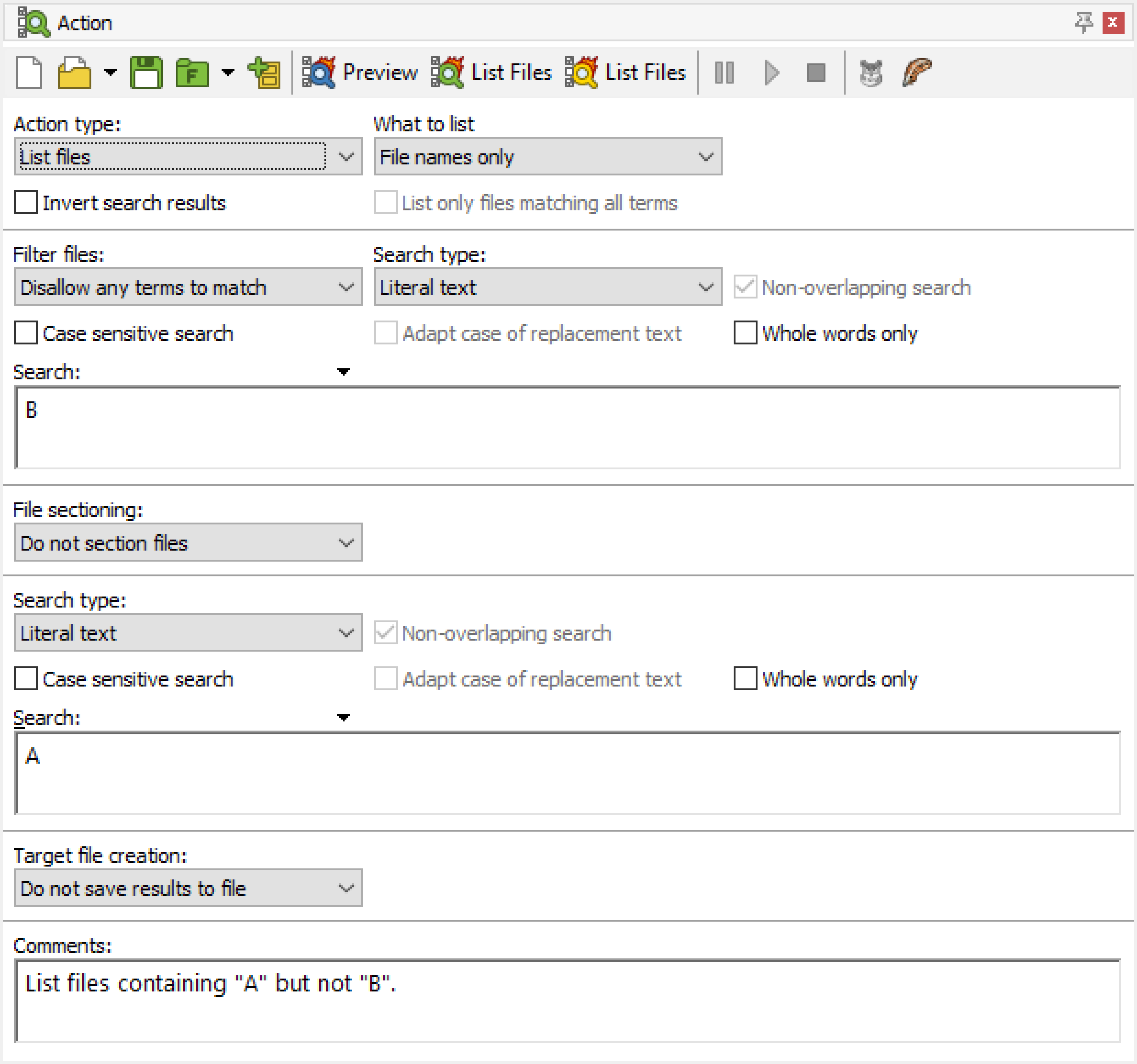
The "filter files“ action part is available for all action types except ”simple search“. It enables you to use an extra set of search terms to filter out files. The ”filter files" drop-down list gives you four choices:

- **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.

Filtering files is always done by searching through the contents of files. Even for "file or folder name search", "file or folder name collect", and "rename files or folders" actions where the main part of the action searches through the file's name. If you want to filter out files based on their file names, use the "exclude files" box on the File Selector panel.

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



Action type: List files
What to list: File names only
☐ Invert search results
☐ List only files matching all terms

Filter files: Disallow any terms to match
Search type: Literal text
☑ Non-overlapping search
☐ Case sensitive search
☐ Adapt case of replacement text
☐ Whole words only
Search:
B

File sectioning: Do not section files

Search type: Literal text
☑ Non-overlapping search
☐ Case sensitive search
☐ Adapt case of replacement text
☐ Whole words only
Search:
A

Target file creation: Do not save results to file

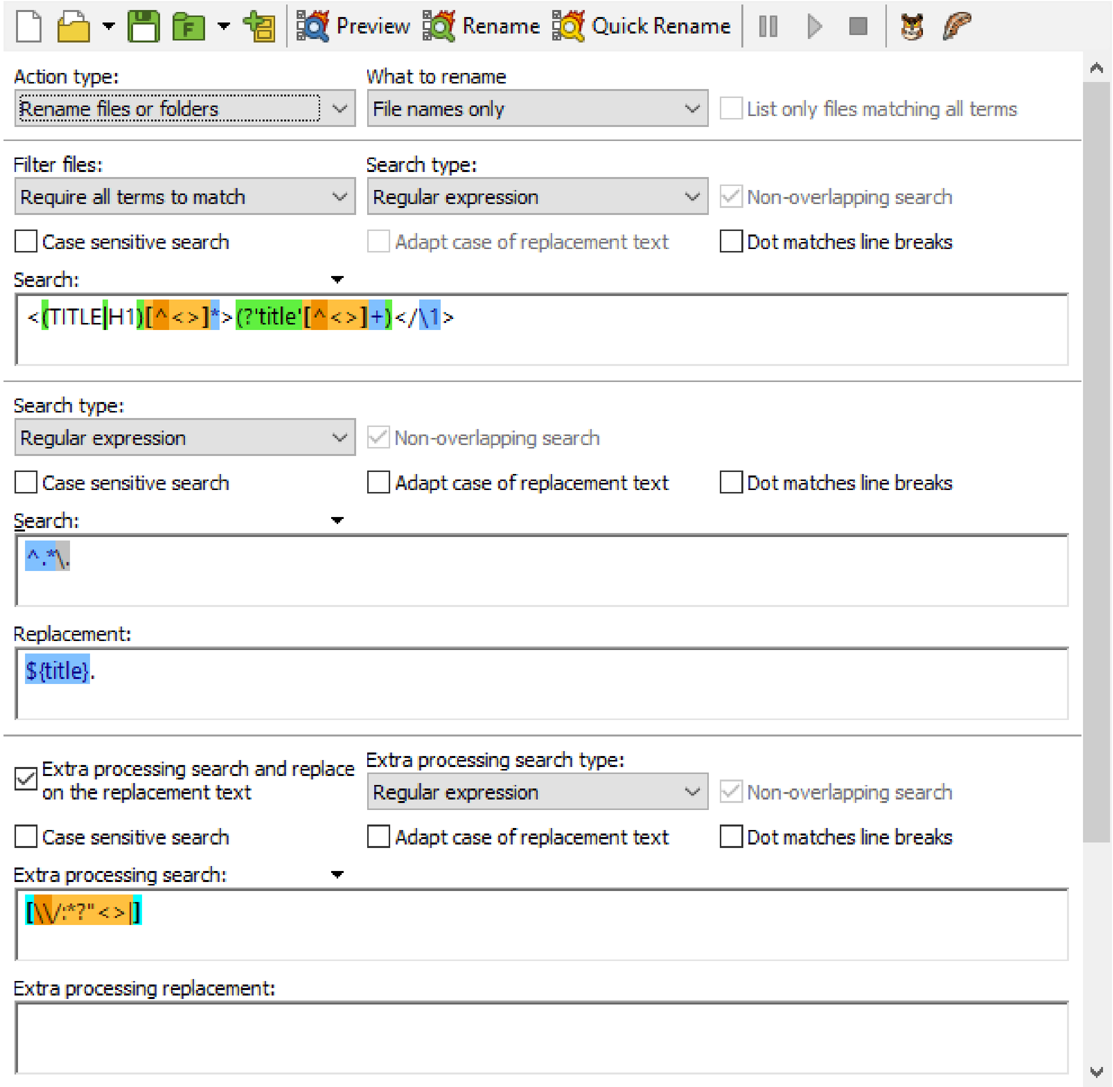
Comments:
List files containing "A" but not "B".

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



Action type: Rename files or folders

What to rename: File names only

□ List only files matching all terms

Filter files: Require all terms to match

Search type: Regular expression

☑ Non-overlapping search

□ Case sensitive search

□ Adapt case of replacement text

□ Dot matches line breaks

Search:

```
<(TITLE|H1)[^<>]*>(?'title'[^<>]+)</\1>
```

Search type: Regular expression

☑ Non-overlapping search

□ Case sensitive search

□ Adapt case of replacement text

□ Dot matches line breaks

Search:

```
^.*\.
```

Replacement:

```
${title}.
```

☑ Extra processing search and replace on the replacement text

Extra processing search type: Regular expression

☑ Non-overlapping search

□ Case sensitive search

□ Adapt case of replacement text

□ Dot matches line breaks

Extra processing search:

```
[\\/:*?"<>|]
```

Extra processing replacement:

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 then search for that part elsewhere in the file by using a backreference to that named capturing group in the regular expression in the main part of the action. Or you can insert that part somewhere by using a backreference in the replacement string in the main part of the action.

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.

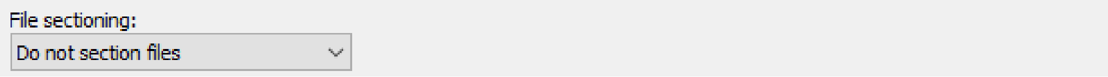
Named capturing groups also carry over when "filter files" works on the contents of files (which it always does) and the main part of the action works on file names (which it does for "file or folder name search", "file or folder name collect", and "rename files or folders" actions). This allows you to search through the file's name for something found in the file's contents, or collect part of the file's name along with part of the file's contents, or change the file's name to something found in its contents. The screen shot shows an example of using this technique to rename HTML files to their titles or top-level headers.

Examples: Rename Audio Files Using Meta Data, Insert proper HTML title tags and Rename files based on HTML title tags

13. 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 or folders”.

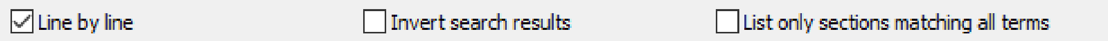
Disabling File Sectioning



File sectioning:
Do not section files

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)



☑ Line by line ☐ Invert search results ☐ List only sections matching all terms

The “simple search” action type offers only two file sectioning options. Instead of the “file sectioning” drop-down 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.

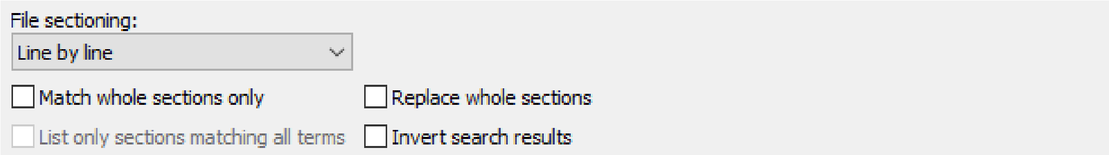
When you tick the “line by line” checkbox, one or two additional checkboxes appear. The option "invert search results"makes the main part of the action match sections (lines) in which the search terms can *not* be found. The entire section (line) is then treated as the search match. That means that when 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)



File sectioning:
Line by line

□ Match whole sections only
□ Replace whole sections
□ List only sections matching all terms
□ Invert search results

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. When sectioning a file line by line, for example, only search matches spanning a complete line are retained.

“Collect whole sections”, “replace whole sections” , “delete whole sections” , or “split 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. The label of this option changes with the action type, but the core behavior remains the same. Actions that collect data will collect the whole section, search-and-replace actions replace the whole section, search-and-delete actions delete the whole section, and split actions write the whole section to the target file.

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

Invert Search Results

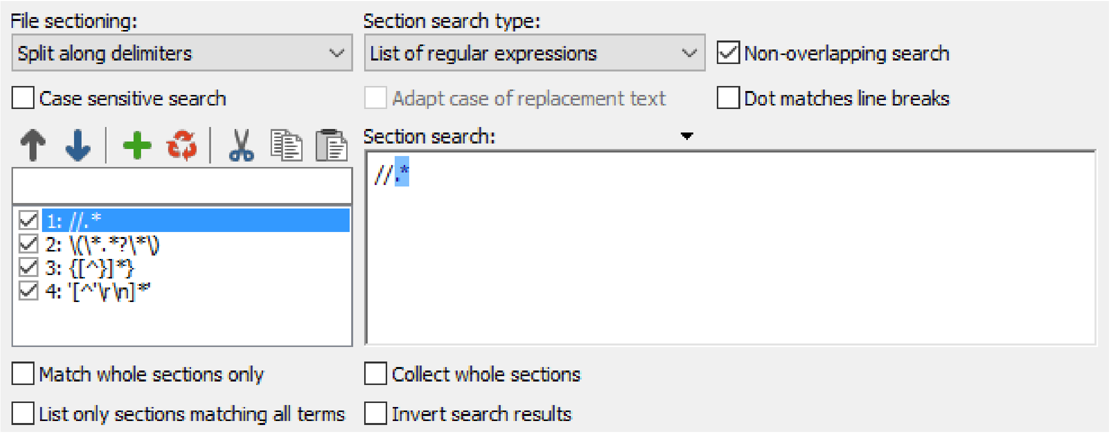
Turn on to treat sections in which the search term cannot be found as search matches and to treat sections in which the search term can be found as not having matched.

If you turn on both “invert search results” and “match whole sections only” then sections which are matched only partially or not at all by the search term are treated as search matches.

If you turn on both “invert search results” and “list only sections matching all terms” then sections in which all terms can be found are treated as not having matched while sections in which none or some (but not all) search terms can be found are treated as having matched.

Turning on “invert search results” automatically turns on "collect/replace/delete/split whole sections" because when sections without search terms have to be treated as matches, the only way to do that is to treat the whole section as the match.

Other Ways to Split Files into Sections



File sectioning: Split along delimiters

Section search type: List of regular expressions

☑ Non-overlapping search

☐ Case sensitive search

☐ Adapt case of replacement text

☐ Dot matches line breaks

Section search:

```
//.*
```

- ☑ 1: //.*
- ☑ 2: \(*.*?*\)
- ☑ 3: {[^}]*}
- ☑ 4: '[^'\r\n]*'

☐ Match whole sections only

☐ Collect whole sections

☐ List only sections matching all terms

☐ Invert search results

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 lookaround, 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 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 newlines”. 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.

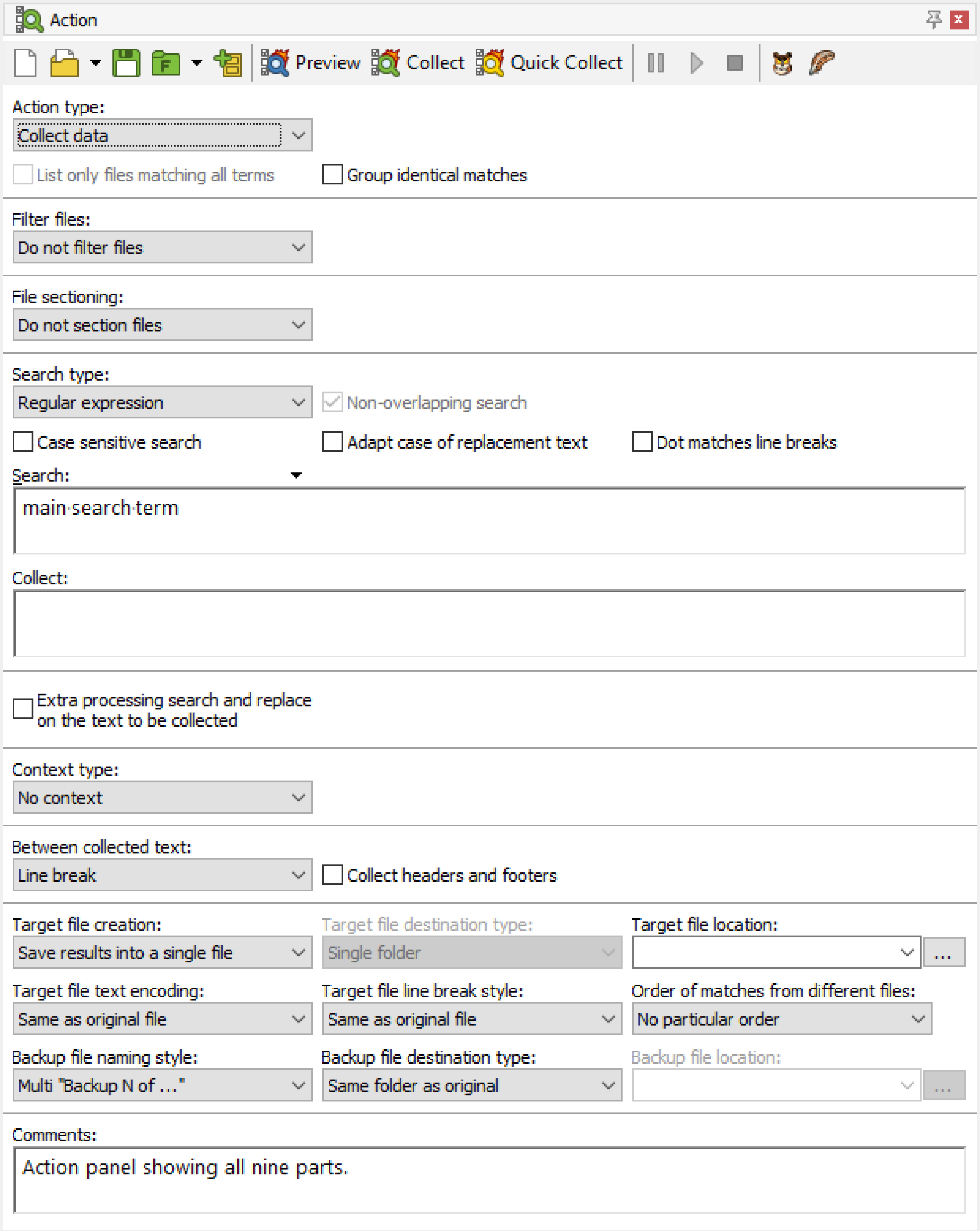
Named Capturing Groups Carry Over

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.

Examples: Collect a list of header and item pairs and Make sections and their contents consistent

14. 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:



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 line breaks

Search:

```
main·search·term
```

Collect:

☐ Extra processing search and replace on the text to be collected

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 matches from different files: 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.

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.

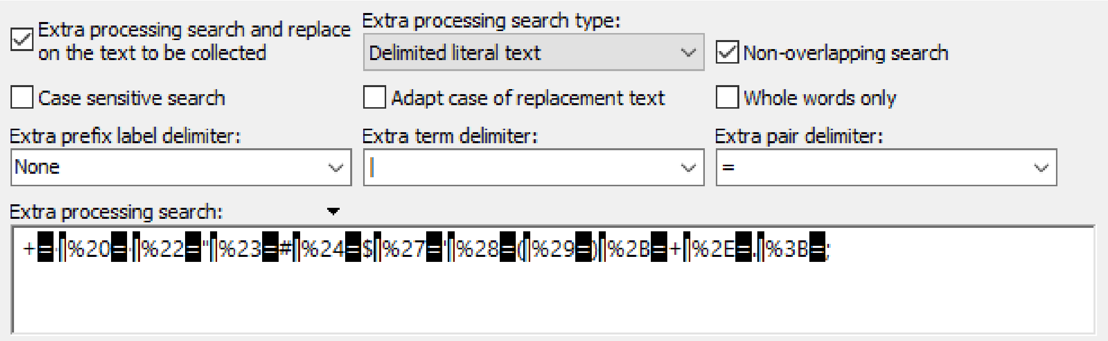
What PowerGREP does with the main search term of your action depends on the “action type” that you’ve selected at the top of the Action panel. All action types except “list files”, “file or folder name collect” and “merge files” need at least one search term in the main part of the action.

The action type also determines whether 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” and “rename files or folders” action types need a replacement text to replace search matches with. The “collect data” and “file or folder name collect” action types need 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 or folder name search”, file or folder name collect“, and ”rename files or folders" action types use them to search through the names of files instead.

The search term(s) can be literal text, binary data, or a regular expression. You choose this through the “search type” setting. If the search term is a regular expression, the replacement text or text to be collected uses the replacement string syntax with backreferences from the regular expression. Otherwise, the replacement text or text to be collected is also literal text or binary data.

15. Action Part: Extra Processing



☑ Extra processing search and replace on the text to be collected

Extra processing search type: Delimited literal text

☑ Non-overlapping search

□ Case sensitive search

□ Adapt case of replacement text

□ Whole words only

Extra prefix label delimiter: None

Extra term delimiter: |

Extra pair delimiter: =

Extra processing search:

```
+=·|%20=·|%22="|%23=#|%24=$|%27='|%28=(|%29=)|%2B=+|%2E=.|%3B=;
```

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 folders, 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: Padding replacements, Rename Audio Files Using Meta Data, 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.

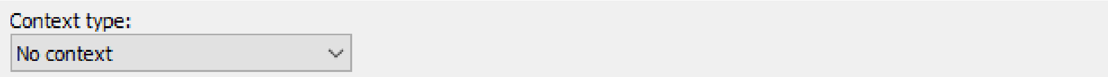
16. Action Part: Context

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 or folders" 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



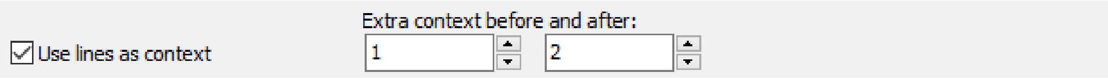
Context type:

No 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)



☑ Use lines as context

Extra context before and after: 1 2

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.

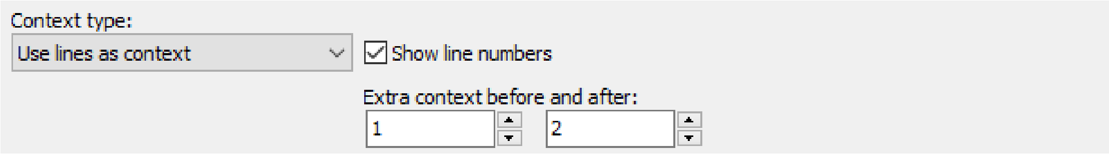
If expand to whole lines is checked, extra lines of context are added. Otherwise, extra blocks according to the context type are added.

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)



Context type: Use lines as context

☑ Show line numbers

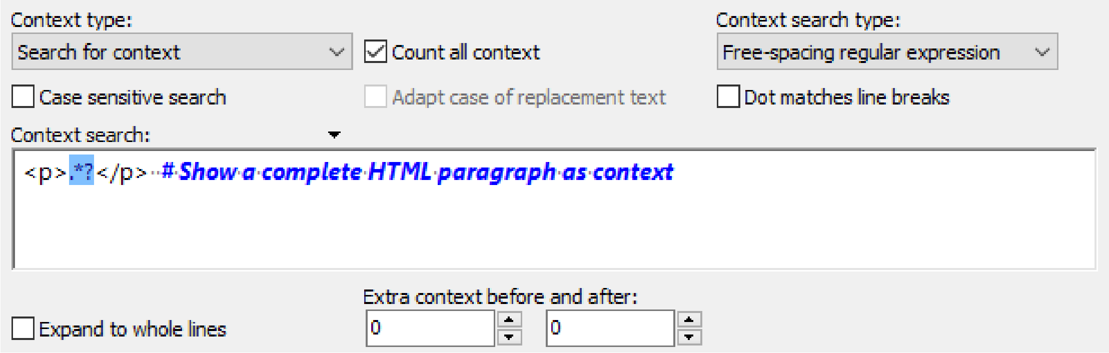
Extra context before and after: 1 2

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



Context type: Search for context

☑ Count all context

Context search type: Free-spacing regular expression

☐ Case sensitive search

☐ Adapt case of replacement text

☐ Dot matches line breaks

Context search:

```
<p>.*?</p>  #Show a complete HTML paragraph as context
```

☐ Expand to whole lines

Extra context before and after: 0 0

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

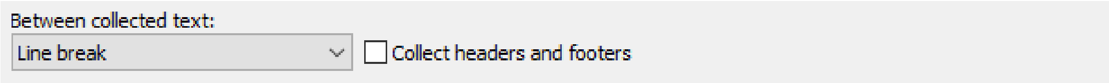
Turn on to expand the context so that all matches are always shown with full lines of context. This makes it easier to interpret the results if you're used to working with the files with a line-based application such as a plain text editor. Turn off to allow the context to be a partial line.

17. 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.

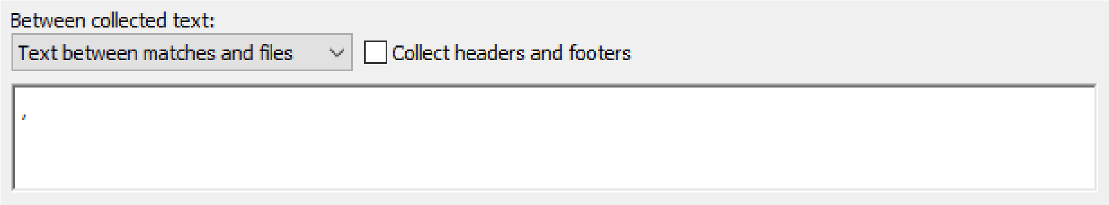
The appearance of the “collect between” part of the action changes based on what you select in the “between collected text” drop-down list and whether you tick “collect headers and footers”. If you’re not using any options that require custom text, no text box appears:



Between collected text: Line break

□ Collect headers and footers

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

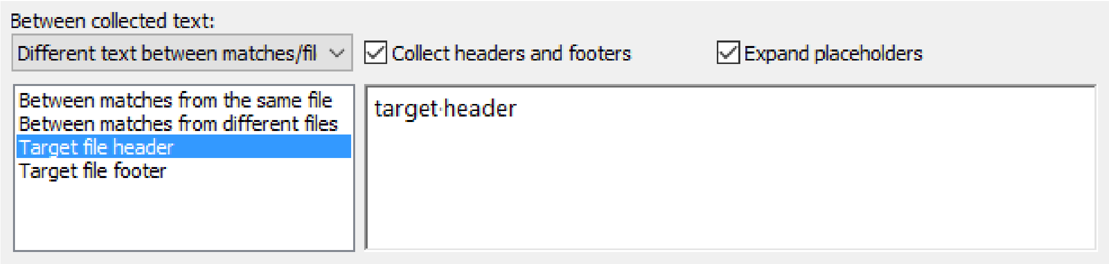


Between collected text: Text between matches and files

□ Collect headers and footers

```
,
```

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:



Between collected text: Different text between matches/fil

☑ Collect headers and footers

☑ Expand placeholders

- Between matches from the same file
- Between matches from different files
- Target file header
- Target file footer

```
target header
```

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.

Expand Placeholders

Expand path placeholders as well as action-related and file-related match placeholders in file headers and footers. In source file headers and footers, path placeholders represent the path to the file that the matches were found in. In target file headers and footers, they represent the file that the collected text is written to.

The match placeholder %MATCHCOUNT% can be used in source file footers to indicate the number of matches found in each source file. %MATCHCOUNT% can be used in target file footers to indicate the number of matches written to each target file. %FILECOUNT% can be used in target file footers to indicate the number of source files that had their matches written to that target file.

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".

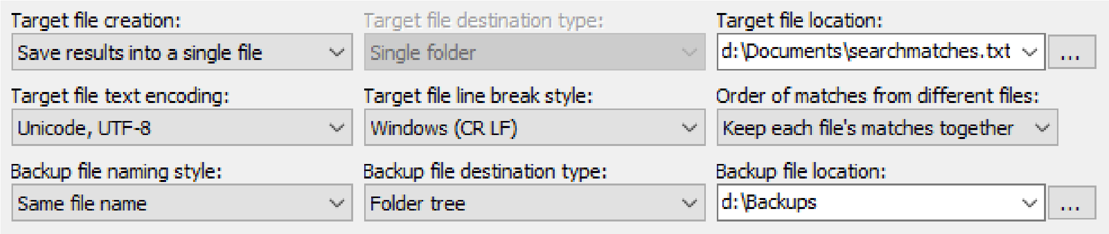
- **Between matches**: Text to be put between collected matches, whether they're from the same file or from different source files.
- **Between matches from the same file**: Text to be put between matches collected from a single source file.
- **Between matches from different files**: Text to be put 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

18. 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 never modifies any files or do anything else you might regret.



Target file creation: Save results into a single file

Target file destination type: Single folder

Target file location: d:\Documents\searchmatches.txt

Target file text encoding: Unicode, UTF-8

Target file line break style: Windows (CR LF)

Order of matches from different files: Keep each file's matches together

Backup file naming style: Same file name

Backup file destination type: Folder tree

Backup file location: d:\Backups

Target Files

Target File Creation

The available target types depend on the kind of action you have prepared. Seven target types are available for “search”, “collect data”, and “file or folder name collect” actions:

- **Do not save results to file**: Display the results on the Results panel in PowerGREP only.
- **Save results into a single file**: Save the text collected from all the files into a single new file.
- **Copy results to the clipboard**: Copy the text collected from all the files to the clipboard.
- **Store results in the editor**: Store the text collected from all the files in an unsaved file on the Editor panel.
- **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.

For a “list files” or file or folder name search action, nine target types are available:

- **Do not save results to file**: Display the results on the Results panel in PowerGREP only.
- **Save list of matched files to file**: Save the list of file names into a single new file.
- **Copy list of matched files to clipboard**: Copy the list of file names to the clipboard.
- **Store list of matched files in the editor**: Store the list of file names in an unsaved file on the Editor panel.
- **Copy matched files**: If a file has one or more search matches, make a copy of it. Copy all included files if there is no seach term. You can use the target file destination type to compress or decompress files, leaving the originals.
- **Move matched files**: If a file has one or more search matches, move it into a different folder. Move all included files if there is no seach term. You can use the target file destination type to compress or decompress files, removing the originals.

- **Convert matched files to text**: Convert files in proprietary file formats that have search matches into plain text files with a .txt extension and using a certain text encoding and/or line break style. Change the text encoding and/or line break styles of plain text files with search matches (overwriting the original file).
- **Convert copies of matched files to text**: Create copies of all files with search matches. Copy the plain text conversion of files in proprietary formats. Change the text encoding and/or line break style of all copies.
- **Delete matched files**: If a file has one or more search matches, permanently delete it.

If you set "what to search through" for a "file and folder name search" action to search through folder names, then you get a different list of nine target types:

- **Do not save results to file**: Display the results on the Results panel in PowerGREP only.
- **Save list of matched folders to file**: Save the list of folder names into a single new file.
- **Copy list of matched folders to clipboard**: Copy the list of folder names to the clipboard.
- **Store list of matched folders in the editor**: Store the list of folder names in an unsaved file on the Editor panel.
- **Copy matched folders**: If a folder's name or path has one or more search matches, make a copy of that folder and all its contents. Copy all included folders if there is no seach term. Copy folders with all their contents, including hidden files and folders. If the target folder already exists, move it to its backup location as a whole. If backups are disabled, delete the existing target folder before copying the matched folder. Do not search through the names of folders inside copied folders to avoid making copies of copies.
- **Move matched folders**: If a folder's name or path has one or more search matches, move that folder and all its contents into another folder. Move all included folders if there is no seach term. Move folders with all their contents, including hidden files and folders. If the target folder already exists, move it to its backup location as a whole. If backups are disabled, delete the existing target folder before moving the matched folder. Also search through the names of folders inside moved folders. If they match, move them on their own after they were moved along with their parent folder.
- **Copy contents of matched folders**: If a folder's name or path has one or more search matches, make a copy of that folder and all its contents. Copy all included folders if there is no seach term. Individually copy the files inside the matched folder, and inside all its subfolders. Skip hidden files and folders. If the target folder already exists, merge the contents of the matched folder with the contents of the target folder. If this causes files to be overwritten, back up those files individually. Do not search through the names of folders inside copied folders to avoid making copies of copies.
- **Move contents of matched folders**: If a folder's name or path has one or more search matches, move that folder and all its contents into another folder. Move all included folders if there is no seach term. Individually move the files inside the matched folder, and inside all its subfolders. If the folder contains hidden files and folders, skip those and leave the original folder behind with those. If the target folder already exists, merge the contents of the matched folder with the contents of the target folder. If this causes files to be overwritten, back up those files individually. Also search through the names of folders inside moved folders. If they match, move them on their own after they were moved along with their parent folder.
- **Delete matched folders**: If a folder's name or path has one or more search matches, delete the folder and all its contents. Delete all included folders if there is no seach term.

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.

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.

When you set the action type to "rename files or folders" you can choose to rename or copy the files or folders to their new names. Because there are different implications to changing a file's name, a file's path, a folder's name, or a folder's path, the available target types depend on the "what to rename" setting at the top of the action panel. When it is set to "file names only", you get these target types:

- **Rename files**: Rename each file to the name created by searching-and-replacing through the file's name.
- **Copy files**: Copy each file to the name created by searching-and-replacing through the file's name.

When renaming relative or full paths to files, the target types change slightly to refect that changes to a file's path can move the file into a new folder:

- **Rename or move files**: Rename or move each file to the path created by searching-and-replacing through the file's path. Folders in the new path are created as needed.
- **Copy files**: Copy each file to the path created by searching-and-replacing through the file's path. Folders in the new path are created as needed.

You can compress or decompress individual files by adding or removing extensions of individual compression formats like .gz or .xz.

When moving or copying folders, there are two ways of dealing with the files in those folders. The folders can be copied or moved entirely, or the contents of those folders can be copied or moved as individual files.

- **Rename or move folders**: Rename or move each folder and its contents to the path created by searching-and-replacing through the folder's path. Renamed and moved folders retain all their contents, including hidden files and folders. If the target folder already exists, move it to its backup location as a whole. If backups are disabled, delete the existing target folder before renaming or moving the matched folder. Also rename or move folders inside renamed folders.
- **Copy folders**: Copy each folder to the path created by searching-and-replacing through the folder's path. Copy folders with all their contents, including hidden files and folders. If the target folder already exists, move it to its backup location as a whole. If backups are disabled, delete the existing target folder before copying the matched folder. Do not search-and-replace through the paths of folders inside copied folders to avoid making copies of copies.
- **Move contents of folders**: Search-and-replace through the paths of folders to determine the target folders to move their contents to. Individually move the files inside the matched folder, and inside all its subfolders. If the folder contains hidden files and folders, skip those and leave the original folder

behind with those. If the target folder already exists, merge the contents of the matched folder with the contents of the target folder. If this causes files to be overwritten, back up those files individually. Also search-and-replace through the paths of folders inside moved folders. Move the contents of those folders on their own after they were moved along with their parent folder.

- **Copy contents of folders**: Search-and-replace through the paths of folders to determine the target folders to move their contents to. Individually copy the files inside the matched folder, and inside all its subfolders. Skip hidden files and folders. If the target folder already exists, merge the contents of the matched folder with the contents of the target folder. If this causes files to be overwritten, back up those files individually. Do not search-and-replace through the paths of folders inside copied folders to avoid making copies of copies.

You can compress folders by adding an extension of an archive formats like .zip or .7z to their name. You can decompress archives by removing those extensions from their names.

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 Matches from Different Files

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 alphanumerically: A..Z, 0..9**: Keep each file's matches together. Create the target file after all files have been processed, collecting text in alphanumeric order of the files.
- **Sort files alphanumerically: Z..A, 9..0**: 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.

19. 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 an the replacement text `\2/\1/\3` to flip the day and month numbers.

Many modern regular expression flavors, included the one used in PowerGREP, also support named capturing groups. 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.

Examples: Insert proper HTML title tags, Rename files based on HTML title tags, Collect a list of header and item pairs and Make sections and their contents consistent

20. 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". Its label depends on the action type and target type you've chosen. The "Search" and "Count" labels are used for actions that do *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" and "Count" labels are used for all actions that do *not* modify any files. All other labels indicate that the action *will* create and/or modify files.

There is no difference between Execute and Quick Execute for the "count matches" and "list files" action types, because such actions never keep individual search matches. For such actions the label of the Quick Execute button on the Action toolbar is the same as that of the Execute button.

For all other actions, Quick Execute can be significantly faster and use far less of your computer's memory than Execute or Preview. 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.

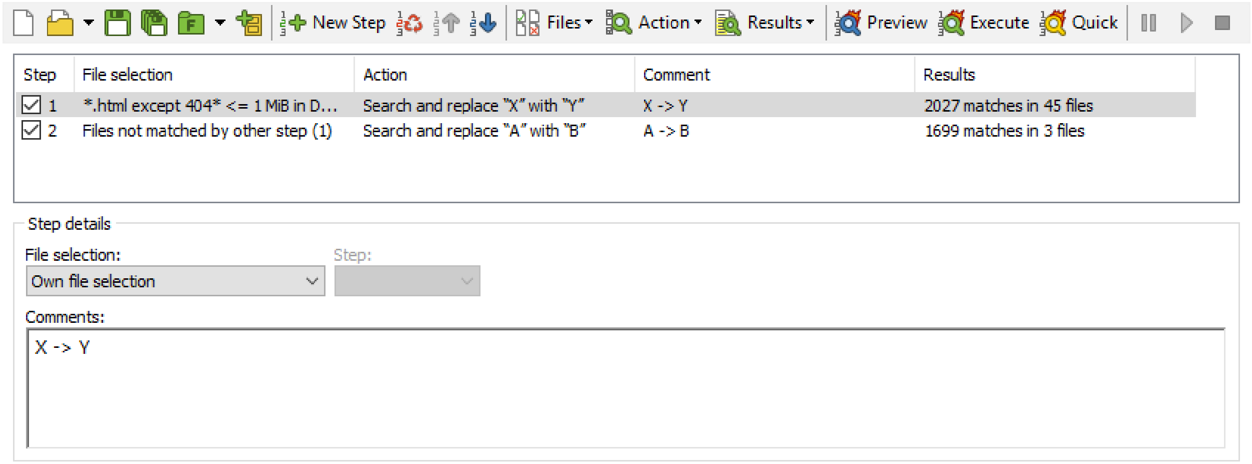
Abort

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.

21. 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.



Step	File selection	Action	Comment	Results
☑ 1	*.html except 404* <= 1 MiB in D...	Search and replace "X" with "Y"	X -> Y	2027 matches in 45 files
☑ 2	Files not matched by other step (1)	Search and replace "A" with "B"	A -> B	1699 matches in 3 files

Step details

File selection: Own file selection

Step:

Comments:

X -> Y

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

22. 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

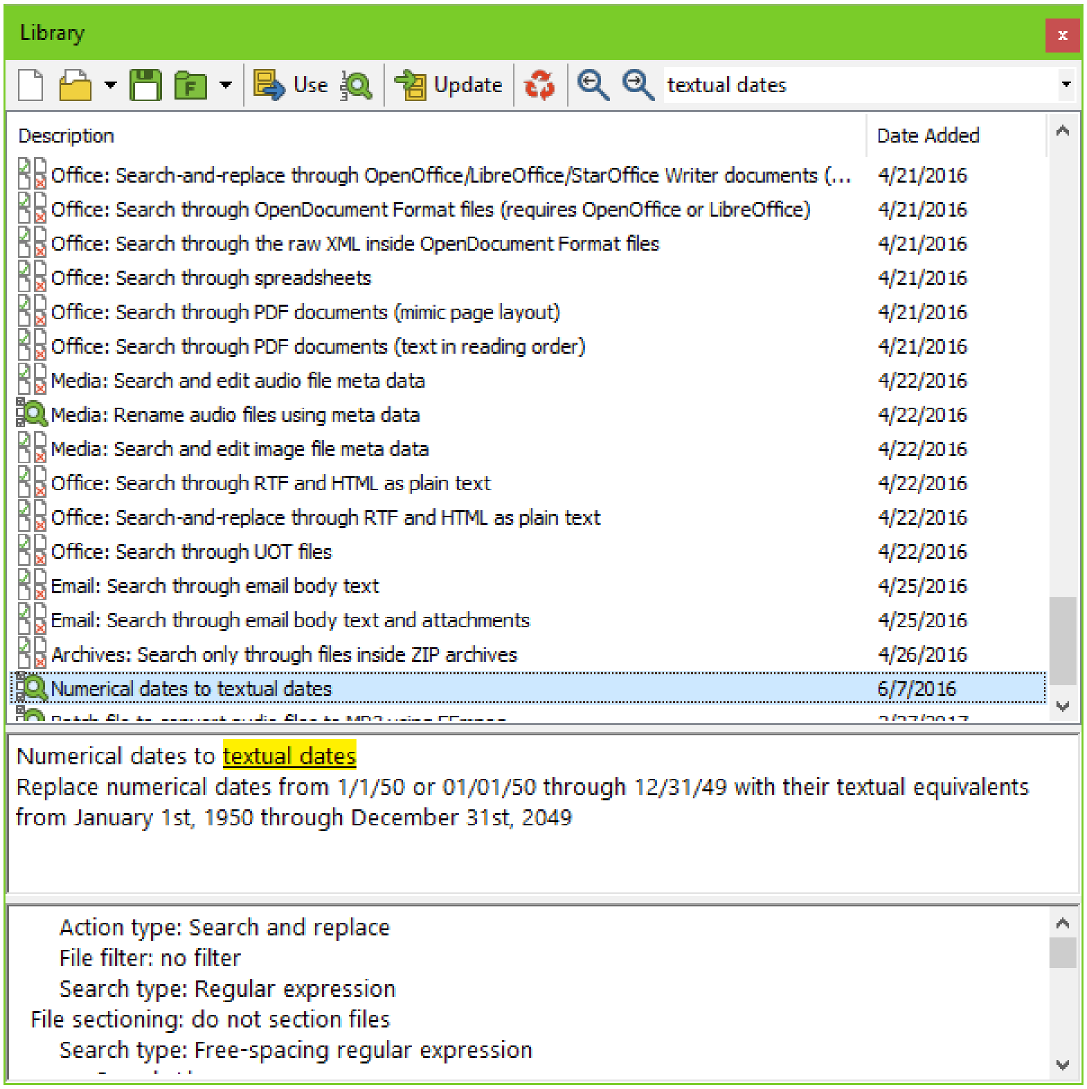
The Quick Execute item in the Sequence menu executes the sequence for real, without keeping individual match results. The Results panel will only show how many matches were found in each file. Files will be created or modified according to the target type of each action in the sequence.

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.

23. 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.



Description	Date Added
Office: Search-and-replace through OpenOffice/LibreOffice/StarOffice Writer documents (…	4/21/2016
Office: Search through OpenDocument Format files (requires OpenOffice or LibreOffice)	4/21/2016
Office: Search through the raw XML inside OpenDocument Format files	4/21/2016
Office: Search through spreadsheets	4/21/2016
Office: Search through PDF documents (mimic page layout)	4/21/2016
Office: Search through PDF documents (text in reading order)	4/21/2016
Media: Search and edit audio file meta data	4/22/2016
Media: Rename audio files using meta data	4/22/2016
Media: Search and edit image file meta data	4/22/2016
Office: Search through RTF and HTML as plain text	4/22/2016
Office: Search-and-replace through RTF and HTML as plain text	4/22/2016
Office: Search through UOT files	4/22/2016
Email: Search through email body text	4/25/2016
Email: Search through email body text and attachments	4/25/2016
Archives: Search only through files inside ZIP archives	4/26/2016
Numerical dates to textual dates	6/7/2016
Batch file to convert audio files to MP3 using FFmpeg	2/27/2017

Numerical dates to textual dates
Replace numerical dates from 1/1/50 or 01/01/50 through 12/31/49 with their textual equivalents from January 1st, 1950 through December 31st, 2049

```
    Action type: Search and replace
    File filter: no filter
    Search type: Regular expression
File sectioning: do not section files
    Search type: Free-spacing regular expression
```

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 Selector, 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, you can use the Add to Library menu item in the File Selector, Action, or Sequence menu 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. Alternatively, you can update the item by reselecting it the library and using the Update Item command in the Library menu. This updates the item without any prompting and changes the item's description if you changed it on the Action or Sequence panel.

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.

24. 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.

Update Item

Updates the selected item in the library with the contents of the File Selector, Action, or Sequence panels depending on which type of item you have selected in the library. When updating an action or sequence, the comments on the library item are also updated with those from the Action or Sequence panels. For file selections the library item keeps its comments, because there is no Comments box on the File Selector panel.

An alternative way to update items is to use the Add to Library item in the Action or Sequence menu. If the library already has an action or sequence with the same comments, PowerGREP will ask you if you want to update the item in the library with the new contents of the Action or Sequence panel.

Delete Item

Deletes the selected item from the library. This cannot be undone.

25. 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 uses 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 severs 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 severs in parallel, this order is not deterministic.
- **Alphanumeric: A..Z, 0..9**: Sort files in ascending alphanumeric order of their full path names.
- **Alphanumeric: Z..A, 9..0**: Sort files in descending alphanumeric 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.
- **Alphabetic: A..Z**: Sort matches alphabetically, from A to Z.
- **Alphabetic:, Z..A**: Sort matches alphabetically, from Z to A.
- **Alphanumeric: A..Z, 0..9**: Sort matches alphanumerically, from A to Z and from 0 to 9.
- **Alphanumeric: Z..A, 9..0**: Sort matches alphanumerically, from Z to A and from 9 to 0.
- **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.

26. 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, use View|Lock Toolbars to unlock the toolbars if you haven't already. Then 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. When you're done, you can lock the toolbars again to prevent accidental changes.

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.

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. You can find this item in the submenu o fthe Locate in File Selector menu item.

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. You can find this item in the submenu o fthe Locate in File Selector menu item.

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.

Open File in EditPad

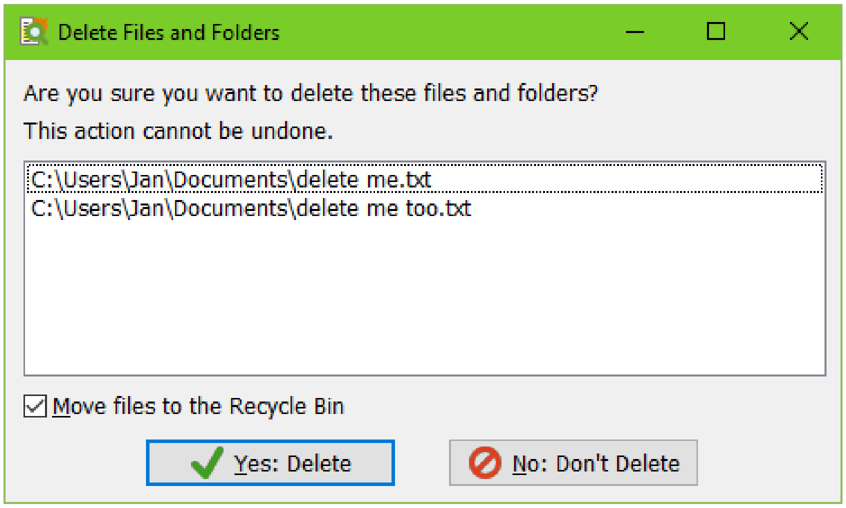
Click on a file name in the results, and select the Edit File command to open that file in EditPad. EditPad is a most convenient text editor. Just like PowerGREP, EditPad has been designed by Jan Goyvaerts and is sold by Just Great Software Co. Ltd. EditPad is available at http://www.editpadpro.com/.

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



Are you sure you want to delete these files and folders?

This action cannot be undone.

C:\Users\Jan\Documents\delete me.txt
C:\Users\Jan\Documents\delete me too.txt

☑ Move files to the Recycle Bin

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.

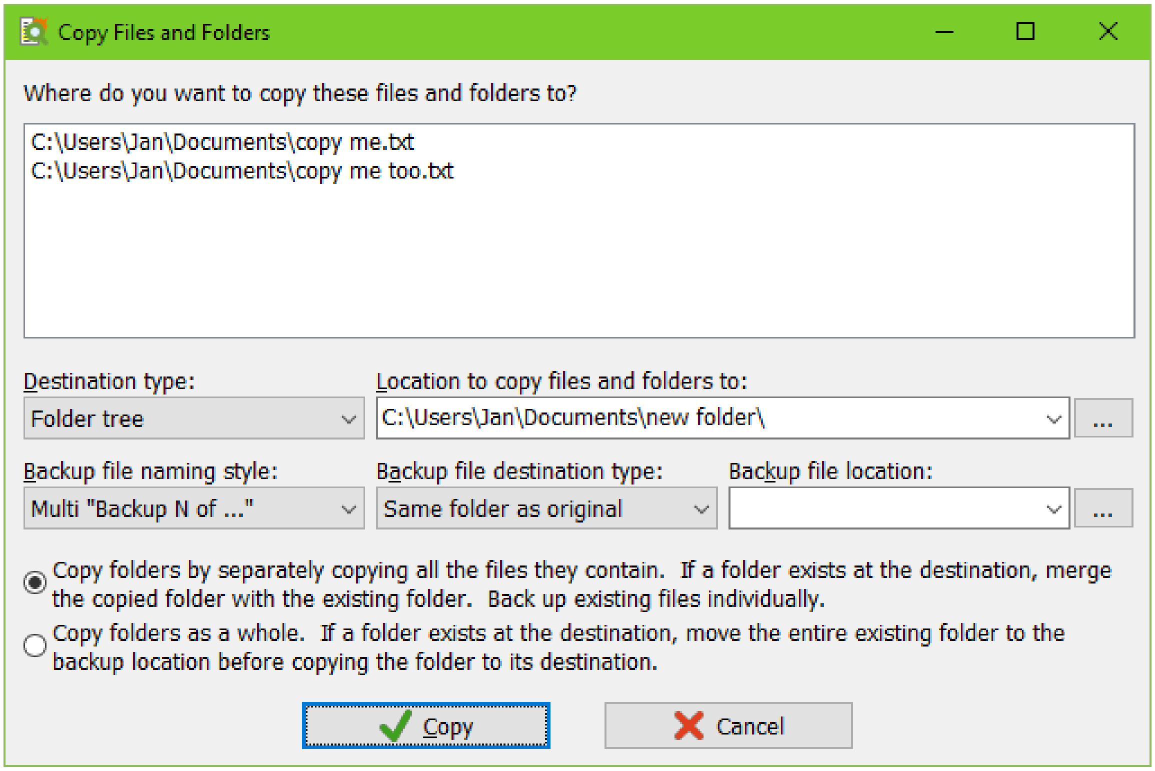
Copy Files

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.



Copy Files and Folders

Where do you want to copy these files and folders to?

C:\Users\Jan\Documents\copy me.txt
C:\Users\Jan\Documents\copy me too.txt

Destination type: Folder tree

Location to copy files and folders to: C:\Users\Jan\Documents\new folder\

Backup file naming style: Multi "Backup N of ..."

Backup file destination type: Same folder as original

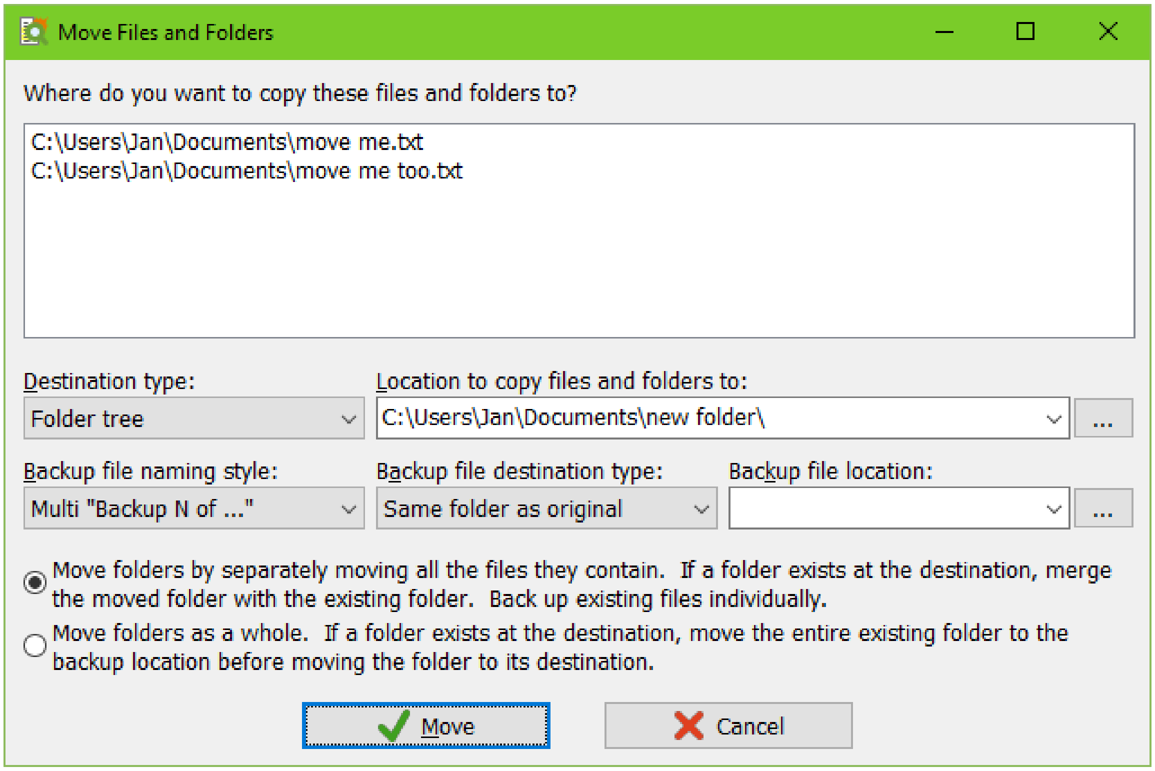
Backup file location:

● Copy folders by separately copying all the files they contain. If a folder exists at the destination, merge the copied folder with the existing folder. Back up existing files individually.

○ Copy folders as a whole. If a folder exists at the destination, move the entire existing folder to the backup location before copying the folder to its destination.

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.



Where do you want to copy these files and folders to?

C:\Users\Jan\Documents\move me.txt
C:\Users\Jan\Documents\move me too.txt

Destination type: Folder tree

Location to copy files and folders to: C:\Users\Jan\Documents\new folder\

Backup file naming style: Multi "Backup N of ..."

Backup file destination type: Same folder as original

Backup file location:

● Move folders by separately moving all the files they contain. If a folder exists at the destination, merge the moved folder with the existing folder. Back up existing files individually.

○ Move folders as a whole. If a folder exists at the destination, move the entire existing folder to the backup location before moving the folder to its destination.

27. 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 help 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.

28. 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, use View|Lock Toolbars to unlock the toolbars if you haven't already. Then 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. When you're done, you can lock the toolbars again to prevent accidental changes.

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

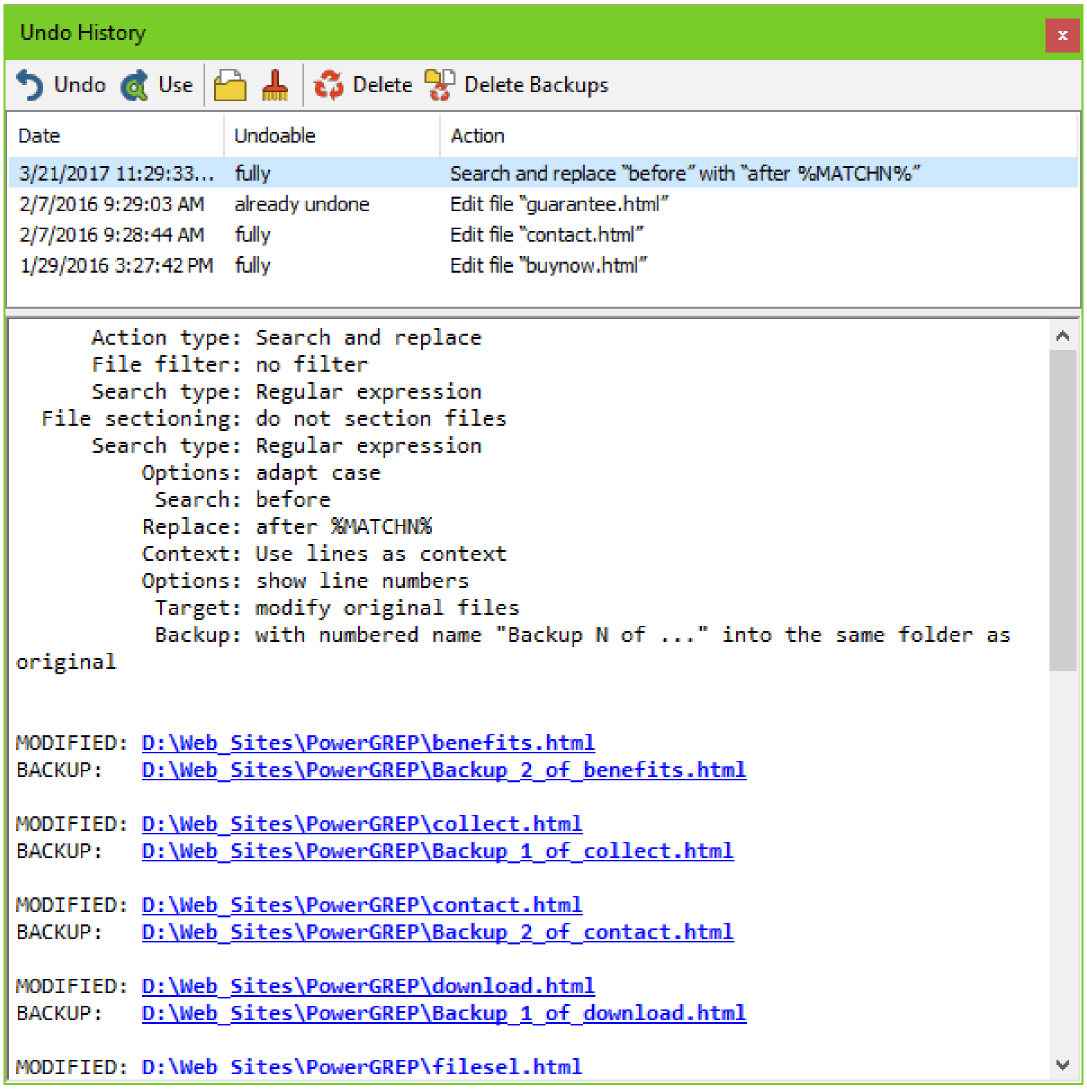
Toggles showing line numbers in the left margin on or off.

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.

29. 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.



Date	Undoable	Action
3/21/2017 11:29:33...	fully	Search and replace "before" with "after %MATCHN%"
2/7/2016 9:29:03 AM	already undone	Edit file "guarantee.html"
2/7/2016 9:28:44 AM	fully	Edit file "contact.html"
1/29/2016 3:27:42 PM	fully	Edit file "buynow.html"

```
      Action type: Search and replace
      File filter: no filter
      Search type: Regular expression
  File sectioning: do not section files
      Search type: Regular expression
          Options: adapt case
           Search: before
          Replace: after %MATCHN%
          Context: Use lines as context
          Options: show line numbers
           Target: modify original files
           Backup: with numbered name "Backup N of ..." into the same folder as
original


MODIFIED: D:\Web Sites\PowerGREP\benefits.html
BACKUP:   D:\Web Sites\PowerGREP\Backup 2 of benefits.html

MODIFIED: D:\Web Sites\PowerGREP\collect.html
BACKUP:   D:\Web Sites\PowerGREP\Backup 1 of collect.html

MODIFIED: D:\Web Sites\PowerGREP\contact.html
BACKUP:   D:\Web Sites\PowerGREP\Backup 2 of contact.html

MODIFIED: D:\Web Sites\PowerGREP\download.html
BACKUP:   D:\Web Sites\PowerGREP\Backup 1 of download.html

MODIFIED: D:\Web Sites\PowerGREP\filesel.html
```

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.

30. 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

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".

31. Change PowerGREP's Appearance and Layout

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 mid-sized monitor. If your computer has a large 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 dockedto 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 egdes 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

Turn the Large Toolbar Icons option on or off to switch PowerGREP's toolbars between using a larger or a smaller set of icons.

Lock Toolbars

By default, PowerGREP's toolbars are locked into place. If you turn off Lock Toolbars then you can move toolbars and dock them anywhere or let them float by dragging them with the mouse. When the toolbars are unlocked, you can also change which toolbar buttons are visible by clicking the small triangle at the far right hand end of each toolbar. This opens a drop-down menu where you can toggle each button. When you're done configuring the toolbars, you should lock them again, to avoid accidentally moving them with the mouse.

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 through Windows 8.1, the default looks use the Windows theme you selected in your computer's display settings. On Windows 10, the default look is a flat look that fits well with Windows 10.

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.

32. 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 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 is used to email you whenever others participate in one of your discussions. The email address is never displayed to anyone, and is never 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 is 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, 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 is a list where you can select which particular forum you want to participate in. The "PowerGREP" 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. If you find one, 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 stays together, and others can instantly see what you're talking about. It doesn't matter whether the conversation is many years old. If you reply to it, it becomes an active conversation that moves to the top automatically.

In the top right corner of the Forum panel, there is a box on the toolbar that you can use to search for messages. When you enter something into that box, only conversations that include at least one message containing the word or phrase you entered are shown.

The filtering happens in real time as you type in your word or phrase. It includes all conversation subjects, all message summaries, and all author names. Is also includes the message bodies of conversations that have been downloaded. PowerGREP automatically downloads the 20 most recent conversations when you connect to the forum. Other conversations are downloaded only if you click on them to view them. Downloaded messages are cached between PowerGREP sessions.

This means that the real time filtering does not search through the body text of older messages that you've never viewed. To search through those messages, click the Search Forum button that sits immediately to the right of the search box. PowerGREP then shows all conversations with messages containing the search term in their summaries, author names, or body texts, including messages that haven't been downloaded yet.

You can enter only one search term, which is searched for literally. If you enter "find me", only conversations containing the two words "find me" next to each other and in that order are shown. You cannot use boolean operators like "or" or "and". Since the filtering is instant, you can quickly try various keywords.

If your search term can be found in the subject of a conversation, then all messages in that conversation are always shown. If the search term cannot be found in the subject of a conversation, but it can be found in the summary or body text of a message in that conversation, then the Show Complete Conversations button in the far top right corner determines which messages are shown. When this button is up, only the messages in which the search term was found are shown for that conversation. When this button is down, all messages are shown for that conversation.

If you have previously participated in the forums, you can use the Show My Conversations button to show only conversations that you have participated in and conversations that have a message that you gave a +1. If you did not enter a search term, this shows all messages in all those conversations. If you did enter a search term, this reduces the search results to conversations that you participated in.

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 appears 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 moves to the 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 appears 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 uses the summary as the body text, and automatically prepends [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 forever keeps 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.

Dates and Times

The Started column indicates how long ago each conversation was started. The Last Reply column indicates how long ago the last reply was made, if any. For older conversations, it indicates how much later that reply came after the conversation was started. If you sort conversations by last reply, conversations without replies are sorted by their starting date. The sort order is always based on absolute dates.

The Date Posted column indicates how long ago each message was posted. For replies to older conversations, this date indicates how much later that message was posted after the conversation was started. The Date Edited column indicates how long ago the message was edited, if at all. For older messages, it indicates how much later it was edited after it was posted. If you sort messages by the date they were edited, messages that weren't edited are sorted by their posting date. The sort order is always based on absolute dates.

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 panel, 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 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 is 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 has to decompress it manually. If you let PowerGREP compress it automatically, decompression is also 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. It is labeled Cancel if you have not yet sent your message. 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 instantly. 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 Cancel button. Make sure to click it only once! When editing a message, the Delete button changes its caption to Cancel and when clicked reverts 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 does 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 are 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 are marked as "unread". This is indicated with the same "people in the cloud" icon as shown next to the Login button.

33. Forum RSS Feeds

When you're connected to the user forum, you can click the Feeds button to select RSS feedsthat 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:

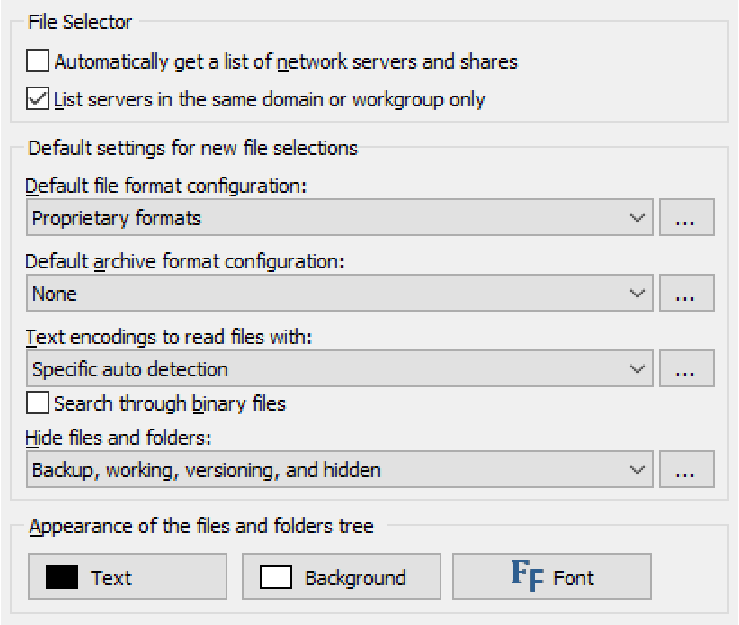
- All conversations in all groups: show all conversations in all the discussion groups that you can access in PowerGREP.
- All conversations in the selected group: show the list of conversations that PowerGREP is presently showing on the Forum panel.
- All conversations you participated in: show all 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 panel 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.

34. File Selector Preferences

In the File Selector section of the Preferences screen, you can configure PowerGREP's File Selector and the way PowerGREP handles the network.



File Selector

☐ Automatically get a list of network servers and shares

☑ List servers in the same domain or workgroup only

Default settings for new file selections

Default file format configuration: Proprietary formats

Default archive format configuration: None

Text encodings to read files with: Specific auto detection

☐ Search through binary files

Hide files and folders: Backup, working, versioning, and hidden

File Selector

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.

Default Settings for New File Selections

On the File Selector panel in PowerGREP, you can select four different configurations that control how PowerGREP deals with certain files.

- File format configuration: Specify which files, if any, should be treated as files (in proprietary file formats) that need to be converted into plain text before they can be searched through.
- Archive format configuration: Specify which files, if any, should be treated as compressed archives (such as ZIP files), disk images (such as ISO files), or mailboxes (such as PST files) that contain other files or email messages that need to be searched through.
- Text encodings to read files with: Specify which text encodings should be used to interpret plain text files.
- Hide files and folders: Specify which files and folders, if any, should be completely invisible to PowerGREP's File Selector.

In the File Selector section in the Preferences, you can select default configurations. These are used by the File Selector|Clear menu item. They are also the defaults when PowerGREP starts up if you select "do not remember the File Selection or Action" in the General section in the Preferences.

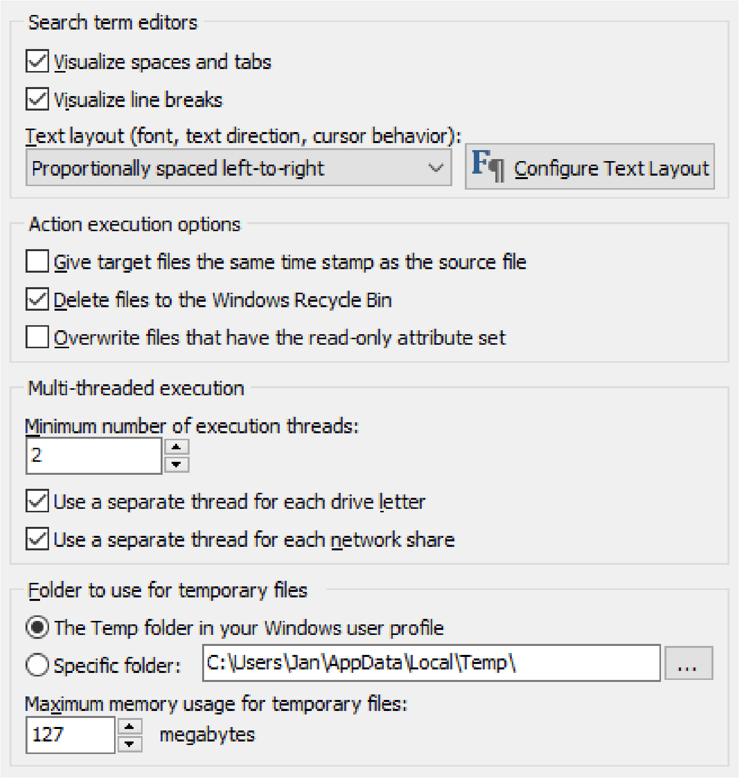
Click the (...) button to edit the configurations. If you edit a configuration presently selected on the File Selector panel, those changes take effect immediately. But editing configurations in the Preferences does not change the behavior of previously saved file selections. When you save a file selection, it stores the full details of the selected configurations. When you load a file selection, it continues to use the configuration you saved it with. If you edited that configuration in the Preferences, the configuration loaded with the file selection will be indicated with a number such as (2) to indicate its details are different from the configuration with the same name in the Preferences. If you want the loaded file selection to use the edited configuration, then you need need to select the edited configuration (without the number in parenthesis) on the File Selector panel after loading the exiting file selection.

Appearance of The Files And Folders Tree

Choose the color of the text, the color of the background, and the font used for the files and folders tree on the File Selector panel.

35. 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

☑ Visualize spaces and tabs

☑ Visualize line breaks

Text layout (font, text direction, cursor behavior): Proportionally spaced left-to-right

Action execution options

□ Give target files the same time stamp as the source file

☑ Delete files to the Windows Recycle Bin

□ Overwrite files that have the read-only attribute set

Multi-threaded execution

Minimum number of execution threads: 2

☑ Use a separate thread for each drive letter

☑ Use a separate thread for each network share

Folder to use for temporary files

● The Temp folder in your Windows user profile

○ Specific folder: C:\Users\Jan\AppData\Local\Temp\

Maximum memory usage for temporary files: 127 megabytes

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

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.

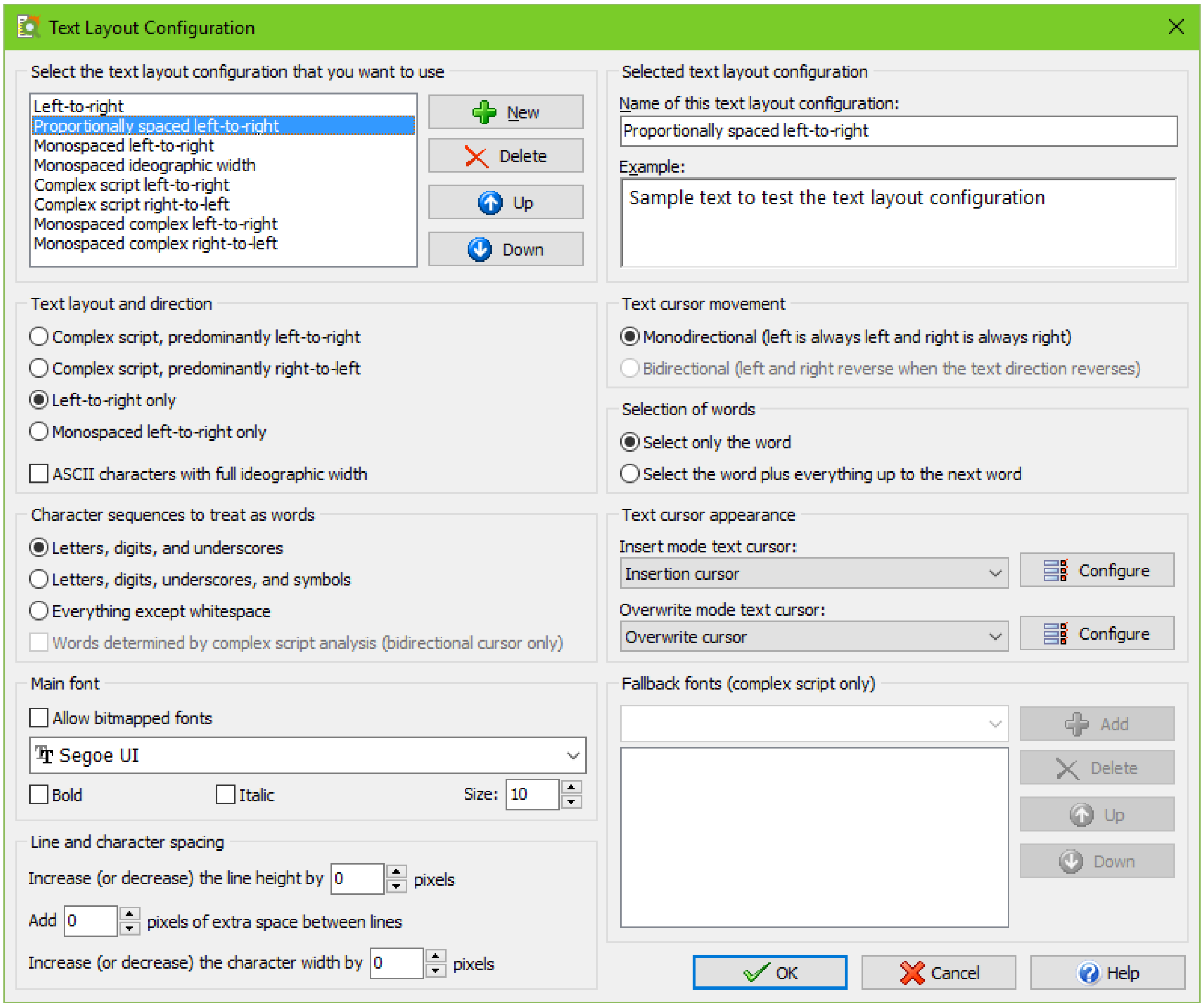
PowerGREP does not use this setting for preparing target files that you want it to save on a local hard disk. Those temporary files are created in the destination folder of the target file. That allows PowerGREP to instantly replace the target with the temporary file instead of having to copy it around.

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.

36. 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.



Text Layout Configuration

Select the text layout configuration that you want to use

- Left-to-right
- Proportionally spaced left-to-right
- Monospaced left-to-right
- Monospaced ideographic width
- Complex script left-to-right
- Complex script right-to-left
- Monospaced complex left-to-right
- Monospaced complex right-to-left

Selected text layout configuration

Name of this text layout configuration: Proportionally spaced left-to-right

Example: Sample text to test the text layout configuration

Text layout and direction

○ Complex script, predominantly left-to-right
○ Complex script, predominantly right-to-left
● Left-to-right only
○ Monospaced left-to-right only
□ ASCII characters with full ideographic width

Text cursor movement

● Monodirectional (left is always left and right is always right)
○ Bidirectional (left and right reverse when the text direction reverses)

Selection of words

● Select only the word
○ Select the word plus everything up to the next word

Character sequences to treat as words

● Letters, digits, and underscores
○ Letters, digits, underscores, and symbols
○ Everything except whitespace
□ Words determined by complex script analysis (bidirectional cursor only)

Text cursor appearance

Insert mode text cursor: Insertion cursor

Overwrite mode text cursor: Overwrite cursor

Main font

□ Allow bitmapped fonts

Segoe UI

□ Bold □ Italic Size: 10

Fallback fonts (complex script only)

Line and character spacing

Increase (or decrease) the line height by 0 pixels

Add 0 pixels of extra space between lines

Increase (or decrease) the character width by 0 pixels

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 tying 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.

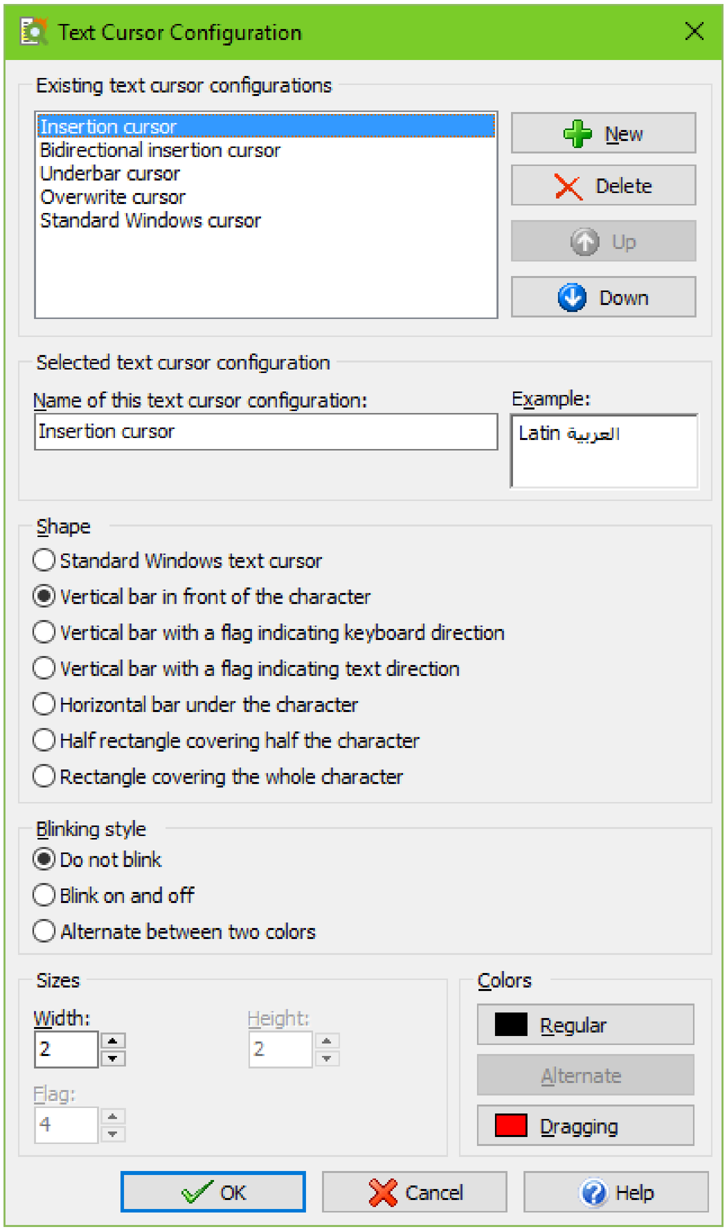
If you find that lines are spaced apart too widely, specify a negative value for “increase (or decrease) the line height”. Set to “add 0 pixels of extra space between lines”.

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.

37. 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.



Text Cursor Configuration

Existing text cursor configurations

- Insertion cursor
- Bidirectional insertion cursor
- Underbar cursor
- Overwrite cursor
- Standard Windows cursor

Selected text cursor configuration

Name of this text cursor configuration: Insertion cursor

Example: Latin العربية

Shape

○ Standard Windows text cursor
● Vertical bar in front of the character
○ Vertical bar with a flag indicating keyboard direction
○ Vertical bar with a flag indicating text direction
○ Horizontal bar under the character
○ Half rectangle covering half the character
○ Rectangle covering the whole character

Blinking style

● Do not blink
○ Blink on and off
○ Alternate between two colors

Sizes

Width: 2
Height: 2
Flag: 4

Colors

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 tying 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.

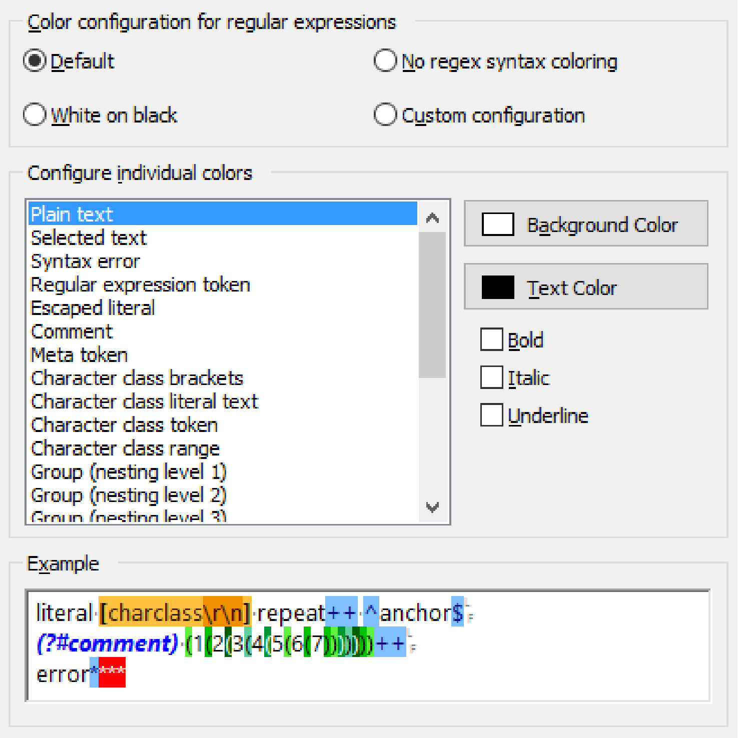
Colors

- 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.

38. 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



Color configuration for regular expressions

● Default
○ No regex syntax coloring
○ White on black
○ Custom configuration

Configure individual colors

- Plain text
- Selected text
- Syntax error
- Regular expression token
- Escaped literal
- Comment
- Meta token
- Character class brackets
- Character class literal text
- Character class token
- Character class range
- Group (nesting level 1)
- Group (nesting level 2)
- Group (nesting level 3)

□ Bold
□ Italic
□ Underline

Example

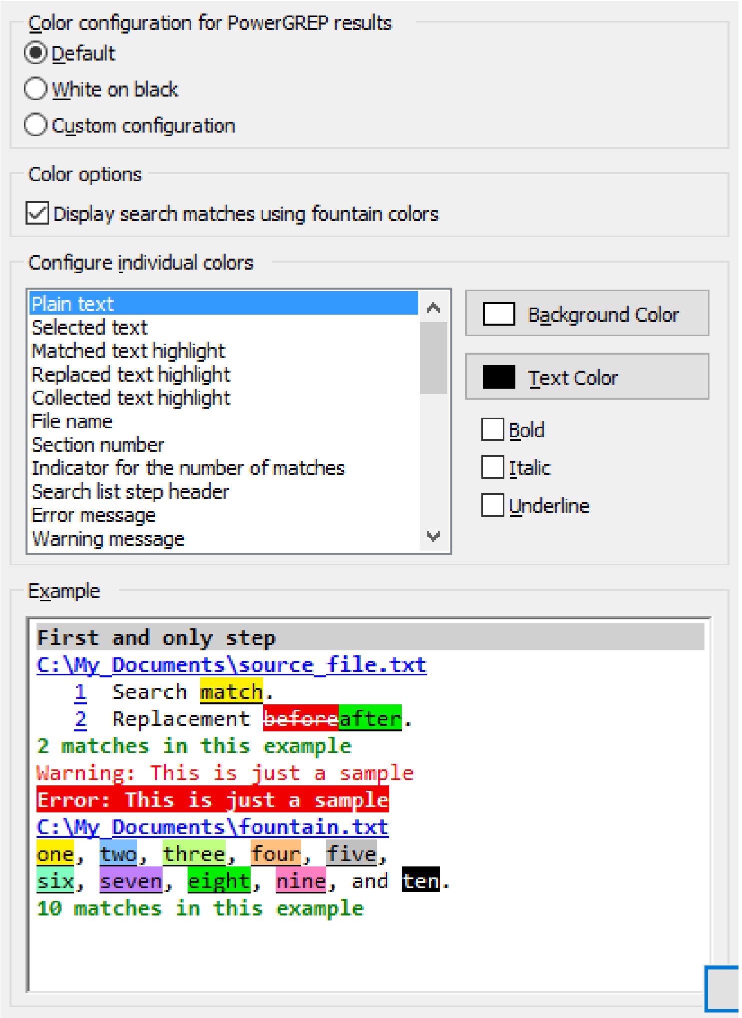
```
literal [charclass\r\n] repeat++ ^anchor$
(?#comment) (1(2(3(4(5(6(7)))))))++
error***
```

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



Color configuration for PowerGREP results

● Default
○ White on black
○ Custom configuration

Color options

☑ Display search matches using fountain colors

Configure individual colors

- Plain text
- Selected text
- Matched text highlight
- Replaced text highlight
- Collected text highlight
- File name
- Section number
- Indicator for the number of matches
- Search list step header
- Error message
- Warning message

□ Bold
□ Italic
□ Underline

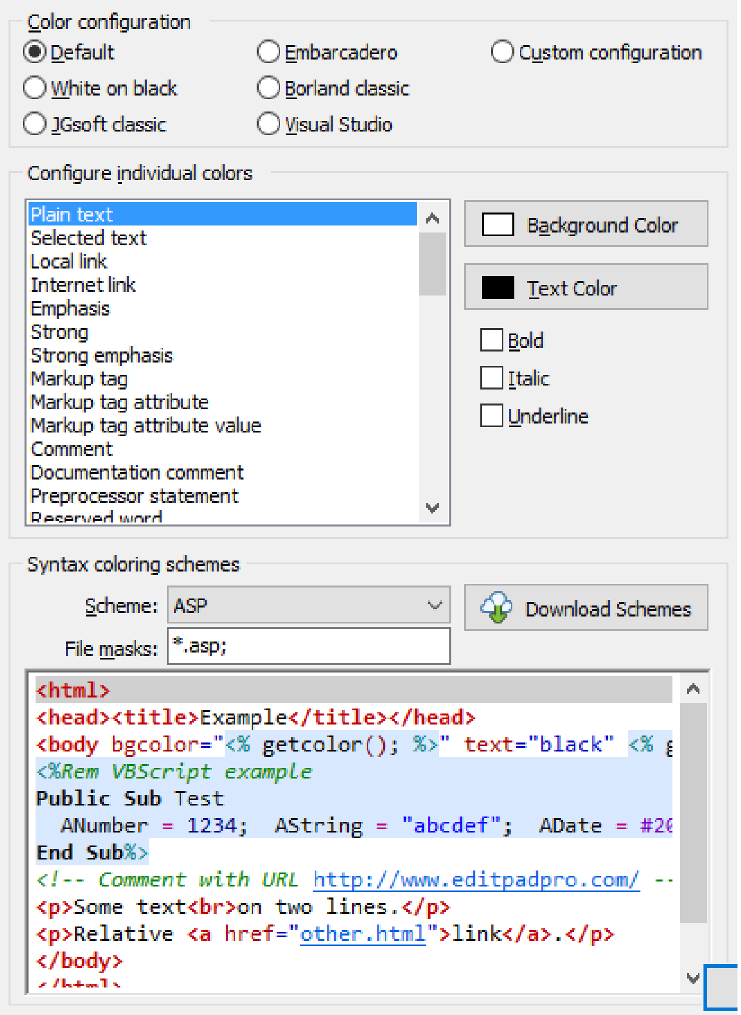
Example

```
First and only step
C:\My Documents\source file.txt
   1  Search match.
   2  Replacement beforeafter.
2 matches in this example
Warning: This is just a sample
Error: This is just a sample
C:\My Documents\fountain.txt
one, two, three, four, five,
six, seven, eight, nine, and ten.
10 matches in this example
```

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



Color configuration

● Default ○ Embarcadero ○ Custom configuration

○ White on black ○ Borland classic

○ JGsoft classic ○ Visual Studio

Configure individual colors

□ Bold

□ Italic

□ Underline

Syntax coloring schemes

Scheme: ASP

File masks: *.asp;

```
<html>
<head><title>Example</title></head>
<body bgcolor="<% getcolor(); %>" text="black" <% g
<%Rem VBScript example
Public Sub Test
  ANumber = 1234;  AString = "abcdef";  ADate = #20
End Sub%>
<!-- Comment with URL http://www.editpadpro.com/ --
<p>Some text<br>on two lines.</p>
<p>Relative <a href="other.html">link</a>.</p>
</body>
</html>
```

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.

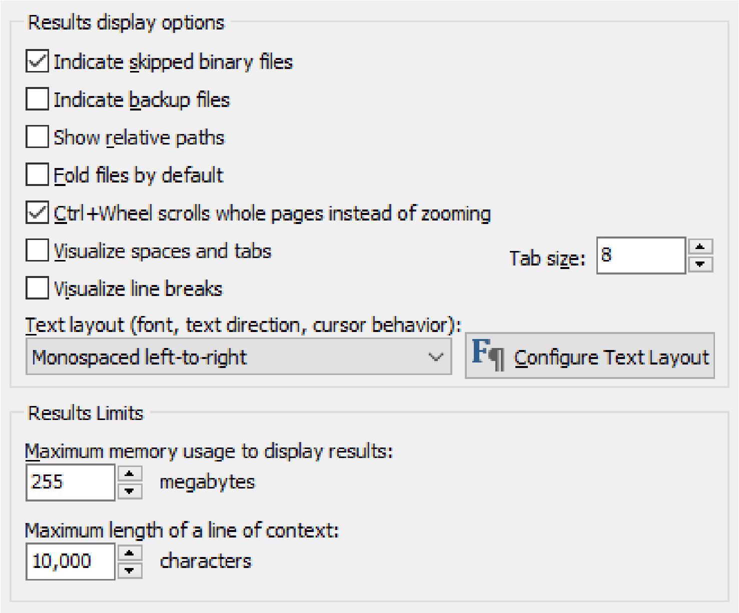
To download additional syntax coloring schemes shared by other PowerGREP and EditPad Pro users, click the Download Schemes button. PowerGREP will show a list of available syntax coloring schemes. Simply select a scheme and click the Install button to download it.

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.

39. 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

☑ Indicate skipped binary files

□ Indicate backup files

□ Show relative paths

□ Fold files by default

☑ Ctrl+Wheel scrolls whole pages instead of zooming

□ Visualize spaces and tabs

Tab size: 8

□ Visualize line breaks

Text layout (font, text direction, cursor behavior): Monospaced left-to-right

Results Limits

Maximum memory usage to display results: 255 megabytes

Maximum length of a line of context: 10,000 characters

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 pathsin 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 computers 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

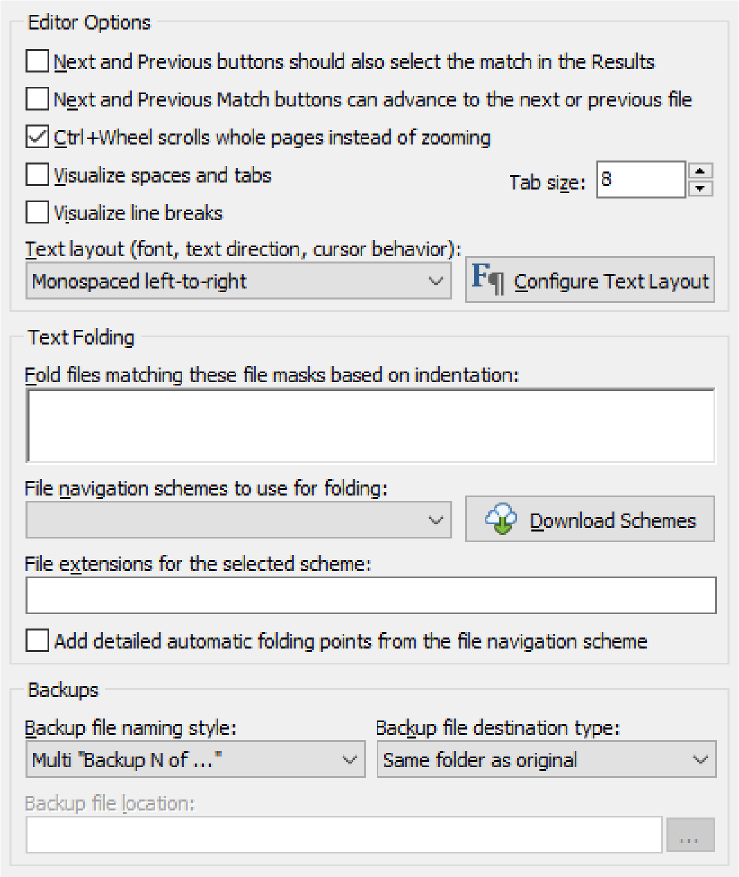
While showing one line of context is usually very convenient, extremely long lines (such as a large XML file with everything on one line) make it hard to see the matches as you'll need to do a lot of horizontal scrolling. The setting for the maximum length of a line of context acts as a safety valve. 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.

40. Editor Preferences

In the Editor section in the Preferences screen, you can configure the behavior of PowerGREP's built-in file editor.



Editor Options

□ Next and Previous buttons should also select the match in the Results

□ Next and Previous Match buttons can advance to the next or previous file

☑ Ctrl+Wheel scrolls whole pages instead of zooming

□ Visualize spaces and tabs

Tab size: 8

□ Visualize line breaks

Text layout (font, text direction, cursor behavior): Monospaced left-to-right

Text Folding

Fold files matching these file masks based on indentation:

File navigation schemes to use for folding:

File extensions for the selected scheme:

□ Add detailed automatic folding points from the file navigation scheme

Backups

Backup file naming style: Multi "Backup N of ..."

Backup file destination type: Same folder as original

Backup file location:

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.

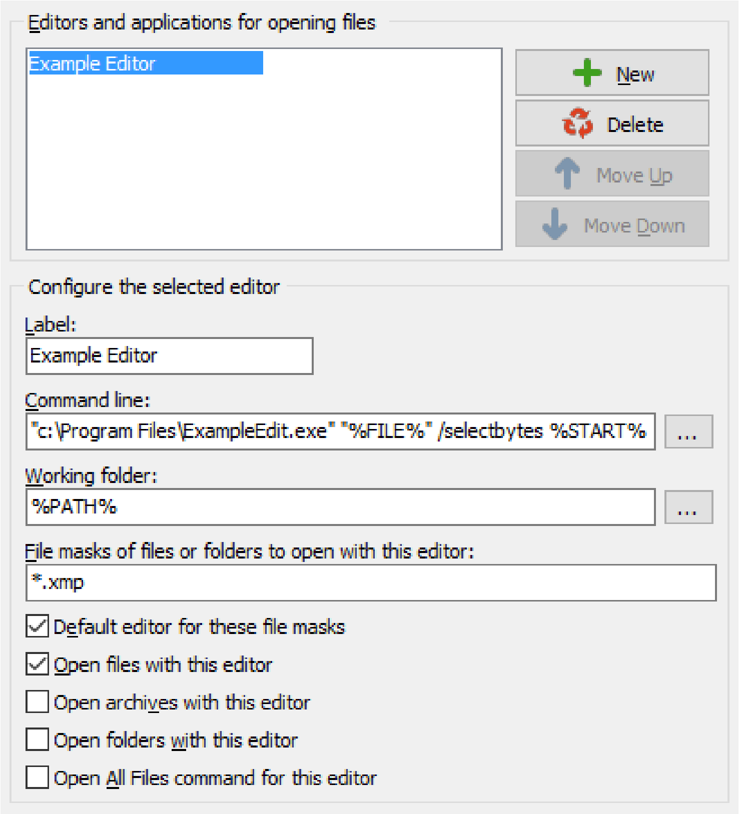
Backups

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.

41. 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.



Editors and applications for opening files

Example Editor

Configure the selected editor

Label: Example Editor

Command line: "c:\Program Files\ExampleEdit.exe" "%FILE%" /selectbytes %START%

Working folder: %PATH%

File masks of files or folders to open with this editor: *.xmp

☑ Default editor for these file masks

☑ Open files with this editor

☐ Open archives with this editor

☐ Open folders with this editor

☐ Open All Files command for this editor

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 Files with This Editor

Turn on to allow this editor to appear in the Edit submenu for files matched by the file mask for this editor; or all files if you don't specify a file mask.

If you turn off this option, the editor will not be available for any files.

You can use the %FILE% placeholder to represent the file on the command line.

Open Archives with This Editor

Turn on to allow this editor to appear in the Edit submenu for archives matched by the file mask for this editor; or all archives if you don't specify a file mask.

If you turn off this option, the editor will not be available for any archives.

You can use the %FILE% placeholder to represent the archive on the command line.

Which files are treated as archives depends on the archive configuration selected in the File Selector.

Open Folders with This Editor

Turn on to allow this editor to appear in the Edit submenu for folders matched by the file mask for this editor; or all folders if you don't specify a file mask.

If you turn off this option, the editor will not be available for any folders.

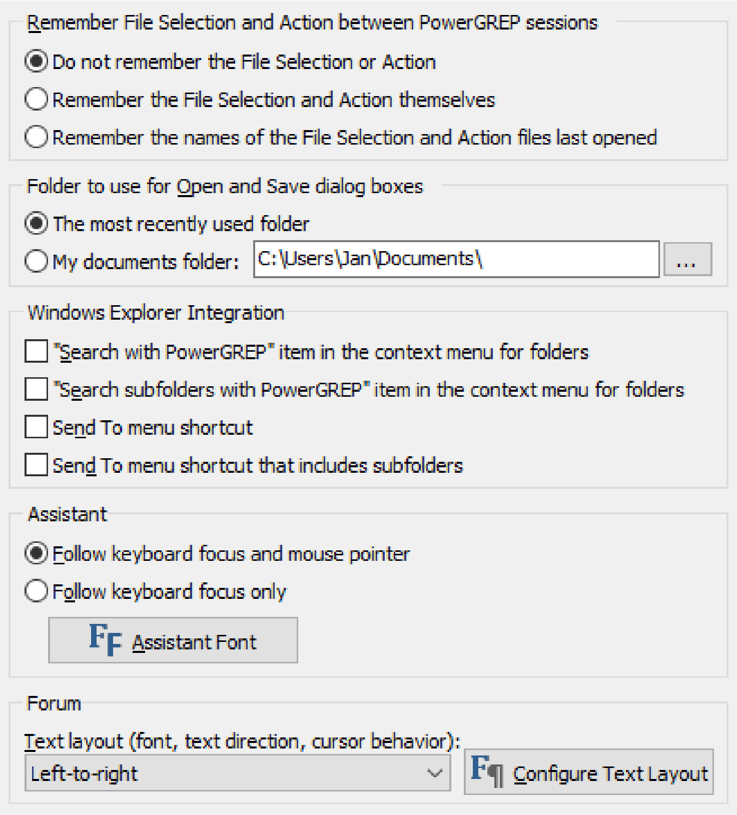
You can use the %PATH% placeholder to represent the folder on the command line.

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.

42. General Preferences



Remember File Selection and Action between PowerGREP sessions

● Do not remember the File Selection or Action

○ Remember the File Selection and Action themselves

○ Remember the names of the File Selection and Action files last opened

Folder to use for Open and Save dialog boxes

● The most recently used folder

○ My documents folder: C:\Users\Jan\Documents\ ...

Windows Explorer Integration

□ "Search with PowerGREP" item in the context menu for folders

□ "Search subfolders with PowerGREP" item in the context menu for folders

□ Send To menu shortcut

□ Send To menu shortcut that includes subfolders

Assistant

● Follow keyboard focus and mouse pointer

○ Follow keyboard focus only

Forum

Text layout (font, text direction, cursor behavior): Left-to-right

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.

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.

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.

Windows Explorer Integration

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

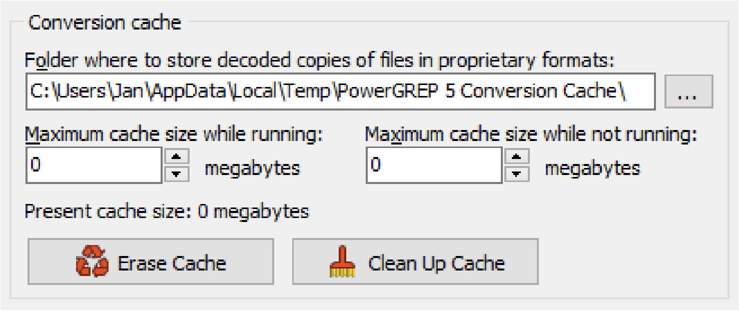
Select the font used by the PowerGREP Assistant which displays helpful hints while you work with PowerGREP.

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.

43. Cache Preferences

In the File Selector, you can select a file format configuration that may specify that PowerGREP should convert files in proprietary formats into plain text so that they can be searched through. In the Cache section in the Preferences screen, you can specify whether PowerGREP should keep a cache of these plain text conversions.



Conversion cache

Folder where to store decoded copies of files in proprietary formats: C:\Users\Jan\AppData\Local\Temp\PowerGREP 5 Conversion Cache\

Maximum cache size while running: 0 megabytes

Maximum cache size while not running: 0 megabytes

Present cache size: 0 megabytes

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 moves its cache. If you select a folder on another drive, PowerGREP clears its cache. You should select a folder on a local drive. If your computer has an SSD (solid state drive) and a traditional spinning hard drive, choose a folder on the SSD to maximize the performance benefit of the cache.

All PowerGREP instances running on your computer automatically share the conversion cache on your computer. PowerGREP instances running on other computers cannot share the cache on your computer. You should choose a local hard drive rather than a network drive to store the cache for maximum performance and to make sure that PowerGREP instances on other computers won't use the same cache. PowerGREP will store its cache on a network drive if you tell it to, but you have to make sure that only one computer will access the cache at any time. If two people using PowerGREP on their own computers access a networked cache folder at the same time, the cache will become corrupted.

The cache should be large enough to store all the files you're working with. If PowerGREP runs out of cache space during a search, it prunes files that were converted during previous searches from the cache. It first prunes the files that you searched through the longest ago. If after pruning all files from previous searches the cache still isn't large enough to hold all the files converted during the present search, then PowerGREP stops adding files to the cache during the present search.

If only half the files you're searching through fit into the cache, then the first half of the files will be cached when the search completes. If you then run another search on the same set of files, the first half of the files will be read from the cache, and the second half will be converted again.

If the cache is large enough then all files will be in the cache when the search completes. If you then search through the same set of files again, PowerGREP will 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 1,000 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 trims 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 clears the cache entirely when you close the last instance. If you set the “running” size to zero, PowerGREP never caches 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 Format Configuration and for which you’ve selected one of the options to convert the files using an external application, an IFilter, or PowerGREP’s built-in decoder. When estimating the required size of the cache, you only need to count these files.

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.

Manual Caching

There may be situations where you want to search repeatedly through the same set of files but don’t want to use the conversion cache. Maybe there’s not enough space on your PC for the cache or maybe others on your network need to search through the same files. In such situations, you can follow the example titled “convert files in proprietary formats to plain text" to create your own cache of plain text conversions on a network server.

44. 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
%ACTIONAMPM%	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. Enter 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 action was last modified.	Everywhere

%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
%FILEAMPM%	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 (1,000 bytes) of the file currently being processed by the action (prior to any changes made by the action).	Everywhere
%FILESIZEKIB%	Size in kibibytes (1,024 bytes) of the file currently being processed by the action (prior to any changes made by the action).	Everywhere
%FILESIZEMB%	Size in megabytes (1,000,000 bytes) of the file currently being processed by the action (prior to any changes made by the action).	Everywhere
%FILESIZEMIB%	Size in mebibytes (1,048,576 bytes) of the file currently being processed by the action (prior to any changes made by the action).	Everywhere
%FILESIZEGB%	Size in gigabytes (1,000,000,000 bytes) of the file currently being processed by the action (prior to any changes made by the action).	Everywhere
%FILESIZEGIB%	Size in gibibytes (1,073,741,824 bytes) 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. Enter 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. Enter 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` and `$1` in the replacement text. The advantage	In any regular expression and any replacement text

	of %GROUP1% is that you can use the padding and case conversion specifiers listed below. Example: Padding replacements	corresponding with a regular expression.
%MATCHSTART%	The byte offset relative to the start of the file of the first character in the search match. Zero for a match at the very start of the file.	In the replacement text of the main action and extra processing.
%MATCHSTOP%	The byte offset relative to the start of the file of the first character after the search match. Equal to the size of the file for a match at the very end of the file.	In the replacement text of the main action and extra processing.
%GROUP1START%, %GROUP2START%, etc.	The byte offset relative to the start of the file of the first character of the text matched by capturing group number 1, 2, etc. For named capturing groups you can use %GROUPNAMESTART% where NAME is the name of the group. Group names are case sensitive.	In any replacement text corresponding with a regular expression that has capturing groups.
%GROUP1STOP%, %GROUP2STOP%, etc.	The byte offset relative to the start of the file of the first character after the text matched by capturing group number 1, 2, etc. For named capturing groups you can use %GROUPNAMESTOP% where NAME is the name of the group. Group names are case sensitive.	In any replacement text corresponding with a regular expression that has capturing groups.
%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. Enter 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	In the search terms and replacement text of the main action and extra processing.

	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	
%MATCHFILENA%	The sequential letter of the search match. The first match is “a”, the second “b”, etc. Enter 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 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. Example: Add line numbers	In the search terms and replacement text of the main action and extra processing, but only when using file sectioning.
%MATCHSECTIONNA%	The sequential letter of the search match. The first match is “a”, the second “b”, etc. Enter 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	In the search terms and replacement text of the main action and extra processing.

	other search terms.	
%MATCHTERMNA%	The sequential letter of the search match. The first match is “a”, the second “b”, etc. Enter 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 %MATCHFILETERMNZ%	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 own numbering sequence independent of the other search terms.	In the search terms and replacement text of the main action and extra processing.
%MATCHFILETERMNA%	The sequential letter of the search match. The first match is “a”, the second “b”, etc. Enter 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.
%MATCHSECTIONTERMN% and %MATCHSECTIONTERMNZ%	The number of the search match. %MATCHSECTIONTERMN% starts counting at one, and %MATCHSECTIONTERMNZ% 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.
%MATCHSECTIONTERMNA%	The sequential letter of the search match. The first match is “a”, the second “b”, etc. Enter 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	In the search terms and replacement text of the main action and extra processing, but only when using

	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.	file sectioning.
%TERMN% and %TERMNZ%	The number of the search term in the list of search terms or the delimited search terms in the main part of the action that is being searched for or that found the match being replaced or collected. The number depends solely on the order of the search terms in the main part of the action as you entered them on the Action panel. %TERMN% starts counting at one, and %TERMNZ% at zero.	In the search terms and replacement text of the main action and extra processing.
%TERMNA%	The sequential letter of the search term in the list of search terms or the delimited search terms in the main part of the action that is being searched for or that found the match being replaced or collected. The letter depends solely on the order of the search terms in the main part of the action as you entered them on the Action panel.	In the search terms and replacement text of the main action and extra processing. Enter the placeholder in upper or lowercase to determine the case of the letter.
%MATCHCOUNT% (collect)	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.
%MATCHCOUNT% (footers)	The number of matches found in a file when used in source file footers. The number of matches written to a target file when used in target file footers.	Only available in file footers when collecting headers and footers.
%FILECOUNT%	The number of source files that had their matches written to a target file.	Only available in target file footers when collecting headers and footers.
%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.
%GUIDFILE%	Generates a new GUID for each file searched through. The GUID stays constant when the placeholder is used multiple times for the same file.	In replacement text, text to be collected, and file headers and footers.
%GUID1% to	Generates nine different GUIDs each time an action or sequence is executed. The nine GUIDs stays	In replacement text, text to be

%GUID9%	constant throughout the execution of the action or sequence.	collected, and file headers and footers.

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. Arithmetic specifiers are evaluated strictly from left to right. When %MATCHN% evaluates to 2, %MATCHN:+1*2% evaluates to 6 because 2+1=3 and 3*2=6. Multiplication and division do not take precedence over addition and subtraction. Integer division drops the fractional part of the division's result, so %MATCHN:/3% evaluates to 0 for the first two matches.

45. 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: Process files in a batch file or script, 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\

46. 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. `/replacetext ""` 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 sequence 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 kind. 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.

If you specify both an action and a sequence file then both are loaded into their respective panels. But the parameters for previewing and executing will execute only the sequence.

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. `/collect`sets the action type to “collect data”.

4. `/count`sets the action type to “count matches”.

5. `/find`sets 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.

6. `/findname`sets the action type to “file or folder name search”.

7. `/collectname`sets the action type to “file or folder name collect”.

8. `/rename`sets the action type to “rename files or folders”.

9. `/replace`sets the action type to “search and replace”.

10. `/delete`sets the action type to “search and delete”.

11. `/merge`sets the action type to “merge files”.

12. `/split`sets the action type to “split files”.

13. `/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`.

14. `/replacetext <text>`sets the replacement text or text to be collected to “text”.

15. `/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.

16. `/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.

17. `/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.

18. `/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.

19. `/regex`sets the search type to a regular expression.

20. `/literal`sets the search type to literal text.

21. `/delimitprefix <delimiter>`sets the search prefix label delimiter to "delimiter". Also sets the search type to a delimited list.

22. `/delimitsearch <delimiter>`sets the search item delimiter to "delimiter". Also sets the search type to a delimited list.

23. `/delimitreplace <delimiter>`sets the search pair delimiter to "delimiter". Also sets the search type to a delimited list.

24. `/optnonoverlap <0|1>`Sets the option "non-overlapping search". Only has an effect when the search type is a delimited list.

25. `/optdotall <0|1>`Sets the option "dot matches newlines". Only has an effect when the search type is a regular expression.

26. `/optwords <0|1>`Sets the option "whole words only". Does not have any effect when the search type is a regular expression.

27. `/optcase <0|1>`Sets the option "case sensitive".

28. `/optadaptive <0|1>`Sets the option "adaptive case".

29. `/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.

30. `/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.

31. `/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`.

32. `/target <type> <destination> <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)
`clipboard`	Copy the results to the clipboard. (search; collect data; list files; merge files)
`editor`	Store the results as an unsaved file on the Editor panel. (search; collect data; list files; merge files)
`move`	Move matching files (list files; rename files)
`"copy recurse"`	Copy the contents of matching folders (list folders; rename folders)
`"move recurse"`	Move the contents of matching folders (list folders; rename folders)
`convert`	Convert matched files to text. Do not specify any destination type or location. (list files)
`"convert copy"`	Convert copies of matched files to text. (list 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.

33. `/backup <type> <destination> <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`	Single backup with .~* extension

`tilde"`	
`"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.

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 files> <exclude files> <include folders> <exclude folders> <0|1>`Sets the file masks. The first two values are the inclusion and exclusion masks for files. The third and fourth value are the inclusion and exclusion masks for folders. The fifth value indicates whether the masks are traditional file masks (0) or regular expressions (1).

40. `/configconvert <configuration name>`Sets "file formats to convert to plain text" to the configuration with the given name. This configuration should have been previously defined in the File Selector section in the Preferences. Passing the empty string "" as the configuration name is the same as passing "(unused)", which is not the same as passing "None".

41. `/configarchives <configuration name>`Sets “archive formats to search inside” to the configuration with the given name. This configuration should have been previously defined in the File Selector section in the Preferences. Passing the empty string “” as the configuration name is the same as passing “(unused)”, which is not the same as passing “None”.

42. `/configencoding <configuration name>`Sets “text encodings to read files with” to the configuration with the given name. This configuration should have been previously defined in the File Selector section in the Preferences. Passing the empty string “” as the configuration name is the same as passing “(unused)”.

43. `/confighide <configuration name>`Sets “hide files and folders” to the configuration with the given name. This configuration should have been previously defined in the File Selector section in the Preferences. Passing the empty string “” as the configuration name is the same as passing “(unused)”.

44. `/optbinary <0|1>`Sets the option “search through binary files”.

45. `/resultsoptions <display> <group> <totals> <sortfiles> <sortmatches> <replacements>`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 six values, one for each of the six 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
`"alnum inc"`	Alphanumerically, A..Z, 0..9
`"alnum dec"`	Alphanumerically, Z..A, 0..9
`"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
`"alnum inc"`	Alphanumerically, A..Z, 0..9
`"alnum dec"`	Alphanumerically, Z..A, 0..9
`"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

46. `/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.

47. `/preview`Preview the action. If you passed a PowerGREP sequence action file on the command line then this parameter previews the whole sequence.

48. `/execute`Execute the action. If you passed a PowerGREP sequence action file on the command line then this parameter executes the whole sequence.

49. `/quick`Quick execute the action. If you passed a PowerGREP sequence action file on the command line then this parameter quick executes the whole sequence.

50. `/quit`This 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.

51. `/reuse`Tells 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 appears 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.

52. `/silent`Quick execute the action or sequence 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 appears on the command before the invalid parameter, then the message box is all that is displayed. If the `/silent` parameter appears 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.

53. `/noundo`Tells 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.

54. `/noundomanager`Tells 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.

55. `/nocache`can 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.

47. 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/powergrep50.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 Expressions Tutorial

This tutorial teaches you all you need to know to be able to craft powerful time-saving regular expressions. It starts with the most basic concepts, so that you can follow this tutorial even if you know nothing at all about regular expressions yet.

The tutorial doesn’t stop there. It also explains how a regular expression engine works on the inside, and alert you at the consequences. This helps you to quickly understand 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. You will usually find the name abbreviated to “regex" or "regexp". This tutorial uses "regex", because it is easy to pronounce the plural "regexes”. In this book, regular expressions are highlighted in red as `regex`.

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 highlighted in blue in this book.

`\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}\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 two or more letters. In other words: this pattern describes an email address. This also shows the syntax highlighting applied to regular expressions in this book. Word boundaries and quantifiers are blue, character classes are orange, and escaped literals are gray. You’ll see additional colors like green for grouping and purple for meta tokens later in the tutorial.

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. This tutorial uses the term “string” to indicate the text that the regular expression is applied to. This book highlights them in `green`. 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.

2. Literal Characters

The most basic regular expression consists of a single literal character, such as `a`. It matches the first occurrence of that character in the string. If the string is `Jack is a boy`, it matches 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 only does 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` matches `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 PowerGREP there are 12 characters with special meanings: 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 parenthesis `(`, the closing parenthesis `)`, the opening square bracket `[`, and the opening curly brace `{`, These special characters are often called "metacharacters". Most of them are errors when used alone.

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 has a special meaning.

Note that `1+1=2`, with the backslash omitted, is a valid regex. So you won't get an error message. But it doesn't 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.

PowerGREP treats the brace `{` as a literal character, unless it is part of a repetition operator like `a{1,3}`. So you generally do not need to escape it with a backslash, though you can do so if you want.

`]` is a literal outside character classes. Different rules apply inside character classes. Those are discussed in the topic about character classes.

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` is a shorthand that matches a single digit from `0` to `9`.

PowerGREP also supports 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`.

3. 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), and `\f` (form feed, 0x0C). Remember that Windows text files use `\r\n` to terminate lines, while UNIX text files use `\n`.

You can use `\uFFFF` or `\x{FFFF}` to insert a Unicode character. The euro currency sign occupies Unicode code point U+20AC. If you cannot type it on your keyboard, you can insert it into a regular expression with `\u20AC` or `\x{20AC}`. See the tutorial section on Unicode for more details on matching Unicode code points.

When working with files in 8-bit code pages in PowerGREP, you can include any character in your regular expression if you know its position in 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.

PowerGREP supports `\o{377}` for octal escapes. You can have any number of octal digits between the curly braces, with or without leading zero. There is no confusion with backreferences and literal digits that follow are cleanly separated by the closing curly brace. Do be careful to only put octal digits between the curly braces.

4. First Look at How a Regex Engine Works Internally

Knowing how the regex engineworks enables you to craft better regexes more easily. It helps you understand quickly why a particular regex does not do what you initially expected. This saves you lots of guesswork and head scratching when you need to write more complex regexes.

After introducing a new regex token, this tutorial explains 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 enables you to use its full power and help you avoid common mistakes.

While there are many implementations of regular expressions that differ sometimes slightly and sometimes significantly in syntax and behavior, there are basically only two kinds of regular expression engines: text-directed engines, and regex-directed engines. Nearly all modern regex flavors 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.

A regex-directed engine walks through the regex, attempting to match the next token in the regex to the next character. If a match is found, the engine advances through the regex and the subject string. If a token fails to match, the engine *backtracks* to a previous position in the regex and the subject string where it can try a different path through the regex. This tutorial will talk a lot more about backtracking later on. Modern regex flavors using regex-directed engines have lots of features such as atomic grouping and possessive quantifiers that allow you to control this backtracking.

A text-directed engine walks through the subject string, attempting all permutations of the regex before advancing to the next character in the string. A text-directed engine never backtracks. Thus, there isn't much to discuss about the matching process of a text-directed engine. In most cases, a text-directed engine finds the same matches as a regex-directed engine.

When this tutorial talks about regex engine internals, the discussion assumes a regex-directed engine. It only mentions text-directed engines in situations where they find different matches. And that only really happens when your regex uses alternation with two alternatives that can match at the same position.

The Regex Engine Always Returns the Leftmost Match

This is a very important point to understand: a regex engine always returns the leftmost match, even if a "better" match could be found later. When applying a regex to a string, the engine starts at the first character of the string. It tries all possible permutations of the regular expression at the first character. Only if all possibilities have been tried and found to fail, does the engine continue with the second character in the text. Again, it tries all possible permutations of the regex, in exactly the same order. The result is that the regex engine returns the *leftmost* match.

When applying `cat` to `He captured a catfish for his cat.`, the engine tries 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 string, `c` matches `c`. The engine then tries 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 string. So it continues with the 5th: `a`. Again, `c` fails to match here and the engine carries on.

At the 15th character in the string, `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 therefore reports 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. However, it is important that you can follow the steps the engine takes in your mind. In following examples, the way the engine works has a profound impact on the matches it finds. 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` does 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.

Character classes are one of the most commonly used features of regular expressions. You can find a word, even if it is misspelled, such as `sep[ae]r[ae]te` or `li[cs]en[cs]e`. You can find an identifier in a programming language with `[A-Za-z_][A-Za-z_0-9]*`. You can find a C-style hexadecimal number with `0[xX][A-Fa-f0-9]+`.

Negated Character Classes

Typing a caret after the opening square bracket negates the character class. The result is that the character class matches any character that is *not* in the character class. Unlike the dot, negated character classes also match (invisible) line break characters. If you don’t want a negated character class to match line breaks, you need to include the line break characters in the class. `[^0-9\r\n]` matches any character that is not a digit or a line break.

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 does not match the q in the string `Iraq`. It does match the q and the space after the q in `Iraq is a country`. Indeed: the space becomes 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

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.

To include an unescaped caret as a literal, place it anywhere except right after the opening bracket. `[x^]` matches an x or a caret.

You can include an unescaped closing bracket by placing it right after the opening bracket, or right after 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. `[^-x]` and `[^x-]` match any character thas is not an x or a hyphen.

Many regex tokens that work outside character classes can also be used inside character classes. This includes character escapes, octal escapes, and hexadecimal escapes for non-printable characters. It also includes Unicode character escapes and Unicode properties. `[$\u20AC]` matches a dollar or euro sign.

Repeating Character Classes

If you repeat a character class by using the `?`, `*` or `+` operators, you're repeating the entire character class. You're not repeating 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 need to use backreferences. `([0-9])\1+` matches `222` but not `837`. When applied to the string `833337`, it matches `3333` in the middle of this string. If you do not want that, you need to use lookaround.

Looking Inside The Regex Engine

As was mentioned earlier: the order of the characters inside a character class does not matter. `gr[ae]y` matches `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. Now we'll see 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 fails to match `g` at every step, and continues with the next character in the string. When the engine arrives at the 13th character, `g` is matched. The engine then tries 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 first attempts to match `a`, and fails.

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 continues with the other option, and finds 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 returns `grey` as the match result, and looks no further. Again, the *leftmost match* is 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. `gray` is only matched if you tell the regex engine to continue looking for a second match in the remainder of the subject string after the first match.

6. Character Class Subtraction

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, these flavors interpret the hyphen as the subtraction operator rather than the range operator. You can use the full character class syntax within the subtracted character class.

The character class `[a-z-[aeiuo]]` matches a single letter that is not a vowel. In other words: it matches a single consonant. Without character class subtraction or intersection, the only way to do this would be to list all consonants: `[b-df-hj-np-tv-z]`.

The character class `[\p{Nd}-[^\p{IsThai}]]` matches any single Thai digit. The base class matches any Unicode digit. All non-Thai characters are subtracted from that class. `[\p{Nd}-[\P{IsThai}]]` does the same. `[\p{IsThai}-[^\p{Nd}]]` and `[\p{IsThai}-[\P{Nd}]]` also match a single Thai digit by subtracting all non-digits from the Thai characters.

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. `[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 characters and Greek characters from a class with all Unicode letters, combine the ASCII and Greek characters into one class, and subtract that, as in `[\p{L}-[\p{IsBasicLatin}\p{IsGreek}]]`.

Negation Takes Precedence over Subtraction

The character class `[^1234-[3456]]` is both negated and subtracted from. In all flavors that support character class subtraction, the base class is negated before it is subtracted from. This class should be read as "(not 1234) minus 3456". Thus this character class matches any character other than the digits 1, 2, 3, 4, 5, and 6.

7. Character Class Intersection

Character class intersection makes it easy to match any single character that must be present in two sets of characters. The syntax for this is `[class&&[intersect]]`. You can use the full character class syntax within the intersected character class.

You cannot omit the nested square brackets in PowerGREP. If you do, PowerGREP interprets the ampersands as literals. So in PowerGREP `[class&&intersect]` is a character class containing only literals, just like `[clas&inter]`.

The character class `[a-z&&[^aeiuo]]` matches a single letter that is not a vowel. In other words: it matches a single consonant. Without character class subtraction or intersection, the only way to do this would be to list all consonants: `[b-df-hj-np-tv-z]`.

The character class `[\p{Nd}&&[\p{IsThai}]]` matches any single Thai digit. `[\p{IsThai}&&[\p{Nd}]]` does exactly the same.

Intersection of Multiple Classes

You can intersect the same class more than once. `[0-9&&[0-6&&[4-9]]]` is the same as `[4-6]` as those are the only digits present in all three parts of the intersection.

PowerGREP does not allow anything after the nested `]`. The characters `56` in `[0-9&&[12]56]` are an error.

You also shouldn't put && at the very start or very end of the regex. PowerGREP treats leading and trailing && as literal ampersands.

Intersection in Negated Classes

The character class `[^1234&&[3456]]` is both negated and intersected. In PowerGREP, negation takes precedence over intersection. PowerGREP reads this regex as "(not 1234) and 3456". Thus in PowerGREP this class is the same as `[56]` and matches the digits 5 and 6.

8. Shorthand Character Classes

PowerGREP supports the following shorthand character classes. The full character classes that they represent are based on Unicode categories.

Shorthand	Description	Full character class
`\d`	Digits	`\p{Nd}`
`\h`	Horizontal whitespace	`[\t\p{Zs}]`
`\l`	Lowercase letters	`\p{Ll}`
`\s`	All whitespace characters, including line breaks	`[\p{Z}\t\r\n\x0B\f]`
`\u`	Uppercase letters	`\p{Lu}`
`\v`	Vertical whitespace	`[\n\x0B\f\r\u0085\u2028\u2029]`
`\w`	Word characters (letters, numbers and underscores)	`[\p{L}\p{Nl}\p{Nd}\p{Pc}]`

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 matches any character that is neither a digit nor whitespace. It matches `x`, but not `8`. The former, however, matches any character that is either not a digit, or is not whitespace. Because all digits are not whitespace, and all whitespace characters are not digits, `[\D\S]` matches any character; digit, whitespace, or otherwise.

XML Character Classes

PowerGREP supports four more shorthands. `\i` matches any character that may be the first character of an XML name. `\c` matches any character that may occur after the first character in an XML name. `\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`.

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.

9. 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 line break characters. In all regex flavors discussed in this tutorial, the dot does *not* match line breaks by default.

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 line breaks, so the dot could never match them.

Line Break Characters

In PowerGREP the dot recognizes the newline `\n`, the carriage return `\r`, the form feed `\f`, the vertical tab `\x0B`, the Latin-1 next line control character `\u0085`, the Unicode line separator `\u2028` and the Unicode page separator `\u2029` as line break characters. The dot does not match any of these characters unless you turn on the “dot matches line breaks” option. The dot treats a CRLF pair as two separate characters.

\N Never Matches Line Breaks

`\N` matches any single character that is not a line break, just like the dot does. Unlike the dot, `\N` is not affected by the “dot matches line breaks” option. `(?s)\N.` turns on single-line mode and then matches any character that is not a line break followed by any character regardless of whether it is a line break.

Use The Dot Sparingly

The dot is a very powerful regex metacharacter. It allows you to be lazy. Put in a dot, and everything matches just fine when you test the regex on valid data. The problem is that the regex also matches 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.

Let’s illustrate this with a simple example. 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 matches 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. `[01]\d[- /.][0-3]\d[- /.]\d\d` is a step ahead, though it still matches `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 Classes Instead of the Dot

A negated character class is often more appropriate than the dot. The tutorial section that explains the repeat operators star and plus covers this in more detail. But the warning is important enough to mention it here as well. Again let's 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 matches `"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 do the same with a negated character class. 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]*"`.

10. Start of String and End of String Anchors

Thus far, we have learned about literal characters, character classes, and the dot. Putting one of these in a regex tells 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` does 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.

A regex that consists solely of an anchor can only find zero-length matches. This can be useful, but can also create complications that are explained near the end of this tutorial.

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, most regex engines 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, `$` still matches at the end of the string (after the last `e`), and also before every line break (between `e` and `\n`).

In PowerGREP, the caret and dollar always match at the start and end of each line. This makes sense because PowerGREP is designed to work with entire files, rather than short strings.

Line Break Characters

Just like the dot, anchors in PowerGREP recognize the newline `\n`, the carriage return `\r`, the form feed `\f`, the vertical tab `\x0B`, the Latin-1 next line control character `\u0085`, the Unicode line separator `\u2028` and the Unicode page separator `\u2029` as line break characters. In addition, the anchors treat CRLF as an indivisible pair. `^` matches after CRLF and `$` matches before CRLF, but neither match in the middle of a CRLF pair.

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.

Strings Ending with a Line Break

Because Perl returns a string with a newline at the end when reading a line from a file, Perl's regex engine matches `$` at the position before the line break at the end of the string even when multi-line mode is turned off. Perl also matches `$` at the very end of the string, regardless of whether that character is a line break. So `^\d+$` matches `123` whether the subject string is `123` or `123\n`.

Most modern regex flavors have copied this behavior. That includes PowerGREP.

`\Z` also matches before the final line break. 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\d+\z` does not match `123\n`. `\z` matches after the line break, which is not matched by the shorthand character class.

Strings Ending with Multiple Line Breaks

If a string ends with multiple line breaks and multi-line mode is off then `$` only matches before the last of those line breaks in all flavors where it can match before the final break. The same is true for `\Z` regardless of multi-line mode.

Boost is the only exception. In Boost, `\Z` can match before any number of trailing line breaks as well as at the very end of the string. So if the subject string ends with three line breaks, Boost's `\Z` has four positions that it can match at. Like in all other flavors, Boost's `\Z` is independent of multi-line mode. Boost's `$` only matches at the very end of the string when you turn off multi-line mode (which is on by default in Boost).

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-length 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-length, 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 `$`, and the current character is advanced to the very last position in the string: the void after the string. No regex token that needs a character to match can match here. Not even a negated character class. However, we are trying to match a dollar sign, and the mighty dollar is a strange beast. It is zero-length, so it tries to match the position before the current character. It does not matter that this "character" is the void after the string. In fact, the dollar checks the current character. It must be either a newline, or the void after the string, for `$` 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.

11. 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 PowerGREP, characters that are matched by the short-hand character class \w are the characters that are treated as word characters by word boundaries.

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 does 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".

\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-length. 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 `is`, it would have matched the `is` in `This`.

12. 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 need to use parentheses 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 parentheses then 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 stops 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` is attempted before `Set`, and the engine matches 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` is 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`.

13. Optional Items

The question mark makes the preceding token in the regular expression optional. `colou?r` matches both `colour` and `color`. The question mark is called a quantifier.

You can make several tokens optional by grouping them together using parentheses, and placing the question mark after the closing parenthesis. E.g.: `Nov(ember)?` matches `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`.

You can also use curly braces to make something optional. `colou{0,1}r` is the same as `colou?r`.

Important Regex Concept: Greediness

The question mark is the first metacharacter introduced by this tutorial 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 always tries 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 is always `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.

The discussion about the other repetition operators has more details on greedy and lazy quantifiers.

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 `colonel`. 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 skips 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 `colonel`. Therefore, the engine starts again trying to match `c` to the first o in `colonel`.

After a series of failures, `c` matches 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.

14. Repetition with Star and Plus

One repetition operator or quantifier was already introduced: 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 angle 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 `<B>`. 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

There's an additional quantifier that allows you to specify how many times a token can be repeated. The syntax is `{min,max}`, where *min* is zero or 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 `{0,1}` is the same as `?`, `{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 <EM>first</EM> test`. You might expect the regex to match `<EM>` and when continuing after that match, `</EM>`.

But it does not. The regex will match `<EM>first</EM>`. 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 `<EM>first</EM> 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 `EM>first</EM> 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 `<EM>first</EM> te`. But `>` still cannot match. So the engine continues backtracking until the match of `.+` is reduced to `EM>first</EM`. Now, `>` can match the next character in the string. The last token in the regex has been matched. The engine reports that `<EM>first</EM>` 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 markafter 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 `EM`, 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 `<EM>` 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.

Only regex-directed engines backtrack. Text-directed engines don't and thus do not get the speed penalty. But they also do not support lazy quantifiers.

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.

15. Use Parentheses for Grouping and Capturing

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 quantifier to the entire group or to restrict alternation to part of the regex.

Only parentheses can be used for grouping. Square brackets define a character class, and curly braces are used by a quantifier with specific limits.

Parentheses Create Numbered Capturing Groups

Besides grouping part of a regular expression together, parentheses also create a numbered capturing group. It stores the part of the string matched by the part of the regular expression inside the parentheses.

The regex `Set(Value)?` matches `Set` or `SetValue`. In the first case, the first (and only) capturing group remains empty. In the second case, the first capturing group matches `Value`.

Non-Capturing Groups

If you do not need the group to capture its match, you can optimize this regular expression into `Set(?:Value)?`. The question mark and the colon after the opening parenthesis are the syntax that creates a non-capturing group. The question mark after the opening bracket is unrelated to the question mark at the end of the regex. The final question mark is the quantifier that makes the previous token optional. This quantifier cannot appear after an opening parenthesis, because there is nothing to be made optional at the start of a group. Therefore, there is no ambiguity between the question mark as an operator to make a token optional and the question mark as part of the syntax for non-capturing groups, even though this may be confusing at first. There are other kinds of groups that use the `(?` syntax in combination with other characters than the colon that are explained later in this tutorial.

`color=(?:red|green|blue)` is another regex with a non-capturing group. This regex has no quantifiers.

Using Text Matched By Capturing Groups

Capturing groups make it easy to extract part of the regex match. You can reuse the text inside the regular expression via a backreference. Backreferences can also be used in replacement strings. Please check the replacement text tutorial for details.

16. Using Backreferences To Match The Same Text Again

Backreferences match the same text as previously matched by a capturing group. 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]*`. This is the opening HTML tag. (Since HTML tags are case insensitive, this regex requires case insensitive matching.) The backreference `\1` (backslash one) references the first capturing group. `\1` matches the exact same text that was matched by the first capturing group. The `/` before it is a literal character. 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. Count the opening parentheses of all the numbered capturing groups. The first parenthesis starts backreference number one, the second number two, etc. Skip parentheses that are part of other syntax such as non-capturing groups. This 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` matches `axaxa`, `bxbxb` and `cxcxc`.

Most regex flavors support up to 99 capturing groups and double-digit backreferences. So `\99` is a valid backreference if your regex has 99 capturing groups.

Looking Inside The Regex Engine

Let's see how the regex engine applies the regex `<([A-Z][A-Z0-9]*)\b[^>]*>.*?</\1>` to the string `Testing <B><I>bold italic</I></B> text`. The first token in the regex is the literal `<`. The regex engine traverses 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 word boundary `\b` matches at the `>` because it is preceded by `B`. The word boundary does not make the engine advance through the string. 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 initially skips 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 reads 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`. `\1` matches `B`. The last token in the regex, `>` matches `>`. A complete match has been found: `<B><I>bold italic</I></B>`.

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</b>`. 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]*)[^>]*>.*?</\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 fails to match, quickly ending the match attempt.

If you don't want the regex engine to backtrack into capturing groups, you can use an atomic group. The tutorial section on atomic grouping has all the details.

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

Parentheses cannot be used inside character classes, at least not as metacharacters. When you put a parenthesis in a character class, it is treated as a literal character. So the regex `[(a)b]` matches `a`, `b`, `(`, and `)`.

Backreferences, too, cannot be used inside a character class. The \1 in a regex like `(a)[\1b]` is a needlessly escaped literal 1.

17. Backreferences to Failed Groups

The previous topic on backreferences applies to all regex flavors, except those few that don't support backreferences 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` matches `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` fails 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`. 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.

Backreferences to Non-Existent Capturing Groups

Backreferences to groups that do not exist, such as `(one)\7`, are an error. A backslash followed by two digits forms a single-digit backreference followed by a literal digit if there are fewer capturing groups than the two-digit number. So `(one)\12` matches `oneone2`.

Forward References

PowerGREP allows you to 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 fails like a backreference to a failed group does.

The regex `(\2two|(one))+` matches `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.

Nested References

A nested reference is a backreference inside the capturing group that it references. Like forward references, nested references are only useful if they're inside a repeated group, as in `(\1two|(one))+`. When nested references are supported, this regex also matches `oneonetwo`. At the start of the string, `\1` 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, `\1` matches `one` as captured by the last iteration of the first group. It doesn't matter that the regex engine has re-entered the first group. The text matched by the group was stored into the backreference when the group was previously exited. `two` then matches `two`. With two repetitions of the first group, the regex has matched the whole subject string. If you retrieve the text from the capturing groups after the match, the first group stores `onetwo` while the second group captured the first occurrence of `one` in the string.

18. Named Capturing Groups and Backreferences

PowerGREP supports numbered capturing groups and numbered backreferences. Long regular expressions with lots of groups and backreferences may be hard to read. They can be particularly difficult to maintain as adding or removing a capturing group in the middle of the regex upsets the numbers of all the groups that follow the added or removed group.

Python's re module was the first to offer a solution: named capturing groups and named backreferences. `(?P<name>group)` captures the match of `group` into the backreference "name". `name` must be an alphanumeric sequence starting with a letter. `group` can be any regular expression. You can reference the contents of the group with the named backreference `(?P=name)`. The question mark, P, angle brackets, and equals signs are all part of the syntax. Though the syntax for the named backreference uses parentheses, it's just a backreference that doesn't do any capturing or grouping. The HTML tags example can be written as `<(?P<tag>[A-Z][A-Z0-9]*)\b[^>]*>.*?</(?P=tag)>`.

The .NET framework also supports named capture. Microsoft's developers invented their own syntax, rather than follow the one pioneered by Python and copied by PCRE (the only two regex engines that supported named capture at that time). `(?<name>group)` or `(?'name'group)` captures the match of `group` into the backreference "name". The named backreference is `\k<name>` or `\k'name'`. Compared with Python, there is no P in the syntax for named groups. The syntax for named backreferences is more similar to that of numbered backreferences than to what Python uses. You can use single quotes or angle brackets around the name. This makes absolutely no difference in the regex. You can use both styles interchangeably. The syntax using angle brackets is preferable in programming languages that use single quotes to delimit strings, while the syntax using single quotes is preferable when adding your regex to an XML file, as this minimizes the amount of escaping you have to do to format your regex as a literal string or as XML content.

Because Python and .NET introduced their own syntax, we refer to these two variants as the "Python syntax" and the ".NET syntax" for named capture and named backreferences. PowerGREP supports both.

Numbers for Named Capturing Groups

Mixing named and numbered capturing groups is not recommended because flavors are inconsistent in how the groups are numbered. If a group doesn't need to have a name, make it non-capturing using the `(?:group)` syntax. You can make all unnamed groups non-capturing by putting the `(?n)` mode modifier at the start of your regex. If you make all unnamed groups non-capturing, you can skip this section and save yourself a headache.

Most flavors number both named and unnamed capturing groups by counting their opening parentheses from left to right. Adding a named capturing group to an existing regex still upsets the numbers of the unnamed groups. In .NET, however, unnamed capturing groups are assigned numbers first, counting their opening parentheses from left to right, skipping all named groups. After that, named groups are assigned the numbers that follow by counting the opening parentheses of the named groups from left to right.

PowerGREP copied the Python and the .NET syntax at a time when only Python and PCRE used the Python syntax, and only .NET used the .NET syntax. Therefore it also copied the numbering behavior of both Python and .NET, so that regexes intended for Python and .NET would keep their behavior. It numbers Python-style named groups along unnamed ones, like Python does. It numbers .NET-style named groups afterward, like .NET does. These rules apply even when you mix both styles in the same regex.

As an example, 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` or `$1$2$3$4`, you will get `abcd`. All four groups were numbered from left to right, from one till four.

Things are 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`. First, the unnamed groups `(a)` and `(c)` got the numbers 1 and 2. Then the named groups "x" and "y" got the numbers 3 and 4.

In all other flavors that copied the .NET syntax the regex `(a)(?<x>b)(c)(?<y>d)` still matches `abcd`. But in all those flavors, except the PowerGREP flavor, the replacement `\1\2\3\4` or `$1$2$3$4` gets you `abcd`. All four groups were numbered from left to right.

In PowerGREP named capturing groups play a special role. 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.

Multiple Groups with The Same Name

PowerGREP allows multiple groups in the regular expression to have the same name. All groups with the same name share the same storage for the text they match. Thus, a backreference to that name matches the text that was matched by the group with that name that most recently captured something. A reference to the name in the replacement text inserts the text matched by the group with that name that was the last one to capture something.

19. Branch Reset Groups

Alternatives inside a branch reset group share the same capturing groups. The syntax is `(?|regex)` where `(?|` opens the group and `regex` is any regular expression. If you don't use any alternation or capturing groups inside the branch reset group, then its special function doesn't come into play. It then acts as a non-capturing group.

The regex `(?|(a)|(b)|(c))` consists of a single branch reset group with three alternatives. This regex matches either `a`, `b`, or `c`. The regex has only a single capturing group with number 1 that is shared by all three alternatives. After the match, `$1` holds `a`, `b`, or `c`.

Compare this with the regex `(a)|(b)|(c)` that lacks the branch reset group. This regex also matches `a`, `b`, or `c`. But it has three capturing groups. After the match, `$1` holds `a` or nothing at all, `$2` holds `b` or nothing at all, while `$3` holds `c` or nothing at all.

Backreferences to capturing groups inside branch reset groups work like you'd expect. `(?|(a)|(b)|(c))\1` matches `aa`, `bb`, or `cc`. Since only one of the alternatives inside the branch reset group can match, the alternative that participates in the match determines the text stored by the capturing group and thus the text matched by the backreference.

The alternatives in the branch reset group don't need to have the same number of capturing groups. `(?|abc|(d)(e)(f)|g(h)i)` has three capturing groups. When this regex matches `abc`, all three groups are empty. When `def` is matched, `$1` holds `d`, `$2` holds `e` and `$3` holds `f`. When `ghi` is matched, `$1` holds `h` while the other two are empty.

You can have capturing groups before and after the branch reset group. Groups before the branch reset group are numbered as usual. Groups in the branch reset group are numbered continued from the groups before the branch reset group, which each alternative resetting the number. Groups after the branch reset group are numbered continued from the alternative with the most groups, even if that is not the last alternative. So `(x)(?|abc|(d)(e)(f)|g(h)i)(y)` defines five capturing groups. `(x)` is group 1, `(d)` and `(h)` are group 2, `(e)` is group 3, `(f)` is group 4, and `(y)` is group 5.

Named Capturing Groups in Branch Reset Groups

You can use named capturing groups inside branch reset groups. You have to use the same group names in the same order in each alternative. Mismatched group names are an error.

`(?'before'x)(?|abc|(?'left'd)(?'middle'e)(?'right'f)|g(?'left'h)i)(?'after'y)` is the same as the previous regex. It names the five groups "before", "left", "middle", "right", and "after". Notice that because the 3rd alternative has only one capturing group, that must be the name of the first group in the other alternatives.

If you omit the names in some alternatives, the groups will still share the names with the other alternatives. In the regex `(?'before'x)(?|abc|(?'left'd)(?'middle'e)(?'right'f)|g(h)i)(?'after'y)` the group `(h)` is still named "left" because the branch reset group makes it share the name and number of `(?'left'd)`.

In PowerGREP, groups with the same name are always treated as one and the same group. So you don't really need to use a branch reset group in PowerGREP when using named capturing groups.

Day and Month with Accurate Number of Days

It's time for a more practical example. These two regular expressions match a date in m/d or mm/dd format. They exclude invalid dates such as 2/31.

```
^(?:(0?[13578]|1[02])/(3[01]|[12][0-9]|0?[1-9]) # 31 days
 |  (0?[469]|11)/(30|[12][0-9]|0?[1-9])          # 30 days
 |  (0?2)/([12][0-9]|0?[1-9])                    # 29 days
 )$
```

The first version uses a non-capturing group `(?:…)` to group the alternatives. It has six separate capturing groups. `$1` and `$2` hold the month and the day for months with 31 days. `$3` and `$4` hold them for months with 30 days. `$5` and `$6` are only used for February.

```
^(?|(0?[13578]|1[02])/(3[01]|[12][0-9]|0?[1-9]) # 31 days
 |  (0?[469]|11)/(30|[12][0-9]|0?[1-9])          # 30 days
 |  (0?2)/([12][0-9]|0?[1-9])                    # 29 days
 )$
```

The second version uses a branch reset group `(?|…)` to group the alternatives and merge their capturing groups. The 4th character is the only difference between these two regexes. Now there are only two capturing groups. These are shared between the tree alternatives. When a match is found `$1` always holds the month and 2 always holds the day, regardless of the number of days in the month.

20. Free-Spacing Regular Expressions

Most modern regex flavors support a variant of the regular expression syntax called free-spacing mode. This mode allows for regular expressions that are much easier for people to read. Of the flavors discussed in this tutorial, only XML Schema and the POSIX and GNU flavors don't support it. Plain JavaScript doesn't either, but XRegExp does. The mode is usually enabled by setting an option or flag outside the regex. With flavors that support mode modifiers, you can put `(?x)` the very start of the regex to make the remainder of the regex free-spacing.

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. `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)`. The latter is 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 lookaround, named groups, etc.

PowerGREP ignores all Unicode spaces and line breaks in free-spacing mode. Most other regex flavors, even if they normally support Unicode, only ignore the ASCII space, tab, line feed, carriage return, and form feed characters.

Free-Spacing in Character Classes

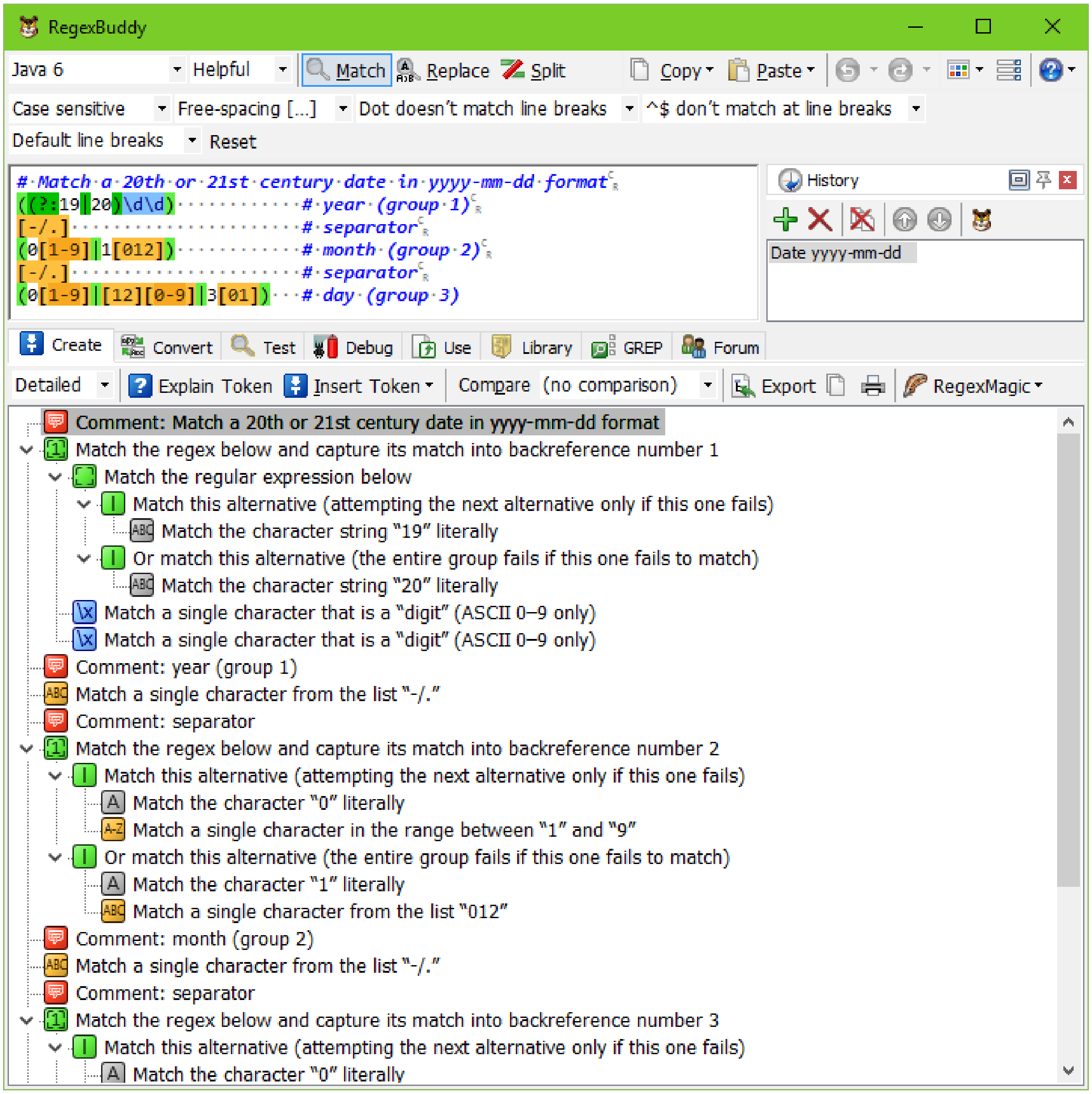
A character class is also treated as a single token. `[abc]` is not the same as `[ a b c ]`. The former matches one of three letters, while the latter matches those three letters or a space. In other words: free-spacing mode has no effect inside character classes. Spaces and line breaks inside character 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. The hexadecimal escape `\x20` also works, of course.

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 line break character is ignored.

Putting it all together, the regex to match a valid date can be clarified by writing it across multiple lines:

```
# 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]|[12][0-9]|3[01])   # day (group 3)
```



```
# 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]|[12][0-9]|3[01])   # day (group 3)
```

```
Comment: Match a 20th or 21st century date in yyyy-mm-dd format
Match the regex below and capture its match into backreference number 1
    Match the regular expression below
        Match this alternative (attempting the next alternative only if this one fails)
            Match the character string "19" literally
        Or match this alternative (the entire group fails if this one fails to match)
            Match the character string "20" literally
    Match a single character that is a "digit" (ASCII 0–9 only)
    Match a single character that is a "digit" (ASCII 0–9 only)
Comment: year (group 1)
Match a single character from the list "-/."
Comment: separator
Match the regex below and capture its match into backreference number 2
    Match this alternative (attempting the next alternative only if this one fails)
        Match the character "0" literally
        Match a single character in the range between "1" and "9"
    Or match this alternative (the entire group fails if this one fails to match)
        Match the character "1" literally
        Match a single character from the list "012"
Comment: month (group 2)
Match a single character from the list "-/."
Comment: separator
Match the regex below and capture its match into backreference number 3
    Match this alternative (attempting the next alternative only if this one fails)
        Match the character "0" literally
```

Comments Without Free-Spacing

PowerGREP also allows you to add comments to your regex without using free-spacing mode. The syntax is `(?#comment)` where "comment" can be whatever you want, as long as it does not contain a closing parenthesis. The regex engine ignores everything after the `(?#` until the first closing parenthesis.

21. 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.

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".

PowerGREP treats 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: simply use `\X`. You can consider `\X` the Unicode version of the dot. 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.

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.

Unicode Categories

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 the "letter" category with `\p{L}`. You can match a single character *not* belonging to that category with `\P{L}`.

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".

In addition to the standard notation, PowerGREP allows 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 `Al` or `àl` or any Unicode letter followed by a literal `l`.

PowerGREP also supports 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{Cased_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 or Latin-1 control character: 0x00–0x1F and 0x7F–0x9F.
  - `\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.

Here’s a list:

1. `\p{Common}`
2. `\p{Arabic}`
3. `\p{Armenian}`
4. `\p{Bengali}`
5. `\p{Bopomofo}`
6. `\p{Braille}`
7. `\p{Buhid}`
8. `\p{Canadian_Aboriginal}`
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}`

PowerGREP allows you to use `\p{IsLatin}` instead of `\p{Latin}`. The "Is" syntax is useful for distinguishing between scripts and blocks, as explained in the next section.

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.

For example, the Currency block does not include the dollar and yen symbols. Those are found in the Basic_Latin and Latin-1_Supplement blocks instead, even though both are currency symbols, and the yen symbol is not a Latin character. This is for historical reasons, because the ASCII standard includes the dollar sign, and the ISO-8859 standard includes the yen sign. 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. The Unicode property `\p{Sc}` or `\p{Currency_Symbol}` would be a better choice than the Unicode block `\p{InCurrency_Symbols}` 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

22. Specifying Modes Inside The Regular Expression

Normally, matching modes are specified outside the regular expression. In a programming language, you pass them as a flag to the regex constructor or append them to the regex literal. In an application, you'd toggle the appropriate buttons or checkboxes. You can find the specifics in the tools and languages section of this book.

Sometimes, the tool or language does not provide the ability to specify matching options. The handy `String.matches()` method in Java does not take a parameter for matching options like `Pattern.compile()` does. Or, the regex flavor may support matching modes that aren't exposed as external flags. The regex functions in R have `ignore.case` as their only option, even though the underlying PCRE library has more matching modes than any other discussed in this tutorial.

In those situation, you can add the following mode modifiers to the start of the regex. To specify multiple modes, simply put them together as in `(?ismx)`.

- `(?i)` makes the regex case insensitive.
- `(?x)` turn on free-spacing mode.
- `(?s)` for "single line mode" makes the dot match all characters, including line breaks.
- `(?m)` for "multi-line mode" makes the caret and dollar match at the start and end of each line in the subject string.
- `(?n)` turns all unnamed groups into non-capturing groups.

Turning Modes On and Off for Only Part of The Regular Expression

PowerGREP allows you to apply modifiers to only part of the regular expression. If you insert the modifier `(?ism)` in the middle of the regex then 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.

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)caseless(?-i)cased(?i)caseless` is equivalent to `(?i)caseless(?-i:cased)caseless`. This syntax resembles that of the non-capturing group `(?:group)`. You could think of a non-capturing group as a modifier span that does not change any modifiers. Like a non-capturing group, the modifier span does not create a backreference.

23. 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 PowerGREP. PowerGREP also supports 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.

24. Possessive Quantifiers

The topic on repetition operators or quantifiers explains 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 first tries to repeat the token as many times as possible, and gradually gives up matches as the engine backtracks to find an overall match. A lazy quantifier first repeats the token as few times as required, and gradually expands 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

Like a greedy quantifier, a possessive quantifier repeats the token as many times as possible. Unlike a greedy quantifier, it does *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. `*` 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 happens 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 `"[^"]*"` with 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 possibly have 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 PowerGREP's 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. For example, `".*"` matches `"abc"` in `"abc"x`, but `".*+"` does not match this string at all.

In both regular expressions, the first `"` matches 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"` is 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.

25. Lookahead and Lookbehind Zero-Length Assertions

Lookahead and lookbehind, collectively called "lookaround", are zero-length assertions just like the start and end of line, and start and end of word anchors explained earlier in this tutorial. The difference is that lookaround actually matches characters, but then gives up the match, returning only 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. Lookaround allows 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, this tutorial 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 parentheses, with the opening parenthesis 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 parentheses, with the opening parenthesis followed by a question mark and an equals sign.

You can use any regular expression inside the lookahead (but not lookbehind, as explained below). Any valid regular expression can be used inside the lookahead. If it contains capturing groups then those groups will capture as normal and backreferences to them will work normally, even outside the lookahead. The lookahead itself is not a capturing group. It is not included in the count towards numbering the backreferences. If you want to store the match of the regex inside a lookahead, 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 capturing group is to store its match.

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 causes the engine to traverse the string until the `q` in the string is matched. The position in the string is now the void after 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 after 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`. The lookahead is now positive and is followed by another token. 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 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 doesn't match `cab`, but matches 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 parentheses, with the opening parenthesis 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 `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-length, 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

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 matches `John` and the latter matches `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 also doesn't

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.

PowerGREP allows you to use a full regular expression inside lookbehind, including infinite repetition and backreferences. PowerGREP really applies the regex inside the lookbehind backwards, going through the regex inside the lookbehind and through the subject string from right to left. PowerGREP only needs to evaluate the lookbehind once, regardless of how many different possible lengths it has.

Lookaround Is Atomic

The fact that lookaround is zero-length 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` never matches `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` does match `56x56` in `456x56`.

If you don’t use capturing groups inside lookaround, then all this doesn’t matter. Either the lookaround condition can be satisfied or it cannot be. In how many ways it can be satisfied is irrelevant.

26. Testing The Same Part of a String for More Than One Requirement

Lookaround, which was 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-length. 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 traverses part of the string twice.

A more practical example makes this clear. 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 first attempts 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 fails and the engine continues trying the regex from the start at the next character position in the string.

The lookahead is zero-length. 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. The regex engine attempts the remainder of the regex at this position. 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*` matches 6 times. The engine then backtracks, 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*` consumes 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 we did above. The third and last `\b` are guaranteed to match. Since word boundaries are zero-length, and therefore do 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-length 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.

27. Keep The Text Matched So Far out of The Overall Regex Match

`\K` keeps the text matched so far out of the overall regex match. `h\Kd` matches only the second `d` in `adhd`.

This feature is not really needed in PowerGREP because it supports unrestricted lookbehind, which is much more flexible than `\K`. Still, PowerGREP supports `\K` if you prefer this way of working.

Looking Inside The Regex Engine

Let's see how `h\Kd` works. The engine begins the match attempt at the start of the string. `h` fails to match `a`. There are no further alternatives to try. The match attempt at the start of the string has failed.

The engine advances one character through the string and attempts the match again. `h` fails to match `d`.

Advancing again, `h` matches `h`. The engine advances through the regex. The regex has now reached `\K` in the regex and the position between `h` and the second `d` in the string. `\K` does nothing other than to tell that if this match attempt ends up succeeding, the regex engine should pretend that the match attempt started at the present position between `h` and `d`, rather than between the first `d` and `h` where it really started.

The engine advances through the regex. `d` matches the second `d` in the string. An overall match is found. Because of the position saved by `\K`, the second `d` in the string is returned as the overall match.

`\K` only affects the position returned after a successful match. It does not move the start of the match attempt during the matching process. The regex `hhh\Kd` matches the `d` in `hhhhd`. This regex first matches `hhh` at the start of the string. Then `\K` notes the position between `hhh` and `hd` in the string. Then `d` fails to match the fourth `h` in the string. The match attempt at the start of the string has failed.

Now the engine must advance one character in the string before starting the next match attempt. It advances from the actual start of the match attempt, which was at the start of the string. The position stored by `\K` does not change this. So the second match attempt begins at the position after the first `h` in the string. Starting there, `hhh` matches `hhh`, `\K` notes the position, and `d` matches `d`. Now, the position remembered by `\K` is taken into account, and `d` is returned as the overall match.

\K Can Be Used Anywhere

You can use `\K` pretty much anywhere in any regular expression. You should only avoid using it inside lookbehind. You can use it inside groups, even when they have quantifiers. You can have as many instances of `\K` in your regex as you like. `(ab\Kc|d\Ke)f` matches `cf` when preceded by `ab`. It also matches `ef` when preceded by `d`.

`\K` does not affect capturing groups. When `(ab\Kc|d\Ke)f` matches `cf`, the capturing group captures `abc` as if the `\K` weren't there. When the regex matches `ef`, the capturing group stores `de`.

Limitations of \K

Lookbehind really goes backwards through the string. This allows lookbehind check for a match before the start of the match attempt. When the match attempt was started at the end of the previous match, lookbehind can match text that was part of the previous match. `\K` cannot do this, because it does not affect the way the regex engine goes through the matching process.

If you iterate over all matches of `(?<=a)a` in the string `aaaa`, you will get three matches: the second, third, and fourth `a` in the string. The first match attempt begins at the start of the string and fails because the lookbehind fails. The second match attempt begins between the first and second `a`, where the lookbehind succeeds and the second `a` is matched. The third match attempt begins after the second `a` that was just matched. Here the lookbehind succeeds too. It doesn't matter that the preceding `a` was part of the previous match. Thus the third match attempt matches the third `a`. Similarly, the fourth match attempt matches the fourth `a`. The fifth match attempt starts at the end of the string. The lookbehind still succeeds, but there are no characters left for `a` to match. The match attempt fails. The engine has reached the end of the string and the iteration stops. Five match attempts have found three matches.

Things are different when you iterate over `a\Ka` in the string `aaaa`. You will get only two matches: the second and the fourth `a`. The first match attempt begins at the start of the string. The first `a` in the regex matches the first `a` in the string. `\K` notes the position. The second `a` matches the second `a` in the string, which is returned as the first match. The second match attempt begins after the second `a` that was just matched. The first `a` in the regex matches the third `a` in the string. `\K` notes the position. The second `a` matches the fourth `a` in the string, which is returned as the first match. The third match attempt begins at the end of the string. `a` fails. The engine has reached the end of the string and the iteration stops. Three match attempts have found two matches.

Basically, you'll run into this issue when the part of the regex before the `\K` can match the same text as the part of the regex after the `\K`. If those parts can't match the same text, then a regex using `\K` will find the same matches than the same regex rewritten using lookbehind. In that case, you should use `\K` instead of lookbehind as that will give you better performance in Perl, PCRE, and Ruby.

Another limitation is that while lookbehind comes in positive and negative variants, `\K` does not provide a way to negate anything. `(?<!a)b` matches the string `b` entirely, because it is a "b" not preceded by an "a". `[^a]\Kb` does not match the string `b` at all. When attempting the match, `[^a]` matches `b`. The regex has now reached the end of the string. `\K` notes this position. But now there is nothing left for `b` to match. The match attempt fails. `[^a]\Kb` is the same as `(?<=[^a])b`, which are both different from `(?<!a)b`.

28. 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 parentheses. 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 parentheses, 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 parentheses 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)` consists of the optional capturing group `(a)?`, the literal `b`, and the conditional `(?(1)c|d)` that tests the capturing group. This regex 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 restarts 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 fails to match before the second and third characters in the string.

Named and Relative Conditionals

You can use the name of a capturing group instead of its number as the *if* test. The syntax is slightly inconsistent between regex flavors. Simply specify the name of the group between parentheses. `(?<test>a)?b(?(test)c|d)` is the regex from the previous section using named capture.

PowerGREP also supports relative conditionals. The syntax is the same as that of a conditional that references a numbered capturing group with an added plus or minus sign before the group number. The conditional then counts the opening parentheses to the left (minus) or to the right (plus) starting at the `(?(` that opens the conditional. `(a)?b(?(-1)c|d)` is another way of writing the above regex. The benefit is that this regex won't break if you add capturing groups at the start or the end of the regex.

Conditionals Referencing Non-Existent Capturing Groups

Conditionals that reference a non-existent capturing group are not an error. They always attempt the "else" part.

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 parentheses 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]+|(?(3)mm/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 of 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.

29. Matching Nested Constructs with Balancing Groups

PowerGREP has a special feature called balancing groups. The main purpose of balancing groups is to match balanced constructs or nested constructs, which is where they get their name from. A technically more accurate name for the feature would be capturing group subtraction. That's what the feature really does. This feature was first invented by the .NET regex flavor, which does not support regular expression recursion.

`(?<capture-subtract>regex)` or `(?'capture-subtract'regex)` is the basic syntax of a balancing group. It's the same syntax used for named capturing groups but with two group names delimited by a minus sign. The name of this group is "capture". You can omit the name of the group. `(?<-subtract>regex)` or `(?'-subtract'regex)` is the syntax for a non-capturing balancing group.

The name "subtract" must be the name of another group in the regex. When the regex engine enters the balancing group, it subtracts one match from the group "subtract". If the group "subtract" did not match yet, or if all its matches were already subtracted, then the balancing group fails to match. You could think of a balancing group as a conditional that tests the group "subtract", with "regex" as the "if" part and an "else" part that always fails to match. The difference is that the balancing group has the added feature of subtracting one match from the group "subtract", while a conditional leaves the group untouched.

If the balancing group succeeds and it has a name ("capture" in this example), then the group captures the text between the end of the match that was subtracted from the group "subtract" and the start of the match of the balancing group itself ("regex" in this example).

Looking Inside The Regex Engine

Let's apply the regex `(?'open'o)+(?'between-open'c)+` to the string `ooccc`. `(?'open'o)` matches the first `o` and stores that as the first capture of the group "open". The quantifier `+` repeats the group. `(?'open'o)` matches the second `o` and stores that as the second capture. Repeating again, `(?'open'o)` fails to match the first `c`. But the `+` is satisfied with two repetitions.

The regex engine advances to `(?'between-open'c)`. Before the engine can enter this balancing group, it must check whether the subtracted group "open" has captured something. It has captured the second `o`. The engine enters the group, subtracting the most recent capture from "open". This leaves the group "open" with the first `o` as its only capture. Now inside the balancing group, `c` matches `c`. The engine exits the balancing group. The group "between" captures the text between the match subtracted from "open" (the second `o`) and the `c` just matched by the balancing group. This is an empty string but it is captured anyway.

The balancing group too has `+` as its quantifier. The engine again finds that the subtracted group "open" captured something, namely the first `o`. The regex enters the balancing group, leaving the group "open" without any matches. `c` matches the second `c` in the string. The group "between" captures `oc` which is the text between the match subtracted from "open" (the first `o`) and the second `c` just matched by the balancing group.

The balancing group is repeated again. But this time, the regex engine finds that the group "open" has no matches left. The balancing group fails to match. The group "between" is unaffected, retaining its most recent capture.

The `+` is satisfied with two iterations. The engine has reached the end of the regex. It returns `oocc` as the overall match. `Match.Groups['open'].Success` will return `false`, because all the captures of that group were subtracted. `Match.Groups['between'].Value` returns `"oc"`.

Matching Balanced Pairs

We need to modify this regex if we want it to match a balanced number of o's and c's. To make sure that the regex won't match `ooccc`, which has more c's than o's, we can add anchors: `^(?'open'o)+(?'-open'c)+$`. This regex goes through the same matching process as the previous one. But after `(?'-open'c)+` fails to match its third iteration, the engine reaches `$` instead of the end of the regex. This fails to match. The regex engine will backtrack trying different permutations of the quantifiers, but they will all fail to match. No match can be found.

But the regex `^(?'open'o)+(?'-open'c)+$` still matches `ooc`. The matching process is again the same until the balancing group has matched the first `c` and left the group 'open' with the first `o` as its only capture. The quantifier makes the engine attempt the balancing group again. The engine again finds that the subtracted group "open" captured something. The regex enters the balancing group, leaving the group "open" without any matches. But now, `c` fails to match because the regex engine has reached the end of the string.

The regex engine must now backtrack out of the balancing group. When backtracking a balancing group, PowerGREP also backtracks the subtraction. Since the capture of the the first `o` was subtracted from "open" when entering the balancing group, this capture is now restored while backtracking out of the balancing group. The repeated group `(?'-open'c)+` is now reduced to a single iteration. But the quantifier is fine with that, as `+` means "once or more" as it always does. Still at the end of the string, the regex engine reaches `$` in the regex, which matches. The whole string `ooc` is returned as the overall match. `Match.Groups['open'].Captures` will hold the first `o` in the string as the only item in the CaptureCollection. That's because, after backtracking, the second `o` was subtracted from the group, but the first `o` was not.

To make sure the regex matches `oc` and `oocc` but not `ooc`, we need to check that the group "open" has no captures left when the matching process reaches the end of the regex. We can do this with a conditional. `(?(open)(?!))` is a conditional that checks whether the group "open" matched something. In PowerGREP, having matched something means still having captures on the stack that weren't backtracked or subtracted. If the group has captured something, the "if" part of the conditional is evaluated. In this case that is the empty negative lookahead `(?!)`. The empty string inside this lookahead always matches. Because the lookahead is negative, this causes the lookahead to always fail. Thus the conditional always fails if the group has captured something. If the group has not captured anything, the "else" part of the conditional is evaluated. In this case there is no "else" part. This means that the conditional always succeeds if the group has not captured something. This makes `(?(open)(?!))` a proper test to verify that the group "open" has no captures left.

The regex `^(?'open'o)+(?'-open'c)+(?(open)(?!))$` fails to match `ooc`. When `c` fails to match because the regex engine has reached the end of the string, the engine backtracks out of the balancing group, leaving "open" with a single capture. The regex engine now reaches the conditional, which fails to match. The regex engine will backtrack trying different permutations of the quantifiers, but they will all fail to match. No match can be found.

The regex `^(?'open'o)+(?'-open'c)+(?(open)(?!))$` does match `oocc`. After `(?'-open'c)+` has matched `cc`, the regex engine cannot enter the balancing group a third time, because "open" has no captures left. The engine advances to the conditional. The conditional succeeds because "open" has no captures left and the conditional does not have an "else" part. Now `$` matches at the end of the string.

Matching Balanced Constructs

`^(?:(?'open'o)+(?'-open'c)+)+(?(open)(?!))$` wraps the capturing group and the balancing group in a non-capturing group that is also repeated. This regex matches any string like `ooocooccocccoc` that contains any number of perfectly balanced o's and c's, with any number of pairs in sequence, nested to any depth. The balancing group makes sure that the regex never matches a string that has more c's at any point in the string than it has o's to the left of that point. The conditional at the end, which must remain outside the repeated group, makes sure that the regex never matches a string that has more o's than c's.

`^(?>(?'open'o)+(?'-open'c)+)+(?(open)(?!))$` optimizes the previous regex by using an atomic group instead of the non-capturing group. The atomic group, which is also non-capturing, eliminates nearly all backtracking when the regular expression cannot find a match, which can greatly increase performance when used on long strings with lots of o's and c's but that aren't properly balanced at the end. The atomic group does not change how the regex matches strings that do have balanced o's and c's.

`^m*(?>(?>(?'open'o)m*)+(?>(?'-open'c)m*)+)+(?(open)(?!))$` allows any number of letters `m` anywhere in the string, while still requiring all o's and c's to be balanced. `m*` at the start of the regex allows any number of m's before the first o. `(?'open'o)+` was changed into `(?>(?'open'o)m*)+` to allow any number of m's after each o. Similarly, `(?'-open'c)+` was changed into `(?>(?'-open'c)m*)+` to allow any number of m's after each c.

This is the generic solution for matching balanced constructs using PowerGREP's balancing groups or capturing group subtraction feature. You can replace `o`, `m`, and `c` with any regular expression, as long as no two of these three can match the same text.

`^[^()]*(?>(?>(?'open'\()[^()]*)+(?>(?'-open'\))[^()]*)+)+(?(open)(?!))$` applies this technique to match a string in which all parentheses are perfectly balanced.

Backreferences To Subtracted Groups

You can use backreferences to groups that have their matches subtracted by a balancing group. The backreference matches the group's most recent match that wasn't backtracked or subtracted. The regex `(?'x'[ab]){2}(?'-x')\k'x'` matches `aaa`, `aba`, `bab`, or `bbb`. It does not match `aab`, `abb`, `baa`, or `bba`. The first and third letters of the string have to be the same.

Let's see how `(?'x'[ab]){2}(?'-x')\k'x'` matches `aba`. The first iteration of `(?'x'[ab])` captures `a`. The second iteration captures `b`. Now the regex engine reaches the balancing group `(?'-x')`. It checks whether the group "x" has matched, which it has. The engine enters the balancing group, subtracting the match `b` from the stack of group "x". There are no regex tokens inside the balancing group. It matches without advancing through the string. Now the regex engine reaches the backreference `\k'x'`. The match at the top of the stack of group "x" is `a`. The next character in the string is also an `a` which the backreference matches. `aba` is found as an overall match.

When you apply this regex to `abb`, the matching process is the same, except that the backreference fails to match the second `b` in the string. Since the regex has no other permutations that the regex engine can try, the match attempt fails.

Matching Palindromes

`^(?'letter'[a-z])+[a-z]?(?:\k'letter'(?'-letter'))+(?(letter)(?!))$` matches palindrome words of any length. This regular expression takes advantage of the fact that backreferences and capturing group subtraction work well together. It also uses an empty balancing group as the regex in the previous section.

Let's see how this regex matches the palindrome `radar`. `^` matches at the start of the string. Then `(?'letter'[a-z])+` iterates five times. The group "letter" ends up with five matches on its stack: `r`, `a`, `d`, `a`, and `r`. The regex engine is now at the end of the string and at `[a-z]?` in the regex. It doesn't match, but that's fine, because the quantifier makes it optional. The engine now reaches the backreference `\k'letter'`. The group "letter" has `r` at the top of its stack. This fails to match the void after the end of the string.

The regex engine backtracks. `(?'letter'[a-z])+` is reduced to four iterations, leaving `r`, `a`, `d`, and `a` on the stack of the group "letter". `[a-z]?` matches `r`. The backreference again fails to match the void after the end of the string. The engine backtracks, forcing `[a-z]?` to give up its match. Now "letter" has `a` at the top of its stack. This causes the backreference to fail to match `r`.

More backtracking follows. `(?'letter'[a-z])+` is reduced to three iterations, leaving `d` at the top of the stack of the group "letter". The engine again proceeds with `[a-z]?`. It fails again because there is no `d` for the backreference to match.

Backtracking once more, the capturing stack of group "letter" is reduced to `r` and `a`. Now the tide turns. `[a-z]?` matches `d`. The backreference matches `a` which is the most recent match of the group "letter" that wasn't backtracked. The engine now reaches the empty balancing group `(?'-letter')`. This matches, because the group "letter" has a match `a` to subtract.

The backreference and balancing group are inside a repeated non-capturing group, so the engine tries them again. The backreference matches `r` and the balancing group subtracts it from "letter"'s stack, leaving the capturing group without any matches. Iterating once more, the backreference fails, because the group "letter" has no matches left on its stack. This makes the group act as a non-participating group. Backreferences to non-participating groups always fail in PowerGREP.

`(?:\k'letter'(?'-letter'))+` has successfully matched two iterations. Now, the conditional `(?(letter)(?!))` succeeds because the group "letter" has no matches left. The anchor `$` also matches. The palindrome `radar` has been matched.

30. Regular Expression Recursion

`(?R)`, `(?0)`, and `\g<0>` are three different ways to specify regular expression recursion. PowerGREP supports these syntactic variations for maximum compatibility with other regular expression flavors. As we'll see later, there are differences in how PowerGREP deals with backreferences and backtracking during recursion, depending on the syntax you choose. But these differences do not come into play in the basic example on this page.

Simple Recursion

The regexes `a(?R)?z`, `a(?0)?z`, and `a\g<0>?z` all match one or more letters `a` followed by exactly the same number of letters `z`. Since these regexes are functionally identical, we'll use the syntax with R for recursion to see how this regex matches the string `aaazzz`.

First, `a` matches the first `a` in the string. Then the regex engine reaches `(?R)`. This tells the engine to attempt the whole regex again at the present position in the string. Now, `a` matches the second `a` in the string. The engine reaches `(?R)` again. On the second recursion, `a` matches the third `a`. On the third recursion, `a` fails to match the first `z` in the string. This causes `(?R)` to fail. But the regex uses a quantifier to make `(?R)` optional. So the engine continues with `z` which matches the first `z` in the string.

Now, the regex engine has reached the end of the regex. But since it's two levels deep in recursion, it hasn't found an overall match yet. It only has found a match for `(?R)`. Exiting the recursion after a successful match, the engine also reaches `z`. It now matches the second `z` in the string. The engine is still one level deep in recursion, from which it exists with a successful match. Finally, `z` matches the third `z` in the string. The engine is again at the end of the regex. This time, it's not inside any recursion. Thus, it returns `aaazzz` as the overall regex match.

Matching Balanced Constructs

The main purpose of recursion is to match balanced constructs or nested constructs. The generic regex is `b(?:m|(?R))*e` where `b` is what begins the construct, `m` is what can occur in the middle of the construct, and `e` is what can occur at the end of the construct. For correct results, no two of `b`, `m`, and `e` should be able to match the same text. You can use an atomic group instead of the non-capturing group for improved performance: `b(?>m|(?R))*e`.

A common real-world use is to match a balanced set of parentheses. `\((?>[^()]|(?R))*\)` matches a single pair of parentheses with any text in between, including an unlimited number of parentheses, as long as they are all properly paired. If the subject string contains unbalanced parentheses, then the first regex match is the leftmost pair of balanced parentheses, which may occur after unbalanced opening parentheses. If you want a regex that does not find any matches in a string that contains unbalanced parentheses, then you need to use a subroutine call instead of recursion. If you want to find a sequence of multiple pairs of balanced parentheses as a single match, then you also need a subroutine call.

If what may appear in the middle of the balanced construct may also appear on its own without the beginning and ending parts then the generic regex is `b(?R)*e|m`. Again, `b`, `m`, and `e` all need to be mutually exclusive. `\((?R)*\)|[^()]+` matches a pair of balanced parentheses like the regex in the previous section. But it also matches any text that does not contain any parentheses at all.

31. Regular Expression Subroutines

Subroutine calls are very similar to regular expression recursion. Instead of matching the entire regular expression again, a subroutine call only matches the regular expression inside a capturing group. You can make a subroutine call to any capturing group from anywhere in the regex. If you place a call inside the group that it calls, you'll have a recursive capturing group.

As with regex recursion, there is a wide variety of syntax that you can use for exactly the same thing. One set of syntax, borrowed from Perl, uses `(?1)` to call a numbered group, `(?+1)` to call the next group, `(?-1)` to call the preceding group, and `(?&name)` to call a named group. You can use all of these to reference the same group. `(?+1)(?'name'[abc])(?1)(?-1)(?&name)` matches a string that is five letters long and consists only of the first three letters of the alphabet. This regex is exactly the same as `[abc](?'name'[abc])[abc][abc][abc]`.

The second set of syntax, borrowed from PCRE and expanded upon, uses `(?P>1)` to call a numbered group, `(?P>+1)` to call the next group, `(?P>-1)` to call the preceding group, and `(?P>name)` to call a named group. You can use all of these to reference the same group. `(?P>+1)(?P<name>[abc])(?P>1)(?P>-1)(?P>name)` matches a string that is five letters long and consists only of the first three letters of the alphabet. This regex is exactly the same as `[abc](?P<name>[abc])[abc][abc][abc]`.

The third set of syntax, borrowed from Ruby, looks more like that of backreferences. `\g<1>` and `\g'1'` call a numbered group, `\g<name>` and `\g'name'` call a named group, while `\g<-1>` and `\g'-1'` call the preceding group. `\g<+1>` and `\g'+1'` call the next group. `\g<+1>(?<name>[abc])\g<1>\g<-1>\g<name>` and `\g'+1'(?'name'[abc])\g'1'\g'-1'\g'name'` match the same 5-letter string as the preceding examples. The syntax with angle brackets and with quotes can be used interchangeably.

PowerGREP not only has three sets of syntax for subroutine calls. It also has three different behaviors for subroutine calls that mimic the behavior of Perl, PCRE, or Ruby, depending on the syntax you choose. This changes how PowerGREP deals with capturing, backreferences, and backtracking during subroutine calls. But these differences do not come into play in the basic examples on this page.

Matching Balanced Constructs

Recursion into a capturing group is a more flexible way of matching balanced constructs than recursion of the whole regex. We can wrap the regex in a capturing group, recurse into the capturing group instead of the whole regex, and add anchors outside the capturing group. `\A(b(?:m|(?1))*e)\z` is the generic regex for checking that a string consists entirely of a correctly balanced construct. Again, `b` is what begins the construct, `m` is what can occur in the middle of the construct, and `e` is what can occur at the end of the construct. For correct results, no two of `b`, `m`, and `e` should be able to match the same text. You can use an atomic group instead of the non-capturing group for improved performance: `\A(b(?>m|(?1))*e)\z`.

Similarly, `\Ao*(b(?:m|(?1))*eo*)+\z` and the optimized `\Ao*+(b(?>m|(?1))*+eo*+)++\z` match a string that consists of nothing but a sequence of one or more correctly balanced constructs, with possibly other text in between. Here, `o` is what can occur outside the balanced constructs. It will often be the same as `m`. `o` should not be able to match the same text as `b` or `e`.

`\A(\((?>[^()]|(?1))*\))\z` matches a string that consists of nothing but a correctly balanced pair of parentheses, possibly with text between them. `\A[^()]*+(\((?>[^()]|(?1))*+\)[^()]*+)++\z`.

Matching The Same Construct More Than Once

A regex that needs to match the same kind of construct (but not the exact same text) more than once in different parts of the regex can be shorter and more concise when using subroutine calls. Suppose you need a regex to match patient records like these:

```
Name: John Doe
Born: 17-Jan-1964
Admitted: 30-Jul-2013
Released: 3-Aug-2013
```

Further suppose that you need to match the date format rather accurately so the regex can filter out valid records, leaving invalid records for human inspection. In most regex flavors you could easily do this with this regex, using free-spacing syntax:

```
^Name:\ (.*)\r?\n
Born:\ (?:3[01]|[12][0-9]|[1-9])
       -(?:Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)
       -(?:19|20)[0-9][0-9]\r?\n
Admitted:\ (?:3[01]|[12][0-9]|[1-9])
           -(?:Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)
           -(?:19|20)[0-9][0-9]\r?\n
Released:\ (?:3[01]|[12][0-9]|[1-9])
           -(?:Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)
           -(?:19|20)[0-9][0-9]$
```

With subroutine calls you can make this regex much shorter, easier to read, and easier to maintain:

```
^Name:\ (.*)\r?\n
Born:\ (?'date'(?:3[01]|[12][0-9]|[1-9])
               -(?:Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)
               -(?:19|20)[0-9][0-9])\r?\n
Admitted:\ \g'date'\r?\n
Released:\ \g'date'$
```

Separate Subroutine Definitions

You can take this one step further using the special DEFINE group: `(?(DEFINE)(?'subroutine'regex))`. While this looks like a conditional that references the non-existent group DEFINE containing a single named group “subroutine”, the DEFINE group is a special syntax. The fixed text `(?(DEFINE)` opens the group. A parenthesis closes the group. This special group tells the regex engine to ignore its contents, other than to parse it for named and numbered capturing groups. You can put as many capturing groups inside the DEFINE group as you like. The DEFINE group itself never matches anything, and never fails to match. It is completely ignored. The regex `foo(?(DEFINE)(?'subroutine'skipped))bar` matches `foobar`. The DEFINE group is completely superfluous in this regex, as there are no calls to any of the groups inside of it.

With a DEFINE group, our regex becomes:

```
(?(DEFINE)(?'date'(?:3[01]|[12][0-9]|[1-9])
                  -(?:Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)
                  -(?:19|20)[0-9][0-9]))
^Name:\ (.*)\r?\n
Born:\ (?P>date)\r?\n
Admitted:\ (?P>date)\r?\n
Released:\ (?P>date)$
```

Quantifiers On Subroutine Calls

Quantifiers on subroutine calls work just like a quantifier on recursion. The call is repeated as many times in sequence as needed to satisfy the quantifier. `([abc])(?1){3}` matches `abcb` and any other combination of four-letter combination of the first three letters of the alphabet. First the group matches once, and then the call matches three times. This regex is equivalent to `([abc])[abc]{3}`.

Quantifiers on the group are ignored by the subroutine call. `([abc]){3}(?1)` also matches `abcb`. First, the group matches three times, because it has a quantifier. Then the subroutine call matches once, because it has no quantifier. `([abc]){3}(?1){3}` matches six letters, such as `abbcab`, because now both the group and the call are repeated 3 times. These two regexes are equivalent to `([abc]){3}[abc]` and `([abc]){3}[abc]{3}`.

32. Infinite Recursion

Regular expressions such as `(?R)?z` or `a?(?R)?z` or `a|(?R)z` that use recursion without having anything that must be matched in front of the recursion can result in infinite recursion. If the regex engine reaches the recursion without having advanced through the text then the next recursion will again reach the recursion without having advanced through the text. With the first regex this happens immediately at the start of the match attempt. With the other two this happens as soon as there are no further letters `a` to be matched.

PowerGREP treats the first two regexes as a syntax error because they always lead to infinite recursion. It allows the third regex because that one can match `a`.

Circular Infinite Subroutine Calls

Subroutine calls can also lead to infinite recursion. PowerGREP handles the potentially infinite recursion in `((?1)?z)` or `(a?(?1)?z)` or `(a|(?1)z)` in the same way as they handle potentially infinite recursion of the entire regex.

But subroutine calls that are not recursive by themselves may end up being recursive if the group they call has another subroutine call that calls a parent group of the first subroutine call. When subroutine calls are forced to go around in a circle that too leads to infinite recursion. Detecting such circular calls when compiling a regex is more complicated than checking for straight infinite recursion. Unlike most other applications, PowerGREP is able to detect this and treat it as a syntax error.

When infinite recursion does occur, whether it's straight recursion or subroutine calls going in circles, PowerGREP treats it as a matching error that aborts the entire match attempt.

Endless Recursion

A regex such as `a(?R)z` that has a recursion token that is not optional and is not have an alternative without the same recursion leads to endless recursion. Such a regular expression can never find a match. When `a` matches the regex engine attempts the recursion. If it can match another `a` then it has to attempt the recursion again. Eventually `a` will run out of letters to match. The recursion then fails. Because it's not optional the regex fails to match.

PowerGREP detects this situation when compiling your regular expression. It flags endless recursion as a syntax error.

33. Quantifiers On Recursion

The introduction to recursion shows how `a(?R)?z` matches `aaazzz`. The quantifier `?` makes the preceding token optional. In other words, it repeats the token between zero or one times. In `a(?R)?z` the `(?R)` is made optional by the `?` that follows it. You may wonder why the regex attempted the recursion three times, instead of once or not at all.

The reason is that upon recursion, the regex engine takes a fresh start in attempting the whole regex. All quantifiers and alternatives behave as if the matching process prior to the recursion had never happened at all, other than that the engine advanced through the string. The regex engine restores the states of all quantifiers and alternatives when it exits from a recursion, whether the recursion matched or failed. Basically, the matching process continues normally as if the recursion never happened, other than that the engine advanced through the string.

If you're familiar with procedural programming languages, regex recursion is basically a recursive function call and the quantifiers are local variables in the function. Each recursion of the function gets its own set of local variables that don't affect and aren't affected by the same local variables in recursions higher up the stack.

Let's see how `a(?R){3}z|q` behaves. The simplest possible match is `q`, found by the second alternative in the regex.

The simplest match in which the first alternative matches is `aqqqz`. After `a` is matches, the regex engine begins a recursion. `a` fails to match `q`. Still inside the recursion, the engine attempts the second alternative. `q` matches `q`. The engine exits from the recursion with a successful match. The engine now notes that the quantifier `{3}` has successfully repeated once. It needs two more repetitions, so the engine begins another recursion. It again matches `q`. On the third iteration of the quantifier, the third recursion matches `q`. Finally, `z` matches `z` and an overall match is found.

This regex does not match `aqqz` or `aqqqqz`. `aqqz` fails because during the third iteration of the quantifier, the recursion fails to match `z`. `aqqqqz` fails because after `a(?R){3}` has matched `aqqq`, `z` fails to match the fourth `q`.

The regex can match longer strings such as `aqaqqqzqz`. With this string, during the second iteration of the quantifier, the recursion matches `aqqqz`. Since each recursion tracks the quantifier separately, the recursion needs three consecutive recursions of its own to satisfy its own instance of the quantifier. This can lead to arbitrarily long matches such as `aaaqqaqqqzzaqqqzqzqaqqaaqqqzqqzzz`.

Boost's issues with quantifiers on recursion also affect quantifiers on parent groups of the recursion token. They also affect quantifiers on subroutine calls and quantifiers groups that contain a subroutine call to a parent group of the group with the quantifier.

Quantifiers on Other Tokens in The Recursion

Quantifiers on other tokens in the regex behave normally during recursion. They track their iterations separately at each recursion. So `a{2}(?R)z|q` matches `aaqz`, `aaaaqzz`, `aaaaaaqzzz`, and so on. `a` has to match twice during each recursion.

Quantifiers like these that are inside the recursion but do not repeat the recursion itself do work correctly in Boost.

34. Subroutine Calls May or May Not Capture

This tutorial introduced regular expression subroutines with this example that we want to match accurately:

```
Name: John Doe
Born: 17-Jan-1964
Admitted: 30-Jul-2013
Released: 3-Aug-2013
```

You can use this regular expression with syntax borrowed from Ruby. You can remember this by the fact that the `\g` syntax is a Ruby invention.

```
^Name:\ (.*)\n
Born:\ (?'date'(?:3[01]|[12][0-9]|[1-9])
               -(?:Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)
               -(?:19|20)[0-9][0-9])\n
Admitted:\ \g'date'\n
Released:\ \g'date'$
```

Or you can use this regular expression with syntax borrowed from Perl. You can remember this by the fact that Perl uses ampersands for subroutine calls in procedural code too.

```
^Name:\ (.*)\n
Born:\ (?'date'(?:3[01]|[12][0-9]|[1-9])
               -(?:Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)
               -(?:19|20)[0-9][0-9])\n
Admitted:\ (?&date)\n
Released:\ (?&date)$
```

Finally, you can use this regular expression with syntax borrowed from PCRE. You can remember this by the fact that PCRE 4.0 invented subroutine calls using this syntax.

```
^Name:\ (.*)\n
Born:\ (?P<date>(?:3[01]|[12][0-9]|[1-9])
                  -(?:Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)
                   -(?:19|20)[0-9][0-9])\n
Admitted:\ (?P>date)\n
Released:\ (?P>date)$
```

A subroutine call using the Ruby syntax in PowerGREP makes the capturing group store the text matched during the subroutine call, just like subroutine calls always do in Ruby. A subroutine call using the Perl or PCRE syntax in PowerGREP does not affect the group that is called, just like subroutine calls never affect the called group in Perl or PCRE.

If you want to extract the dates from the match, the best solution is to add another capturing group for each date. Then you can ignore the text stored by the "date" group and this particular difference between these flavors. Then it doesn't matter which syntax you use for subroutine calls in PowerGREP.

```
^Name:\ (.*)\n
Born:\ (?'born'(?'date'(?:3[01]|[12][0-9]|[1-9])
                       -(?:Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)
                       -(?:19|20)[0-9][0-9]))\n
```

```
Admitted:\ (?'admitted'\g'date')\n
Released:\ (?'released'\g'date')$
```

Capturing Groups Inside Recursion or Subroutine Calls

There are further differences between the Ruby syntax versus the Perl or PCRE syntax for subroutine calls in PowerGREP when your regex makes a subroutine call or recursive call to a capturing group that contains other capturing groups. The same issues also affect recursion of the whole regular expression if it contains any capturing groups. For the remainder of this topic, the term "recursion" applies equally to recursion of the whole regex, recursion into a capturing group, or a subroutine call to a capturing group.

When you use the Perl or PCRE syntax, PowerGREP backs up and restores capturing groups when entering and exiting recursion, just like PCRE and Perl 5.20 and later do. When the regex engine enters recursion, it internally makes a copy of all capturing groups. This does not affect the capturing groups. Backreferences inside the recursion match text captured prior to the recursion unless and until the group they reference captures something during the recursion. After the recursion, all capturing groups are replaced with the internal copy that was made at the start of the recursion. Text captured during the recursion is discarded. This means you cannot use capturing groups to retrieve parts of the text that were matched during recursion.

When you use the Ruby syntax, PowerGREP's behavior is completely different. When the regex engine enters or exists recursion, it makes no changes to the text stored by capturing groups at all. Backreferences match the text stored by the capturing group during the group's most recent match, irrespective of any recursion that may have happened. After an overall match is found, each capturing group still stores the text of its most recent match, even if that was during a recursion. This means you can use capturing groups to retrieve part of the text matched during the last recursion.

Odd Length Palindromes Using The Perl or PCRE Syntax

Using the Perl syntax, you can use `\b(?'word'(?'letter'[a-z])(?&word)\k'letter'|[a-z])\b` to match palindrome words such as `a`, `dad`, `radar`, `racecar`, and `redivider`. This regex only matches palindrome words that are an odd number of letters long. This covers most palindrome words in English. To extend the regex to also handle palindrome words that are an even number of characters long we have to worry about differences in how Perl and PCRE backtrack after a failed recursion attempt which is discussed later in this tutorial. We gloss over these differences here because they only come into play when the subject string is not a palindrome and no match can be found.

Let's see how this regex matches `radar`. The word boundary `\b` matches at the start of the string. The regex engine enters the two capturing groups. `[a-z]` matches `r` which is then stored in the capturing group "letter". Now the regex engine enters the first recursion of the group "word". At this point, Perl forgets that the "letter" group matched `r`. PCRE does not. But this does not matter. `(?'letter'[a-z])` matches and captures `a`. The regex enters the second recursion of the group "word". `(?'letter'[a-z])` captures `d`. During the next two recursions, the group captures `a` and `r`. The fifth recursion fails because there are no characters left in the string for `[a-z]` to match. The regex engine must backtrack.

Because `(?&word)` failed to match, `(?'letter'[a-z])` must give up its match. The group reverts to `a`, which was the text the group held at the start of the recursion. Again, this does not matter because the regex engine must now try the second alternative inside the group "word", which contains no backreferences. The second `[a-z]` matches the final `r` in the string. The engine now exits from a successful recursion. The text stored by the group “letter” is restored to what it had captured prior to entering the fourth recursion, which is `a`.

After matching `(?&word)` the engine reaches `\k'letter'`. The backreference fails because the regex engine has already reached the end of the subject string. So it backtracks once more, making the capturing group give up the `a`. The second alternative now matches the `a`. The regex engine exits from the third recursion. The group “letter” is restored to the `d` matched during the second recursion.

The regex engine has again matched `(?&word)`. The backreference fails again because the group stores `d` while the next character in the string is `r`. Backtracking again, the second alternative matches `d` and the group is restored to the `a` matched during the first recursion.

Now, `\k'letter'` matches the second `a` in the string. That’s because the regex engine has arrived back at the first recursion during which the capturing group matched the first `a`. The regex engine exits the first recursion. The capturing group to the `r` which it matched prior to the first recursion.

Finally, the backreference matches the second `r`. Since the engine is not inside any recursion any more, it proceeds with the remainder of the regex after the group. `\b` matches at the end of the string. The end of the regex is reached and `radar` is returned as the overall match. If you query the groups “word” and “letter” after the match you’ll get `radar` and `r`. That’s the text matched by these groups outside of all recursion.

Why This Regex Does Not Work Using The Ruby Syntax

To match palindromes this way using the Ruby syntax, you need to use a special backreference that specifies a recursion level. If you use a normal backreference as in `\b(?'word'(?'letter'[a-z])\g'word'\k'letter'|[a-z])\b`, PowerGREP will not complain. But it will not match palindromes longer than three letters either. Instead this regex matches things like `a`, `dad`, `radaa`, `raceccc`, and `rediviiii`.

Let’s see why this regex does not match `radar` using the Ruby syntax. The regex starts out like the one with the Perl syntax, entering the recursions until there are no characters left in the string for `[a-z]` to match.

Because `\g'word'` failed to match, `(?'letter'[a-z])` must give up its match. PowerGREP reverts it to `a`, which was the text the group most recently matched. The second `[a-z]` matches the final `r` in the string. The engine now exits from a successful recursion. The group “letter” continues to hold its most recent match `a`.

After matching `\g'word'` the engine reaches `\k'letter'`. The backreference fails because the regex engine has already reached the end of the subject string. So it backtracks once more, reverting the group to the previously matched `d`. The second alternative now matches the `a`. The regex engine exits from the third recursion.

The regex engine has again matched `\g'word'`. The backreference fails again because the group stores `d` while the next character in the string is `r`. Backtracking again, the group reverts to `a` and the second alternative matches `d`.

Now, `\k'letter'` matches the second `a` in the string. The regex engine exits the first recursion which successfully matched `ada`. The capturing group continues to hold `a` which is its most recent match that wasn’t backtracked.

The regex engine is now at the last character in the string. This character is `r`. The backreference fails because the group still holds `a`. The engine can backtrack once more, forcing `(?'letter'[a-z])\g'word'\k'letter'` to give up the `rada` it matched so far. The regex engine is now back at the start of the string. It can still try the second alternative in the group. This matches the first `r` in the string. Since the engine is not inside any recursion any more, it proceeds with the remainder of the regex after the group. `\b` fails to match after the first `r`. The regex engine has no further permutations to try. The match attempt has failed.

If the subject string is `radaa`, PowerGREP's engine goes through nearly the same matching process as described above. Only the events described in the last paragraph change. When the regex engine reaches the last character in the string, that character is now `a`. This time, the backreference matches. Since the engine is not inside any recursion any more, it proceeds with the remainder of the regex after the group. `\b` matches at the end of the string. The end of the regex is reached and `radaa` is returned as the overall match. If you query the groups "word" and "letter" after the match you'll get `radaa` and `a`. Those are the most recent matches of these groups that weren't backtracked.

Basically, this regex matches any word that is an odd number of letters long and in which all the characters to the right of the middle letter are identical to the character just to the left of the middle letter. That's because PowerGREP only restores capturing groups when they backtrack, but not when it exits from recursion using the Ruby syntax.

The solution, when using the Ruby syntax, is to use a backreference that specifies a recursion level instead of the normal backreference used in the regex on this page.

35. Backreferences That Specify a Recursion Level

Earlier topics in this tutorial explain regular expression recursion and regular expression subroutines. In this topic the word "recursion" refers to recursion of the whole regex, recursion of capturing groups, and subroutine calls to capturing groups. The previous topic also explained that these features handle capturing groups differently in PowerGREP when using the Ruby syntax than when using the Perl or PCRE syntax.

When you use the Ruby `\g` syntax for recursion, then backreferences in PowerGREP can match the same text as was matched by a capturing group at any recursion level relative to the recursion level that the backreference is evaluated at. You can do this with the same syntax for named backreferences by adding a sign and a number after the name. In most situations you will use `+0` to specify that you want the backreference to reuse the text from the capturing group at the same recursion level. You can specify a positive number to reference the capturing group at a deeper level of recursion. This would be a recursion the regex engine has already exited from. You can specify a negative number to reference the capturing group a level that is less deep. This would be a recursion that is still in progress.

Odd Length Palindromes The Ruby Way

You can use `\b(?'word'(?'letter'[a-z])\g'word'\k'letter+0'|[a-z])\b` to match palindrome words such as `a`, `dad`, `radar`, `racecar`, and `redivider`. To keep this example simple, this regex only matches palindrome words that are an odd number of letters long.

Let's see how this regex matches `radar`. The word boundary `\b` matches at the start of the string. The regex engine enters the capturing group "word". `[a-z]` matches `r` which is then stored in the stack for the capturing group "letter" at recursion level zero. Now the regex engine enters the first recursion of the group "word". `(?'letter'[a-z])` matches and captures `a` at recursion level one. The regex enters the second recursion of the group "word". `(?'letter'[a-z])` captures `d` at recursion level two. During the next two recursions, the group captures `a` and `r` at levels three and four. The fifth recursion fails because there are no characters left in the string for `[a-z]` to match. The regex engine must backtrack.

The regex engine must now try the second alternative inside the group "word". The second `[a-z]` in the regex matches the final `r` in the string. The engine now exits from a successful recursion, going one level back up to the third recursion.

After matching `\g'word'` the engine reaches `\k'letter+0'`. The backreference fails because the regex engine has already reached the end of the subject string. So it backtracks once more. The second alternative now matches the `a`. The regex engine exits from the third recursion.

The regex engine has again matched `\g'word'` and needs to attempt the backreference again. The backreference specifies `+0` or the present level of recursion, which is 2. At this level, the capturing group matched `d`. The backreference fails because the next character in the string is `r`. Backtracking again, the second alternative matches `d`.

Now, `\k'letter+0'` matches the second `a` in the string. That's because the regex engine has arrived back at the first recursion during which the capturing group matched the first `a`. The regex engine exits the first recursion.

The regex engine is now back outside all recursion. That this level, the capturing group stored `r`. The backreference can now match the final `r` in the string. Since the engine is not inside any recursion any more, it proceeds with the remainder of the regex after the group. `\b` matches at the end of the string. The end of the regex is reached and `radar` is returned as the overall match.

Backreferences to Other Recursion Levels

Backreferences to other recursion levels can be easily understood if we modify our palindrome example. `abcdefedcba` is also a palindrome matched by the previous regular expression. Consider the regular expression `\b(?'word'(?'letter'[a-z])\g'word'(?:\k'letter-1'|z)|[a-z])\b`. The backreference now wants a match the text one level less deep on the capturing group's stack. It is alternated with the letter `z` so that something can be matched when the backreference fails to match.

The new regex matches things like `abcdefdcbaz`. After a whole bunch of matching and backtracking, the second `[a-z]` matches `f`. The regex engine exits form a successful fifth recursion. The capturing group "letter" has stored the matches `a`, `b`, `c`, `d`, and `e` at recursion levels zero to four. Other matches by that group were backtracked and thus not retained.

Now the engine evaluates the backreference `\k'letter-1'`. The present level is 4 and the backreference specifies -1. Thus the engine attempts to match `d`, which succeeds. The engine exits from the fourth recursion.

The backreference continues to match `c`, `b`, and `a` until the regex engine has exited the first recursion. Now, outside all recursion, the regex engine again reaches `\k'letter-1'`. The present level is 0 and the backreference specifies -1. Since recursion level -1 never happened, the backreference fails to match. This is not an error but simply a backreference to a non-participating capturing group. But the backreference has an alternative. `z` matches `z` and `\b` matches at the end of the string. `abcdefdcbaz` was matched successfully.

You can take this as far as you like. The regular expression `\b(?'word'(?'letter'[a-z])\g'word'(?:\k'letter-2'|z)|[a-z])\b` matches `abcdefcbazz`. `\b(?'word'(?'letter'[a-z])\g'word'(?:\k'letter-99'|z)|[a-z])\b` matches `abcdefzzzzzz`.

Going in the opposite direction, `\b(?'word'(?'letter'[a-z])\g'word'(?:\k'letter+1'|z)|[a-z])\b` matches `abcdefzedcb`. Again, after a whole bunch of matching and backtracking, the second `[a-z]` matches `f`, the regex engine is back at recursion level 4, and the group "letter" has `a`, `b`, `c`, `d`, and `e` at recursion levels zero to four on its stack.

Now the engine evaluates the backreference `\k'letter+1'`. The present level is 4 and the backreference specifies +1. The capturing group was backtracked at recursion level 5. This means we have a backreference to a non-participating group, which fails to match. The alternative `z` does match. The engine exits from the fourth recursion.

At recursion level 3, the backreference points to recursion level 4. Since the capturing group successfully matched at recursion level 4, it still has that match on its stack, even though the regex engine has already exited from that recursion. Thus `\k'letter+1'` matches `e`. Recursion level 3 is exited successfully.

The backreference continues to match `d` and `c` until the regex engine has exited the first recursion. Now, outside all recursion, the regex engine again reaches `\k'letter+1'`. The present level is 0 and the backreference specifies +1. The capturing group still retains all its previous successful recursion levels. So the backreference can still match the `b` that the group captured during the first recursion. Now `\b` matches at the end of the string. `abcdefzdcb` was matched successfully.

You can take this as far as you like in this direction too. The regular expression `\b(?'word'(?'letter'[a-z])\g'word'(?:\k'letter+2'|z)|[a-z])\b` matches `abcdefzzedc`. `\b(?'word'(?'letter'[a-z])\g'word'(?:\k'letter+99'|z)|[a-z])\b` matches `abcdefzzzzzz`.

36. Recursion and Subroutine Calls May or May Not Be Atomic

Earlier topics in this tutorial explain regular expression recursion and regular expression subroutines. In this topic the word “recursion” refers to recursion of the whole regex, recursion of capturing groups, and subroutine calls to capturing groups.

PowerGREP lets you choose whether recursion should be atomic or not. Atomic recursion gives better performance, but may exclude certain matches or find different matches as illustrated above. `(?P>0)` is atomic in PowerGREP. You can remember this because this syntax for recursion uses an angle bracket just like an atomic group. You can use a number or a name instead of zero for an atomic subroutine call to a numbered or named capturing group. Any other syntax for recursion is not atomic in PowerGREP.

Consider the regular expressions `aa$|a(?R)a|a` and `aa$|a(?P>0)a|a`. The first uses non-atomic recursion, while the second uses atomic recursion. Let’s see how PowerGREP goes through the matching process of these regexes when `aaa` is the subject string.

The first alternative `aa$` fails because the anchor cannot be matched between the second and third `a` in the string. Attempting the second alternative at the start of the string, `a` matches `a`. Now the regex engine enters the first recursion.

Inside the recursion, the first alternative matches the second and third `a` in the string. The regex engine exists a successful recursion. But now, the `a` that follows `(?R)` or `(?P>0)` in the regex fails to match because the regex engine has already reached the end of the string. Thus the regex engine must backtrack. Here is where the two regexes behave differently.

For the first regex, PowerGREP remembers that inside the recursion the regex matched the second alternative and that there are three possible alternatives. PowerGREP backtracks *into* the recursion. The second alternative inside the recursion is backtracked, reducing the match so far to the first `a` in the string. Now the third alternative is attempted. `a` matches the second `a` in the string. The regex engine again exits successfully from the same recursion. This time, the `a` that follows `(?R)` in the regex matches the third `a` in the string. `aaa` is found as the overall match.

For the second regex, on the other hand, PowerGREP remembers nothing about the recursion other than that it matched `aa` at the end of the string. PowerGREP does backtrack *over* the recursion, reducing the match so far to the first `a` in the string. But this leaves the second alternative in the regex without any further permutations to try. Thus the `a` at the start of the second alternative is also backtracked, reducing the match so far to nothing. PowerGREP continues the match attempt at the start of the string with the third alternative and finds that `a` matches `a` at the start of the string. For the second regex, this is the overall match.

Palindromes of Any Length with Backtracking into Recursion

The topic about recursion and capturing groups explains a regular expression to match palindromes that are an odd number of characters long. The solution seems trivial. `\b(?'word'(?'letter'[a-z])(?&word)\k'letter'|[a-z]?)\b` does the trick Perl-style. The quantifier `?` makes the `[a-z]` that matches the letter in the middle of the palindrome optional. Ruby-style we can use `\b(?'word'(?'letter'[a-z])\g'word'\k'letter+0'|[a-z]?)\b` which adds the same quantifier to the solution that specifies the recursion level for the backreference. PCRE-style, with atomic subroutine calls,

`\b(?'word'(?'letter'[a-z])(?P>word)\k'letter'|[a-z]?)\b` still matches odd-length palindromes, but not even-length palindromes.

Let's see how these regexes match or fail to match `deed`. All 3 regexes start off the same. The group "letter" matches `d`. During three consecutive recursions, the group captures `e`, `e`, and `d`. The fourth recursion fails, because there are no characters left the match. Back in the third recursion, the first alternative is backtracked and the second alternative matches `d` at the end of the string. The engine exists the third recursion with a successful match. Back in the second recursion, the backreference fails because there are no characters left in the string.

Here the behavior diverges. The first 2 regexes backtrack *into* the third recursion and backtrack the quantifier `?` that makes the second alternative optional. In the third recursion, the second alternative gives up the `d` that it matched at the end of the string. The engine exists the third recursion again, this time with a successful zero-length match. Back in the second recursion, the backreference still fails because the group stored `e` for the second recursion but the next character in the string is `d`. This the first alternative is backtracked and the second alternative matches the second `e` in the string. The second recursion is exited with success.

In the first recursion, the backreference again fails. The group stored `e` for the first recursion but the next character in the string is `d`. Again, PowerGREP backtracks into the second recursion to try the permutation where the second alternative finds a zero-length match. Back in the first recursion again, the backreference now matches the second `e` in the string. The engine leaves the first recursion with success. Back in the overall match attempt, the backreference matches the final `d` in the string. The word boundary succeeds and an overall match is found.

The third regex, however, does not backtrack into the third recursion. It does backtrack *over* the third recursion when it backtracks the first alternative in the second recursion. Now, the second alternative in the second alternative matches the second `e` in the string. The second recursion is exited with success.

In the first recursion, the backreference again fails. The group stored `e` for the first recursion but the next character in the string is `d`. Again, PCRE does not backtrack into the second recursion, but immediately fails the first alternative in the first recursion. The second alternative in the first recursion now matches the first `e` in the string. PowerGREP exits the first recursion with success. Back in the overall match attempt, the backreference fails, because the group captured `d` prior to the recursion, and the next character is the second `e` in the string. Backtracking again, the second alternative in the overall regex match now matches the first `d` in the string. Then the word boundary fails. The regex with the atomic subroutine call did not find any matches.

Palindromes of Any Length with Atomic Recursion

To match palindromes of any length with atomic subroutine calls, we need a regex that matches words of an even number of characters and of and number of characters separately. Free-spacing mode makes this regex easier to read:

```
\b(?'word'
  (?'oddword' (?'oddletter' [a-z])(?P>oddword)  \k'oddletter' |[a-z])
| (?'evenword'(?'evenletter'[a-z])(?P>evenword)?\k'evenletter')
)\b
```

Basically, this is two copies of the original regex combined with alternation. The first alternatives has the groups "word" and "letter" renamed to "oddword" and "oddletter". The second alternative has the groups "word" and "letter" renamed to "evenword" and "evenletter". The call `(?P>evenword)` is now made optional with a question mark instead of the alternative `|[a-z]`. A new group "word" combines the two groups "oddword" and "evenword" so that the word boundaries still apply to the whole regex.

The first alternative "oddword" in this regex matches a palindrome of odd length like `radar` in exactly the same way as the regex discussed in the topic about recursion and capturing groups does. The second alternative in the new regex is never attempted.

When the string is a palindrome of even length like `deed`, the new regex first tries all permutations of the first alternative. The second alternative "evenword" is attempted only after the first alternative fails to find a match.

The second alternative off in the same as the original regex. The group "evenletter" matches `d`. During three consecutive recursions, the group captures `e`, `e`, and `d`. The fourth recursion fails, because there are no characters left the match. Back in the third recursion, the regex engine notes that recursive call `(?P>evenword)?` is optional. It proceeds to the backreference `\k'evenletter'`. The backreference fails because there are no characters left in the string. Since the recursion has no further alternatives to try, is is backtracked. The group "evenletter" must give up its most recent match and PowerGREP exits from the failed third recursion.

In the second recursion, the backreference fails because the capturing group matched `e` during that recursion but the next character in the string is `d`. The group gives up another match and PowerGREP exits from the failed second recursion.

Back in the first recursion, the backreference succeeds. The group matched the first `e` in the string during that recursion and the backreference matches the second. PowerGREP exits from the successful first recursion.

Back in the overall match attempt, the backreference succeeds again. The group matched the `d` at the start of the string during the overall match attempt, and the backreference matches the final `d`. Exiting the groups "evenword" and "word", the word boundary matches at the end of the string. `deed` is the overall match.

37. POSIX 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 this tutorial calls a "character class" and what POSIX calls a "bracket expression". `[:digit:]` is a POSIX character class, used inside a bracket expression like `[x-z[:digit:]]`. The POSIX character class names must be written all lowercase.

When used on ASCII strings, these two regular expressions find exactly the same matches: a single character that is either `x`, `y`, `z`, or a digit. When used on strings with non-ASCII characters, the `[:digit:]` class may include digits in other scripts, depending on the locale.

The POSIX standard defines 12 character classes. The table below lists all 12, plus the `[:ascii:]` and `[:word:]` classes that PowerGREP also supports. The table also shows equivalent character classes that you can use in ASCII and Unicode regular expressions if the POSIX classes are unavailable. The ASCII equivalents correspond exactly what is defined in the POSIX standard. The Unicode equivalents correspond to what PowerGREP matches. The POSIX standard does not define a Unicode locale. 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. 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.

PowerGREP supports both the POSIX and Java syntax. It matches only ASCII characters when using the POSIX syntax, and Unicode characters when using the Java syntax.

`[:alnum:]` Alphanumeric characters `[a-zA-Z0-9]` `[\p{L}\p{Nl}\p{Nd}]` `\p{Alnum}`

POSIX:	`[:alpha:]`
Description:	Alphabetic characters
ASCII:	`[a-zA-Z]`
Unicode:	`\p{L}\p{Nl}`
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:	`\h`
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 (anything except spaces and control characters)
ASCII:	[\x21-\x7E]
Unicode:	[^\p{Z}\p{C}]
Shorthand:	
Java:	\p{Graph}

POSIX:	[:lower:]
Description:	Lowercase letters
ASCII:	[a-z]
Unicode:	\p{Ll}
Shorthand:	\l
Java:	\p{Lower}

POSIX:	[:print:]
Description:	Visible characters and spaces (anything except control characters)
ASCII:	[\x20-\x7E]
Unicode:	\P{C}
Shorthand:	
Java:	\p{Print}

[:punct:] Punctuation and symbols. [!"\#$%&'()*+,
\-./:;<=>?@\[
\\\]^_`{|}~] \p{P} \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:	\u
Java:	\p{Upper}

`[:word:]` Word characters (letters, numbers and underscores) `[A-Za-z0-9_]` `[\p{L}\p{Nl}\p{Nd}\p{Pc}]` `\w`

POSIX:	`[:xdigit:]`
Description:	Hexadecimal digits
ASCII:	`[A-Fa-f0-9]`
Unicode:	`[A-Fa-f0-9]`
Shorthand:	
Java:	`\p{XDigit}`

38. Zero-Length Regex Matches

We saw that anchors, word boundaries, and lookaround match at a position, rather than matching a character. This means that when a regex only consists of one or more anchors, word boundaries, or lookaorunds, then it can result in a zero-length match. Depending on the situation, this can be very useful or undesirable.

In email, for example, it is common to prepend a "greater than" symbol and a space to each line of the quoted message. We can easily to this by replacing all matches of the regex `^` with `> `. Though this regex does not match any characters, it does find a position. The replacement string is inserted there, just like we want it.

Using `^\d*$` to test if the user entered a number would give undesirable results. It causes the script to accept an empty string as a valid input. 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*`. One of the star's effects is that it makes the `\d`, in this case, optional. The engine tries to match `\d` with the void after the string. That fails. But the star turns the failure of the `\d` into a zero-length success. The engine proceeds with the next regex token, without advancing the position in the string. So the engine arrives at `$`, and the void after the string. These match. At this point, the entire regex has matched the empty string, and the engine reports success.

The solution is to use the regex `^\d+$` with the proper quantifier to require at least one digit to be entered. If you always make sure that your regexes cannot find zero-length matches, other than special cases such as matching the start or end of each line, then you can save yourself the headache you'll get from reading the remainder of this topic.

Advancing After a Zero-Length Regex Match

If a regex can find zero-length matches at any position in the string, then it will. The regex `\d*` matches zero or more digits. If the subject string does not contain any digits, then this regex finds a zero-length match at every position in the string. It finds 4 matches in the string `abc`, one before each of the three letters, and one at the end of the string.

Things get tricky when a regex can find zero-length matches at any position as well as certain non-zero-length matches. Say we have the regex `\d*|x`, the subject string `x1`, and a regex engine allows zero-length matches. Which and how many matches do we get when iterating over all matches?

The first match attempt begins at the start of the string. `\d` fails to match `x`. But the `*` makes `\d` optional. The first alternative finds a zero-length match at the start of the string.

Now the regex engine is in a tricky situation. We're asking it to go through the entire string to find all non-overlapping regex matches. The first match ended at the start of the string, where the first match attempt began. The regex engine needs a way to avoid getting stuck in an infinite loop that forever finds the same zero-length match at the start of the string.

PowerGREP's solution, which is used by most regex engines, is to start the next match attempt one character after the end of the previous match, if the previous match was zero-length. In this case, the second match attempt begins at the position between the `x` and the `1` in the string. `\d` matches `1`. The end of the string is reached. The quantifier `*` is satisfied with a single repetition. `1` is returned as the overall match.

PowerGREP has an extra rule to skip zero-length matches at the position where the previous match ended, so you can never have a zero-length match immediately adjacent to a non-zero-length match. In our example PowerGREP only finds two matches: the zero-length match at the start of the string, and `1`.

39. 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.

40. Replacement Strings Tutorial

A replacement string, also known as the replacement text, is the text that each regular expression match is replaced with during a search-and-replace. In most applications, the replacement text supports special syntax that allows you to reuse the text matched by the regular expression or parts thereof in the replacement. This tutorial explains this syntax. While replacement strings are fairly simple compared with regular expressions, there is still great variety between the syntax used by various applications and their actual behavior.

In this book, replacement strings are shown as `replace` like you would enter them in the Replace box of an application. Literal text in the replacement is highlighted in yellow. As `$&\$` shows, special tokens are highlighted in blue and escaped characters in gray.

Table of Contents

Conditionals

PowerGREP can use one replacement or another replacement depending on whether a capturing group participated in the match. This allows you to use different replacements for different matches of the regular expression.

41. Special Characters

The most basic replacement string consists only of literal characters. The replacement `replacement` simply replaces each regex match with the text `replacement`.

Because we want to be able to do more than simply replace each regex match with the exact same text, we need to reserve certain characters for special use. In most replacement text flavors, two characters tend to have special meanings: the backslash `\` and the the dollar sign `$`. Whether and how to escape them depends on the application you're using. In some applications, you always need to escape them when you want to use them as literal characters. In other applications, you only need to escape them if they would form a replacement text token with the character that follows.

In PowerGREP, you can use a backslash to escape the backslash and the dollar, and you can use a dollar to escape the dollar. `\\` replaces with a literal backslash, while `\$` and `$$` replace with a literal dollar sign. You only need to escape them to suppress their special meaning in combination with other characters. In `\!` and `$!`, the backslash and dollar are literal characters because they don't have a special meaning in combination with the exclamation point. You can't and needn't escape the exclamation point or any other character except the backslash and dollar, because they have no special meaning in PowerGREP replacement strings.

42. Non-Printable Characters

PowerGREP allows you to use special escape sequences to enter a few common control characters. Use `\t` to replace with a tab character (ASCII 0x09), `\r` for carriage return (0x0D), and `\n` for line feed (0x0A). Remember that Windows text files use `\r\n` to terminate lines, while UNIX text files use `\n`.

PowerGREP also supports hexadecimal escapes. You can use `\uFFFF` or `\x{FFFF}` to insert a Unicode character. The euro currency sign occupies Unicode code point U+20AC. If you cannot type it on your keyboard, you can insert it into the replacement text with `\x{20AC}`. The JGsoft apps also support `\u20AC`. For the 127 ASCII characters, you can use `\x00` through `\x7F`. If you are working with files using 8-bit code pages in PowerGREP you can also use `\x80` through `\xFF` to insert characters from those 8-bit code pages.

43. Matched Text

Reinserting the entire regex match into the replacement text allows a search-and-replace to insert text before and after regular expression matches without really replacing anything. It also allows you to replace the regex match with something that contains the regex match. For example, you could replace all matches of the regex `http://\S+` with `<a href="$&">$&</a>` to turn all URLs in a file into HTML anchors that link to those URLs.

`$&` is substituted with the whole regex match in the replacement text. `\&` works too.

The overall regex match is treated as an implied capturing group number zero. You can use the syntax for backreferences in the replacement text to reference group number zero and thus insert the whole regex match in the replacement text.

44. Numbered and Named Backreferences

If your regular expression has named or numbered capturing groups, then you can reinsert the text matched by any of those capturing groups in the replacement text. Your replacement text can reference as many groups as you like, and can even reference the same group more than once. This makes it possible to rearrange the text matched by a regular expression in many different ways. As a simple example, the regex `\*(\w+)\*` matches a single word between asterisks, storing the word in the first (and only) capturing group. The replacement text `<b>\1</b>` replaces each regex match with the text stored by the capturing group between bold tags. Effectively, this search-and-replace replaces the asterisks with bold tags, leaving the word between the asterisks in place. This technique using backreferences is important to understand. Replacing `*word*` as a whole with `<b>word</b>` is far easier and far more efficient than trying to come up with a way to correctly replace the asterisks separately.

The `\1` syntax for backreferences in the replacement text is borrowed from the syntax for backreferences in the regular expression. PowerGREP supports `\1` through `\99`. If there are not enough capturing groups in the regex for the double-digit backreference to be valid, then PowerGREP treats `\10` through `\99` as a single-digit backreference followed by a literal digit.

`$1` through `$99` for single-digit and double-digit backreferences are also supported by PowerGREP. If there are not enough capturing groups in the regex for a double-digit backreference to be valid, then `$10` through `$99` are treated as a single-digit backreference followed by a literal digit.

Putting curly braces around the digit `${1}` isolates the digit from any literal digits that follow.

If your regular expression has named capturing groups, then you should use named backreferences to them in the replacement text. The regex `(?'name'group)` has one group called "name". You can reference this group with `${name}` in PowerGREP.

Backreferences to Non-Existent Capturing Groups

An invalid backreference is a reference to a number greater than the number of capturing groups in the regex or a reference to a name that does not exist in the regex. Such a backreference is a syntax error.

Backreferences to Non-Participating Capturing Groups

A non-participating capturing group is a group that did not participate in the match attempt at all. This is different from a group that matched an empty string. The group in `a(b?)c` always participates in the match. Its contents are optional but the group itself is not optional. The group in `a(b)?c` is optional. It participates when the regex matches `abc`, but not when the regex matches `ac`.

In PowerGREP, there is no difference between a backreference in the replacement string to a group that matched the empty string or a group that did not participate. Both are replaced with an empty string.

Backreference to The Highest-Numbered Group

In PowerGREP, `$+` inserts the text matched by the highest-numbered group that actually participated in the match. When `(a)(b)|(c)(d)` matches `ab`, `$+` is substituted with `b`. When the same regex matches `cd`, `$+` inserts `d`. `\+` does the same.

45. Match Context

PowerGREP supports special tokens in replacement strings that allow you to insert the subject string or the part of the subject string before or after the regex match. This can be useful when the replacement text syntax is used to collect search matches and their context instead of making replacements in the subject string.

In the replacement text, `` $` `` is substituted with the part of the subject string to the left of the regex match. `` \` `` works too.

You can use `$'` or `\'` to insert the part of the subject string to the right of the regex match.

In the replacement text, `$_` is substituted with the entire subject string. `\_` is just an escaped underscore.

46. Replacement Text Case Conversion

PowerGREP allows you to prefix the matched text token `\0` and the backreferences `\1` through `\99` with a letter that changes the case of the inserted text. `U` is for uppercase, `L` for lowercase, `I` for initial capitals (first letter of each word is uppercase, rest is lowercase), and `F` for first capital (first letter in the inserted text is uppercase, rest is lowercase). The letter only affects the case of the backreference that it is part of.

When the regex `(?i)(Helló) (Wórld)` matches `HeLlÓ WóRlD` the replacement text `\U1 \L2 \I0 \F0` becomes `HELLÓ wórld Helló Wórld Helló wórld`.

47. Replacement String Conditionals

Replacement string conditionals allow you to use one replacement when a particular capturing group participated in the match and another replacement when that capturing group did not participate in the match. PowerGREP supports the syntax invented by Boost as well as the syntax invented by PCRE2.

Boost Replacement String Conditionals

The syntax borrowed from Boost is `(?1matched:unmatched)` where `1` is a number between 1 and 99 referencing a numbered capturing group. `matched` is used as the replacement for matches in which the capturing group participated. `unmatched` is used for matches in which the group did not participate. The colon `:` delimits the two parts. If you want a literal colon in the `matched` part, then you need to escape it with a backslash. If you want a literal closing parenthesis anywhere in the conditional, then you need to escape that with a backslash too.

The parentheses delimit the conditional from the remainder of the replacement string. `start(?1matched:unmatched)finish` replaces with `startmatchedfinish` when the group participates and with `startunmatchedfinish` when it doesn't.

The `matched` and `unmatched` parts can be blank. You can omit the colon if the `unmatched` part is blank. So `(?1matched:)` and `(?1matched)` replace with `matched` when the group participates. They replace the match with nothing when the group does not participate.

You can use the full replacement string syntax in `matched` and `unmatched`. This means you can nest conditionals inside other conditionals. So `(?1one(?2two):(?2two:none))` replaces with `onetwo` when both groups participate, with `one` or `two` when group 1 or 2 participates and the other doesn't, and with `none` when neither group participates. With Boost `?1one(?2two):?2two:none` does exactly the same but omits parentheses that aren't needed.

PowerGREP treats conditionals that reference non-existing capturing groups as an error. If there are two digits after the question mark but not enough capturing groups for a two-digit conditional to be valid, then only the first digit is used for the conditional and the second digit is a literal. So when there are less than 12 capturing groups in the regex, `(?12matched)` replaces with `2matched` when capturing group 1 participates in the match.

You can avoid the ambiguity between single digit and double digit conditionals by placing curly braces around the number. `(?{1}1:0)` replaces with `1` when group 1 participates and with `0` when it doesn't, even if there are 11 or more capturing groups in the regex. `(?{12}twelve:not twelve)` is always a conditional that references group 12, even if there are fewer than 12 groups in the regex (which would make the conditional invalid).

The syntax with curly braces also allows you to reference named capturing groups by their names. `(?{name}matched:unmatched)` replaces with `matched` when the group "name" participates in the match and with `unmatched` when it doesn't. If the group does not exist, PowerGREP treats the conditionals as an error.

PCRE2 Replacement String Conditional

The syntax borrowed from PCRE2 is `${1:+matched:unmatched}` where `1` is a number between 1 and 99 referencing a numbered capturing group. If your regex contains named capturing groups then you can reference them in a conditional by their name: `${name:+matched:unmatched}`.

`matched` is used as the replacement for matches in which the capturing group participated. `unmatched` is used for matches in which the group did not participate. `:+` delimits the group number or name from the first part of the conditional. The second colon delimits the two parts. If you want a literal colon in the `matched` part, then you need to escape it with a backslash. If you want a literal closing curly brace anywhere in the conditional, then you need to escape that with a backslash too. Plus signs have no special meaning beyond the `:+` that starts the conditional, so they don't need to be escaped.

You can use the full replacement string syntax in `matched` and `unmatched`. This means you can nest conditionals inside other conditionals. So `${1:+one${2:+two}:${2:+two:none}}` replaces with `onetwo` when both groups participate, with `one` or `two` when group 1 or 2 participates and the other doesn't, and with `none` when neither group participates.

`${1:-unmatched}` and `${name:-unmatched}` are shorthands for `${1:+${1}:unmatched}` and `${name:+${name}:unmatched}`. They insert the text captured by the group if it participated in the match. They insert `unmatched` if the group did not participate. When using this syntax, `:-` delimits the group number or name from the contents of the conditional. The conditional has only one part in which colons and minus signs have no special meaning.

Escaping Question Marks, Colons, Parentheses, and Curly Braces

As explained above, you need to use backslashes to escape colons that you want to use as literals when used in the `matched` part of the conditional. You also need to escape literal closing parentheses (Boost) or curly braces (PCRE2) with backslashes inside conditionals.

You can escape colons, parentheses, curly braces, and even question marks with backslashes to make sure they are interpreted as literals anywhere in the replacement string. But generally there is no need to.

The colon does not have any special meaning in the `unmatched` part or outside conditionals. So you don't need to escape it there. The question mark does not have any special meaning if it is not followed by a digit or a curly brace. So you only need to escape question marks with backslashes if you want to use a literal question mark followed by a literal digit or curly brace as the replacement.

In PowerGREP, opening parentheses are part of the syntax for conditionals. The first unescaped closing parenthesis that follows it then ends the conditional. All other unescaped opening and closing parentheses are literals.

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.

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 an IP Address.

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-length 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 Numeric Ranges with a Regular Expression

Since regular expressions deal with text rather than with numbers, matching a number in a given range takes a little extra care. You can't just write `[0-255]` to match a number between 0 and 255. Though a valid regex, it matches something entirely different. `[0-255]` is a character class with three elements: the character range 0-2, the character 5 and the character 5 (again). This character class matches a single digit 0, 1, 2 or 5, just like `[0125]`.

Since regular expressions work with text, a regular expression engine treats `0` as a single character, and `255` as three characters. To match all characters from 0 to 255, we'll need a regex that matches between one and three characters.

The regex `[0-9]` matches single-digit numbers 0 to 9. `[1-9][0-9]` matches double-digit numbers 10 to 99. That's the easy part.

Matching the three-digit numbers is a little more complicated, since we need to exclude numbers 256 through 999. `1[0-9][0-9]` takes care of 100 to 199. `2[0-4][0-9]` matches 200 through 249. Finally, `25[0-5]` adds 250 till 255.

As you can see, you need to split up the numeric range in ranges with the same number of digits, and each of those ranges that allow the same variation for each digit. In the 3-digit range in our example, numbers starting with 1 allow all 10 digits for the following two digits, while numbers starting with 2 restrict the digits that are allowed to follow.

Putting this all together using alternation we get: `[0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5]`. This matches the numbers we want, with one caveat: regular expression searches usually allow partial matches, so our regex would match `123` in `12345`. There are two solutions to this.

If you're searching for these numbers in a larger document or input string, use word boundaries to require a non-word character (or no character at all) to precede and to follow any valid match. The regex then becomes `\b([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\b`. Since the alternation operator has the lowest precedence of all, the parentheses are required to group the alternatives together. This way the regex engine will try to match the first word boundary, then try all the alternatives, and then try to match the second word boundary after the numbers it matched. Regular expression engines consider all alphanumeric characters, as well as the underscore, as word characters.

If you're using the regular expression to validate input, you'll probably want to check that the entire input consists of a valid number. To do this, replace the word boundaries with anchors to match the start and end of the string: `^([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])$`.

Here are a few more common ranges that you may want to match:

- 000..255: `^([01][0-9][0-9]|2[0-4][0-9]|25[0-5])$`
- 0 or 000..255: `^([01]?[0-9]?[0-9]|2[0-4][0-9]|25[0-5])$`
- 0 or 000..127: `^(0?[0-9]?[0-9]|1[01][0-9]|12[0-7])$`
- 0..999: `^([0-9]|[1-9][0-9]|[1-9][0-9][0-9])$`
- 000..999: `^[0-9]{3}$`
- 0 or 000..999: `^[0-9]{1,3}$`
- 1..999: `^([1-9]|[1-9][0-9]|[1-9][0-9][0-9])$`

- 001..999: `^(00[1-9]|0[1-9][0-9]|[1-9][0-9][0-9])$`
- 1 or 001..999: `^(0{0,2}[1-9]|0?[1-9][0-9]|[1-9][0-9][0-9])$`
- 0 or 00..59: `^[0-5]?[0-9]$`
- 0 or 000..366: `^(0?[0-9]?[0-9]|[1-2][0-9][0-9]|3[0-5][0-9]|36[0-6])$`

3. Matching Floating Point Numbers with a Regular Expression

This example shows 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. 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 considers a sign by itself or a dot by itself as a valid floating point number. In fact, it even considers an empty string as a valid floating point number. If you tried to use this regex to find floating point numbers in a file, you'd get a zero-length match at every position in the string where no floating point number occurs.

Not escaping the dot is also a common mistake. A dot that is not escaped matches any character, including a dot. If we had not escaped the dot, both `4.4` and `4X4` would be considered a floating point numbers.

When creating a regular expression, it is more important to consider what it should *not* match, than what it should. The above regex indeed matches a proper floating point number, because the regex engine is greedy. But it 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 matches 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 that is followed by one or more digits (an integer).

This is a far better definition. Any match must include at least one digit. 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 RegexBuddy's library.

4. 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,}\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,}$`.

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 have to turn on the case insensitive matching option.

Trade-Offs in Validating Email Addresses

Before ICANN made it possible for any well-funded company to create their own top-level domains, the longest top-level domains were the rarely used `.museum` and `.travel` which are 6 letters long. The most common top-level domains were 2 letters long for country-specific domains, and 3 or 4 letters long for general-purpose domains like `.com` and `.info`. A lot of regexes for validating email addresses you’ll find in various regex tutorials and references still assume the top-level domain to be fairly short. Older editions of this regex tutorial mentioned `\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b` as the regex for email addresses in its introduction. There’s only one little difference between this regex and the one at the top of this page. The `4` at the end of the regex restricts the top-level domain to 4 characters. If you use this regex with anchors to validate the email address entered on your order form, `fabio@disapproved.solutions` has to do his shopping elsewhere. Yes, the `.solutions` TLD exists and when I write this, `disaproved.solutions` can be yours for $16.88 per year.

If you want to be more strict than `[A-Z]{2,}` for the top-level domain, `^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,63}$` is as far as you can practically go. Each part of a domain name can be no longer than 63 characters. There are no single-digit top-level domains and none contain digits. It doesn’t look like ICANN will approve such domains either.

Email addresses can be on servers on a subdomain as in `john@server.department.company.com`. All of the above regexes match this email address, because I included a dot in the character class after the @ symbol. But the above regexes 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. `^[A-Z0-9._%+-]+@(?:[A-Z0-9-]+\.)+[A-Z]{2,}$` matches `john@server.department.company.com` but not `john@aol...com`.

If you want to avoid your system choking on arbitrarily large input, you can replace the infinite quantifiers with finite ones. `^[A-Z0-9._%+-]{1,64}@(?:[A-Z0-9-]{1,63}\.){1,125}[A-Z]{2,63}$` takes into account that the local part (before the @) is limited to 64 characters and that each part of the domain name is limited to 63 characters. There's no direct limit on the number of subdomains. But the maximum length of an email address that can be handled by SMTP is 254 characters. So with a single-character local part, a two-letter top-level domain and single-character sub-domains, 125 is the maximum number of sub-domains.

The previous regex does not actually limit email addresses to 254 characters. If each part is at its maximum length, the regex can match strings up to 8129 characters in length. You can reduce that by lowering the number of allowed sub-domains from 125 to something more realistic like 8. I've never seen an email address with more than 4 subdomains. If you want to enforce the 254 character limit, the best solution is to check the length of the input string before you even use a regex. Though this requires a few lines of procedural code, checking the length of a string is near-instantaneous. If you can only use regexes, `^[A-Z0-9._%+-]{6,254}$` can be used as a first pass to make sure the string doesn't contain invalid characters and isn't too short or too long. If you need to do everything with one regex, you'll need a regex flavor that supports lookahead. The regular expression `^(?=[A-Z0-9._%+-]{6,254}$)[A-Z0-9._%+-]{1,64}@(?:[A-Z0-9-]{1,63}\.){1,8}[A-Z]{2,63}$` uses a lookahead to first check that the string doesn't contain invalid characters and isn't too short or too long. When the lookahead succeeds, the remainder of the regex makes a second pass over the string to check for proper placement of the @ sign and the dots.

All of these regexes allow the characters `._%+-` anywhere in the local part. You can force the local part to begin with a letter by using `^[A-Z0-9][A-Z0-9._%+-]{0,63}` instead of `^[A-Z0-9._%+-]{1,64}` for the local part: `^[A-Z0-9][A-Z0-9._%+-]{0,63}@(?:[A-Z0-9-]{1,63}\.){1,125}[A-Z]{2,63}$`. When using lookahead to check the overall length of the address, the first character can be checked in the lookahead. We don't need to repeat the initial character check when checking the length of the local part. This regex is too long to fit the width of the page, so let's turn on free-spacing mode:

```
^(?=[A-Z0-9][A-Z0-9@._%+-]{5,253}$)
[A-Z0-9._%+-]{1,64}@(?:[A-Z0-9-]{1,63}\.){1,8}[A-Z]{2,63}$
```

Domain names can contain hyphens. But they cannot begin or end with a hyphen. `[A-Z0-9](?:[A-Z0-9-]{0,62}[A-Z0-9])?` matches a domain name between 1 and 63 characters long that starts and ends with a letter or digit. The non-capturing group makes the middle of the domain and the final letter or digit optional as a whole to ensure that we allow single-character domains while at the same time ensuring that domains with two or more characters do not end with a hyphen. The overall regex starts to get quite complicated:

```
^[A-Z0-9][A-Z0-9._%+-]{0,63}@
(?:[A-Z0-9](?:[A-Z0-9-]{0,62}[A-Z0-9])?\.){1,8}[A-Z]{2,63}$
```

Domain names cannot contain consecutive hyphens. `[A-Z0-9]+(?:-[A-Z0-9]+)*` matches a domain name that starts and ends with a letter or digit and that contains any number of non-consecutive hyphens. This is the most efficient way. This regex does not do any backtracking to match a valid domain name. It matches all letters and digits at the start of the domain name. If there are no hyphens, the optional group that follows fails immediately. If there are hyphens, the group matches each hyphen followed by all letters and digits up to the next hyphen or the end of the domain name. We can't enforce the maximum length when hyphens must be paired with a letter or digit, but letters and digits can stand on their own. But we can use the lookahead technique that we used to enforce the overall length of the email address to enforce the length of the domain name while disallowing consecutive hyphens: `(?=[A-Z0-9-]{1,63}\.)[A-Z0-9]+(?:-[A-Z0-9]+)*`. Notice that the lookahead also checks for the dot that must appear after the domain name when it is fully qualified in an email address. This is important. Without checking for the dot, the lookahead would accept longer domain names. Since the lookahead does not consume the text it matches, the dot is not included in the overall match of this regex. When we put this regex into the overall regex for email addresses, the dot will be matched as it was in the previous regexes:

```
^[A-Z0-9][A-Z0-9._%+-]{0,63}@
(?:(?=[A-Z0-9-]{1,63}\.)[A-Z0-9]+(?:-[A-Z0-9]+)*\.){1,8}[A-Z]{2,63}$
```

If we include the lookahead to check the overall length, our regex makes two passes over the local part, and three passes over the domain names to validate everything:

```
^(?=[A-Z0-9][A-Z0-9@._%+-]{5,253}$)[A-Z0-9._%+-]{1,64}@
(?:(?=[A-Z0-9-]{1,63}\.)[A-Z0-9]+(?:-[A-Z0-9]+)*\.){1,8}[A-Z]{2,63}$
```

On a modern PC or server this regex will perform just fine when validating a single 254-character email address. Rejecting longer input would even be faster because the regex will fail when the lookahead fails during first pass. But I wouldn't recommend using a regex as complex as this to search for email addresses through a large archive of documents or correspondence. You're better off using the simple regex at the top of this page to quickly gather everything that looks like an email address. Deduplicate the results and then use a stricter regex if you want to further filter out invalid addresses.

And speaking of backtracking, none of the regexes on this page do any backtracking to match valid email addresses. But particularly the latter ones may do a fair bit of backtracking on something that's not quite a valid email address. If your regex flavor supports possessive quantifiers, you can eliminate all backtracking by making all quantifiers possessive. Because no backtracking is needed to find matches, doing this does not change what is matched by these regexes. It only allows them to fail faster when the input is not a valid email address. Our simplest regex then becomes `^[A-Z0-9._%+-]++@[A-Z0-9.-]++\.[A-Z]{2,}+$` with an extra + after each quantifier. We can do the same with our most complex regex:

```
^(?=[A-Z0-9][A-Z0-9@._%+-]{5,253}+$)[A-Z0-9._%+-]{1,64}+@
(?:(?=[A-Z0-9-]{1,63}+\.)[A-Z0-9]++(?:-[A-Z0-9]++)*+\.){1,8}+[A-Z]{2,63}+$
```

An important trade-off in all these regexes is that they only allow English letters, digits, and the most commonly used 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. Blindly inserting this email address into an SQL query, for example, will at best cause it to fail when strings are delimited with single quotes and at worst open your site up to SQL injection attacks.

And of course, it's been many years already that domain names can include non-English characters. But most software still sticks to the 37 characters Western programmers are used to. Supporting internationalized domains opens up a whole can of worms of how the non-ASCII characters should be encoded. So if you use any of the regexes on this page, anyone with an @ทีเอชนิค.ไทย address will be out of luck. But perhaps it is telling that http://ทีเอชนิค.ไทย simply redirects to http://thnic.co.th even though they're in the business of selling .ไทย domains.

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 because it turned out to be too broad or too narrow? 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. 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. If you really need to be sure an email address is valid, you'll need to send an email to it that contains a code or link for the recipient to perform a second authentication step. And if you're doing that, then there is little point in using a regex that may reject valid email addresses.

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 5322

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 5322. It describes the syntax that valid email addresses must adhere to. You can (but you shouldn't—read on) implement it with the following regular expression. RFC 5322 leaves the domain name part open to implementation-specific choices that won't work on the Internet today. The regex implements the "preferred" syntax from RFC 1035 which is one of the recommendations in RFC 5322:

```
\A(?:[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])?
  |  \[(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
       (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?|[a-z0-9-]*[a-z0-9]:
          (?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]
          |  \\[\x01-\x09\x0b\x0c\x0e-\x7f])+)
     \])\z
```

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 is overly broad. Your application may not be able to handle all email addresses this regex allows. Domain-specific routing addresses can contain non-printable ASCII control characters, which can cause trouble if your application needs to display addresses. Not all applications support the syntax for the local part using double quotes or square brackets. In fact, RFC 5322 itself marks the notation using square brackets as obsolete.

We get a more practical implementation of RFC 5322 if we omit IP addresses, domain-specific addresses, the syntax using double quotes and square brackets. It will still match 99.99% of all email addresses in actual use today.

```
\A[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*@
(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\z
```

Neither of these regexes enforce length limits on the overall email address or the local part or the domain names. RFC 5322 does not specify any length limitations. Those stem from limitations in other protocols like the SMTP protocol for actually sending email. RFC 1035 does state that domains must be 63 characters or less, but does not include that in its syntax specification. The reason is that a true regular language cannot enforce a length limit and disallow consecutive hyphens at the same time. But modern regex flavors aren't truly regular, so we can add length limit checks using lookahead like we did before:

```
\A(?=[a-z0-9@.!#$%&'*+/=?^_`{|}~-]{6,254}\z)
 (?=[a-z0-9.!#$%&'*+/=?^_`{|}~-]{1,64}@)
 [a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*
@ (?:(?=[a-z0-9-]{1,63}\.)[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+
  (?=[a-z0-9-]{1,63}\z)[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\z
```

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.

5. How to Find or Validate an IP Address

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. If your regex flavor supports Unicode, it may even match `١٢٣.१२३.൧൨൩.๑๒๓`. Whether this is a problem depends on the files or data you intend to apply the regex to.

Restricting and Capturing The Four IP Address Numbers

To restrict all 4 numbers in the IP address to 0..255, you can use the following regex. It 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. Free-spacing mode allows this to fit the width of the page.

```
\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
```

The above regex allows one leading zero for numbers 10 to 99 and up to two leading zeros for numbers 0 to 9. Strictly speaking, IP addresses with leading zeros imply octal notation. So you may want to disallow leading zeros. This requires a slightly longer regex:

```
\b(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.
  (25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.
  (25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.
  (25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\b
```

Restricting The Four IP Address Numbers Without Capturing Them

If you don't need access to the individual numbers, you can shorten above 3 regexes with a quantifier to:

```
\b(?:\d{1,3}\.){3}\d{1,3}\b
```

```
\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
```

```
\b(?:(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.){3}
  (?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\b
```

Checking User Input

The above regexes use word boundaries to make sure the first and last number in the IP address aren't part of a longer sequence of alphanumeric characters. These regexes are appropriate for finding IP addresses in longer strings.

If you want to validate user input by making sure a string consists of nothing but an IP address then you need to replace the word boundaries with start-of-string and end-of-string anchors. You can use the dedicated anchors `\A` and `\z` if your regex flavor supports them:

```
\A(?:(?: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]?)\z
```

If not, you'll have to use `^` and `$` and make sure that the option for them to match at line breaks is off:

```
^(?:(?: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]?)$
```

6. 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 1900-01-01 through 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 parentheses 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. Parentheses 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 parentheses 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.

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.

7. Replacing Numerical Dates with Textual Dates

This example shows how you can replace numerical dates from 1/1/50 or 01/01/50 through 12/31/49 with their textual equivalents from January 1st, 1950 through December 31st, 2049. This is only possible with a single regular expression if you can vary the replacement based on what was matched. One way to do this is to build each replacement in procedural code. This example shows how you can do it using replacement string conditionals.

To be able to use replacement string conditionals, the regular expression needs a separate capturing group for each part of the match that needs a different replacement. Each month needs to be replaced with its own name, so we need a separate capturing group to match each month number. Cardinal numbers ending with 1, 2, and 3 have unique suffixes. So we need four groups to match day numbers ending with 1, 2, 3, or another digit. We're assuming year numbers 50 to 99 to be 1950 to 1999, and year numbers 00 to 49 to be 2000 to 2049. So we need two more groups to match each half century.

Putting this all together results in a rather long regular expression. Free-spacing helps to keep it readable. The structure of the regex is the same as what you would use for matching valid dates. It's only more long-winded because we need 12 alternatives to match the month, 4 alternatives to match the day, and 2 alternatives to match the year.

```
\b
(?: # Month
   (?'jan'0?1)|(?'feb'0?2)|(?'mar'0?3)|(?'apr'0?4)|(?'may'0?5)|(?'jun'0?6)
  |(?'jul'0?7)|(?'aug'0?8)|(?'sep'0?9)|(?'oct'10)|(?'nov'11)|(?'dec'12)
  ) /
0?(?: # Day
   (?'1st'[123]?1)|(?'2nd'[12]?2)|(?'3rd'[12]?3)|(?'nth'30|[12]?[4-90])
  ) /
(?: # Year
   (?'19xx'[5-9][0-9])|(?'20xx'[0-4][0-9])
  )
\b
```

The replacement string will use backreferences to reinsert the date numbers. Since we want to omit leading zeros from the replacements, we placed `0?` outside the capturing groups for date numbers. This means that our regex also allows leading zeros for days 10 to 31. Since our goal is to replace dates rather than validate them, we can live with this. Otherwise, we would need two sets of four alternatives to match the day of the month. One set for single digit days, and one set for double digit days.

Unfortunately, free-spacing does not work with replacement strings. So the replacement consists of one very long line. It is broken into multiple lines here to fit te width of the page. This is the replacement using Boost syntax:

```
(?{jan}January)(?{feb}February)(?{mar}March)(?{apr}April)(?{may}May)(?{jun}June)
(?{jul}July)(?{aug}August)(?{sep}September)(?{oct}October)(?{nov}November)
(?{dec}December) (?{1st}${1st}st)(?{2nd}${2nd}nd)(?{3rd}${3rd}rd)(?{nth}${nth}th)
, (?{19xx}19${19xx})(?{20xx}20${20xx})
```

This is the replacement using PCRE2 syntax:

```
${jan:+January}${feb:+February}${mar:+March}${apr:+April}${may:+May}${jun:+June}
${jul:+July}${aug:+August}${sep:+September}${oct:+October}${nov:+November}
```

```
${dec:+December} ${1st:+${1st}st}${2nd:+${2nd}nd}${3rd:+${3rd}rd}${nth:+${nth}th}
, ${19xx:+19${19xx}}${20xx:+20${20xx}}
```

First we have 12 conditionals that reference the 12 capturing groups for the months. Each conditional inserts the month’s name when their group participates. They insert nothing when their group does not participate. Since only one of these groups participates in any match, only one of these conditionals actually inserts anything into the replacement.

Then we have a literal space and 4 more conditionals that reference the 4 capturing groups for the days. When the group participates, the conditional uses a backreference to the same group to reinsert the day number matched by the group. The backreference is followed by a literal suffix.

Finally, we have a literal comma, a literal space, and 2 more conditionals for the year. The conditionals again use literal text and a backreference to expand the year from 2 to 4 digits.

8. 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})?$` All Visa card numbers start with a 4. New cards have 16 digits. Old cards have 13.
- MasterCard: `^(?:5[1-5][0-9]{2}|222[1-9]|22[3-9][0-9]|2[3-6][0-9]{2}|27[01][0-9]|2720)[0-9]{12}$` MasterCard numbers either start with the numbers 51 through 55 or with the numbers 2221 through 2720. 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 using alternation. A non-capturing group puts the anchors outside the alternation. Free-spacing allows for comments and for the regex to fit the width of this page.

```
^(?:4[0-9]{12}(?:[0-9]{3})?          # Visa
 |  (?:5[1-5][0-9]{2}                # MasterCard
     | 222[1-9]|22[3-9][0-9]|2[3-6][0-9]{2}|27[01][0-9]|2720)[0-9]{12}
 |  3[47][0-9]{13}                   # American Express
 |  3(?:0[0-5]|[68][0-9])[0-9]{11}   # Diners Club
 |  6(?:011|5[0-9]{2})[0-9]{12}      # Discover
 |  (?:2131|1800|35\d{3})\d{11}      # JCB
)$
```

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 that there's money in the account. The benefit or 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 as in `\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 entering the numbers may have different ideas yet.

9. 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 parentheses.

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).*$` matches a complete line of text that contains *all* of the words "one", "two" and "three". Again, the anchors must match at the start and end of a line and the dot must not match line breaks. Because of the caret, and the fact that lookahead is zero-length, all of the three lookaheads are attempted at the start of the each line. Each lookahead will match any piece of text on a single line (`.*?`) followed by one of the words. All three must match successfully for the entire regex to match. Note that instead of words like `\bword\b`, you can put any regular expression, no matter how complex, inside the lookahead. Finally, `.*$` 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-length 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.

10. 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)+$` and replacing with `\1`. For this to work, the anchors need to match before and after line breaks (and not just at the start and the end of the file or string), and the dot must *not* match newlines.

Here is how this works. The caret will match only at the start of a line. So the regex engine will only attempt to match the remainder of the regex there. The dot and star combination simply matches an entire line, whatever its contents, if any. The parentheses store the matched line into the first backreference.

Next we will match the line separator. I put the question mark into `\r?\n` to make this regex work with both Windows (\r\n) and UNIX (\n) text files. So up to this point we matched a line and the following line break.

Now we need to check if this combination is followed by a duplicate of that same line. We do this simply with `\1`. This is the first backreference which holds the line we matched. The backreference will match that very same text.

If the backreference fails to match, the regex match and the backreference are discarded, and the regex engine tries again at the start of the next line. If the backreference succeeds, the plus symbol in the regular expression will try to match additional copies of the line. Finally, the dollar symbol forces the regex engine to check if the text matched by the backreference is a complete line. We already know the text matched by the backreference is preceded by a line break (matched by `\r?\n`). Therefore, we now check if it is also followed by a line break or if it is at the end of the file using the dollar sign.

The entire match becomes `line\nline` (or `line\nline\nline` etc.). Because we are doing a search and replace, the line, its duplicates, and the line breaks in between them, are all deleted from the file. Since we want to keep the original line, but not the duplicates, we use `\1` as the replacement text to put the original line back in.

Removing Duplicate Items From a String

We can generalize the above example to `afterseparator(item)(separator\1)+beforeseparator`, where `afterseparator` and `beforeseparator` are zero-length. So if you want to remove consecutive duplicates from a comma-delimited list, you could use `(?<=,|^)([^,]*)(,\1)+(?=,|$)`.

The positive lookbehind `(?<=,|^)` forces the regex engine to start matching at the start of the string or after a comma. `([^,]*)` captures the item. `(,\1)+` matches consecutive duplicate items. Finally, the positive lookahead `(?=,|$)` checks if the duplicate items are complete items by checking for a comma or the end of the string.

11. Example Regexes to Match Common Programming Language Constructs

Regular expressions are very useful to manipulate source code in a text editor or in a regex-based text processing tool. Most programming languages use similar constructs like keywords, comments and strings. But often there are subtle differences that make it tricky to use the correct regex. When picking a regex from the list of examples below, be sure to read the description with each regex to make sure you are picking the correct one.

Unless otherwise indicated, all examples below assume that the dot does *not* match newlines and that the caret and dollar *do* match at embedded line breaks. In many programming languages, this means that single line mode must be off, and multi line mode must be on.

When used by themselves, these regular expressions may not have the intended result. If a comment appears inside a string, the comment regex will consider the text inside the string as a comment. The string regex will also match strings inside comments. The solution is to use more than one regular expression and to combine those into a simple parser, like in this pseudo-code:

```
GlobalStartPosition := 0;
while GlobalStartPosition < LengthOfText do
  GlobalMatchPosition := LengthOfText;
  MatchedRegEx := NULL;
  foreach RegEx in RegExList do
    RegEx.StartPosition := GlobalStartPosition;
    if RegEx.Match and RegEx.MatchPosition < GlobalMatchPosition then
      MatchedRegEx := RegEx;
      GlobalMatchPosition := RegEx.MatchPosition;
    endif
  endforeach
  if MatchedRegEx <> NULL then
    // At this point, MatchedRegEx indicates which regex matched
    // and you can do whatever processing you want depending on
    // which regex actually matched.
  endif
  GlobalStartPosition := GlobalMatchPosition;
endwhile
```

If you put a regex matching a comment and a regex matching a string in `RegExList`, then you can be sure that the comment regex will not match comments inside strings, and vice versa. Inside the loop you can then process the match according to whether it is a comment or a string.

An alternative solution is to combine regexes: `(comment)|(string)`. The alternation has the same effect as the code snipped above. Iterate over all the matches of this regex. Inside the loop, check which capturing group found the regex match. If group 1 matched, you have a comment. If group two matched, you have a string. Then process the match according to that.

You can use this technique to build a full parser. Add regular expressions for all lexical elements in the language or file format you want to parse. Inside the loop, keep track of what was matched so that the following matches can be processed according to their context. For example, if curly braces need to be balanced, increment a counter when an opening brace is matched, and decrement it when a closing brace is matched. Raise an error if the counter goes negative at any point or if it is nonzero when the end of the file is reached.

Comments

`#.*$` matches a single-line comment starting with a `#` and continuing until the end of the line. Similarly, `//.*$` matches a single-line comment starting with `//`.

If the comment must appear at the start of the line, use `^#.*$`. If only whitespace is allowed between the start of the line and the comment, use `^\s*#.*$`. Compiler directives or pragmas in C can be matched this way. Note that in this last example, any leading whitespace will be part of the regex match. Use capturing parentheses to separate the whitespace and the comment.

`/\*.*?\*/` matches a C-style multi-line comment if you turn on the option for the dot to match newlines. The general syntax is `begin.*?end`. C-style comments do not allow nesting. If the "begin" part appears inside the comment, it is ignored. As soon as the "end" part if found, the comment is closed.

If your programming language allows nested comments, there is no straightforward way to match them using a regular expression, since regular expressions cannot count. Additional logic is required.

Strings

`"[^"\r\n]*"` 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. `"[^"\\]*(?:\\.[^"\\]*)*"` 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.

12. 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`.

13. 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 `xxxxxxxxxxy`. 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.

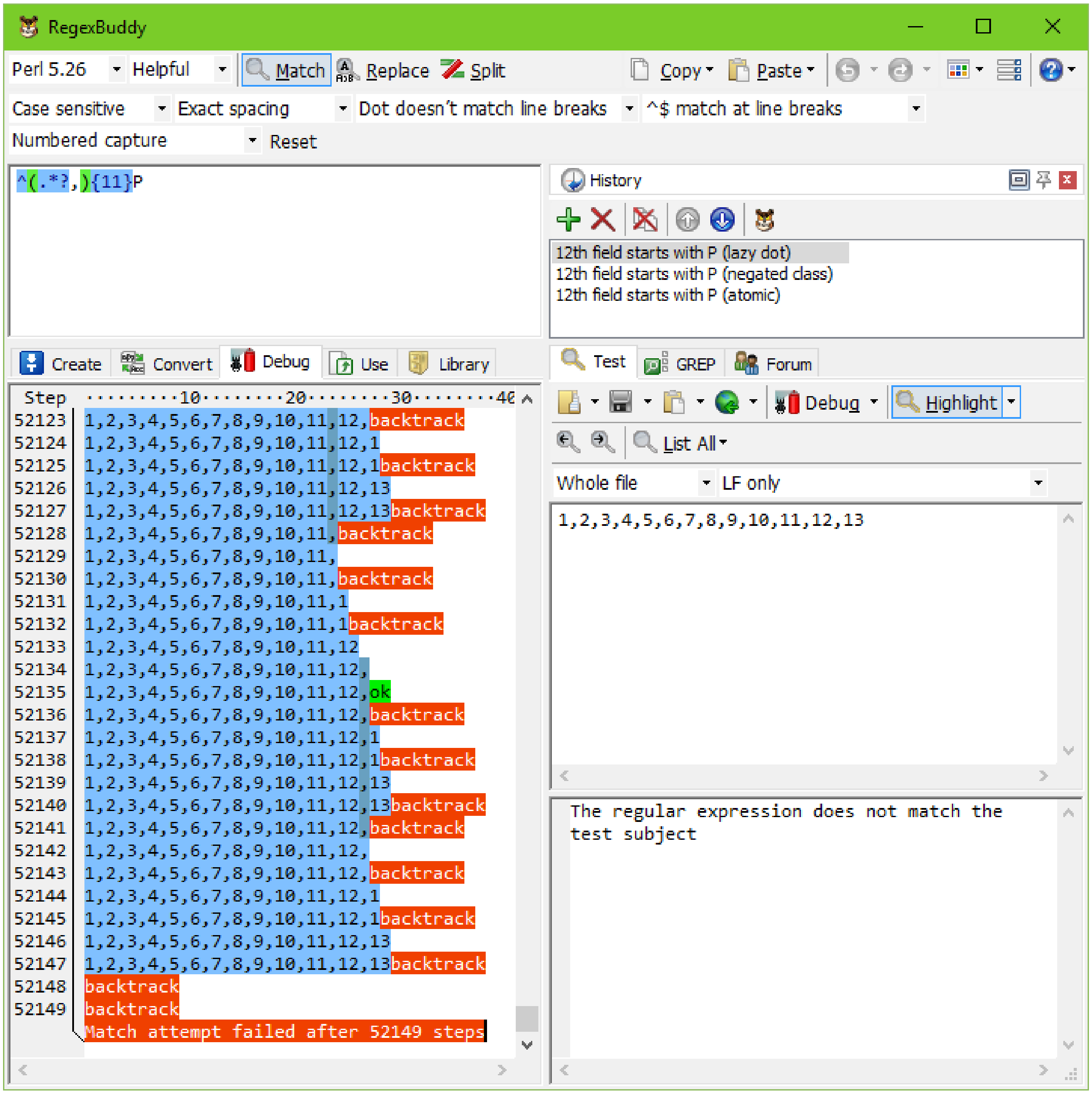
You get the idea: the possible number of combinations that the regex engine will try for each line where the 12th field does not start with a P is huge. All this would take a long time if you ran this regex on a large CSV file where most rows don't have a P at the start of the 12th field.

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



Perl 5.26 · Helpful

Case sensitive · Exact spacing · Dot doesn't match line breaks · ^$ match at line breaks

Numbered capture

```
^(.*?,){11}P
```

12th field starts with P (lazy dot)
12th field starts with P (negated class)
12th field starts with P (atomic)

```
 Step  ·········10········20········30········40
52123  1,2,3,4,5,6,7,8,9,10,11,12,backtrack
52124  1,2,3,4,5,6,7,8,9,10,11,12,1
52125  1,2,3,4,5,6,7,8,9,10,11,12,1backtrack
52126  1,2,3,4,5,6,7,8,9,10,11,12,13
52127  1,2,3,4,5,6,7,8,9,10,11,12,13backtrack
52128  1,2,3,4,5,6,7,8,9,10,11,backtrack
52129  1,2,3,4,5,6,7,8,9,10,11,
52130  1,2,3,4,5,6,7,8,9,10,11,backtrack
52131  1,2,3,4,5,6,7,8,9,10,11,1
52132  1,2,3,4,5,6,7,8,9,10,11,1backtrack
52133  1,2,3,4,5,6,7,8,9,10,11,12
52134  1,2,3,4,5,6,7,8,9,10,11,12,
52135  1,2,3,4,5,6,7,8,9,10,11,12,ok
52136  1,2,3,4,5,6,7,8,9,10,11,12,backtrack
52137  1,2,3,4,5,6,7,8,9,10,11,12,1
52138  1,2,3,4,5,6,7,8,9,10,11,12,1backtrack
52139  1,2,3,4,5,6,7,8,9,10,11,12,13
52140  1,2,3,4,5,6,7,8,9,10,11,12,13backtrack
52141  1,2,3,4,5,6,7,8,9,10,11,12,backtrack
52142  1,2,3,4,5,6,7,8,9,10,11,12,
52143  1,2,3,4,5,6,7,8,9,10,11,12,backtrack
52144  1,2,3,4,5,6,7,8,9,10,11,12,1
52145  1,2,3,4,5,6,7,8,9,10,11,12,1backtrack
52146  1,2,3,4,5,6,7,8,9,10,11,12,13
52147  1,2,3,4,5,6,7,8,9,10,11,12,13backtrack
52148  backtrack
52149  backtrack
       Match attempt failed after 52149 steps
```

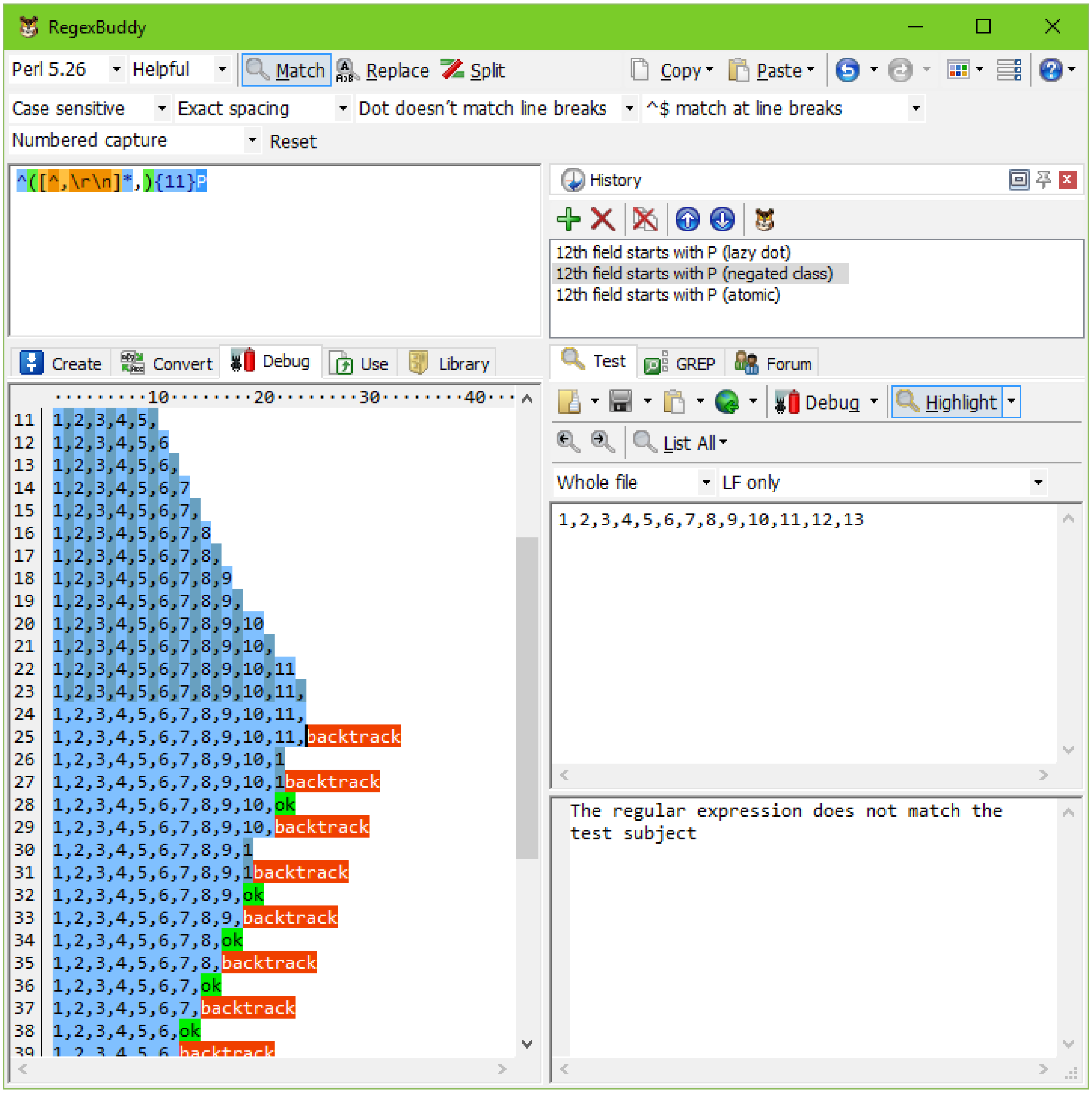
Whole file · LF only

```
1,2,3,4,5,6,7,8,9,10,11,12,13
```

```
The regular expression does not match the
test subject
```

If you try this example with RegexBuddy's debugger, you will see that the original regex `^(.*?,){11}P` needs 25,593 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 52,149. 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 52 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 11th field. The reason is the regex fails almost immediately when it discovers the 12th field doesn't start with a P. It backtracks the 11 iterations of the group without expanding again.



Perl 5.26 | Helpful

Case sensitive | Exact spacing | Dot doesn't match line breaks | ^$ match at line breaks

Numbered capture

```
^([^,\r\n]*,){11}P
```

12th field starts with P (lazy dot)
12th field starts with P (negated class)
12th field starts with P (atomic)

Whole file | LF only

```
1,2,3,4,5,6,7,8,9,10,11,12,13
```

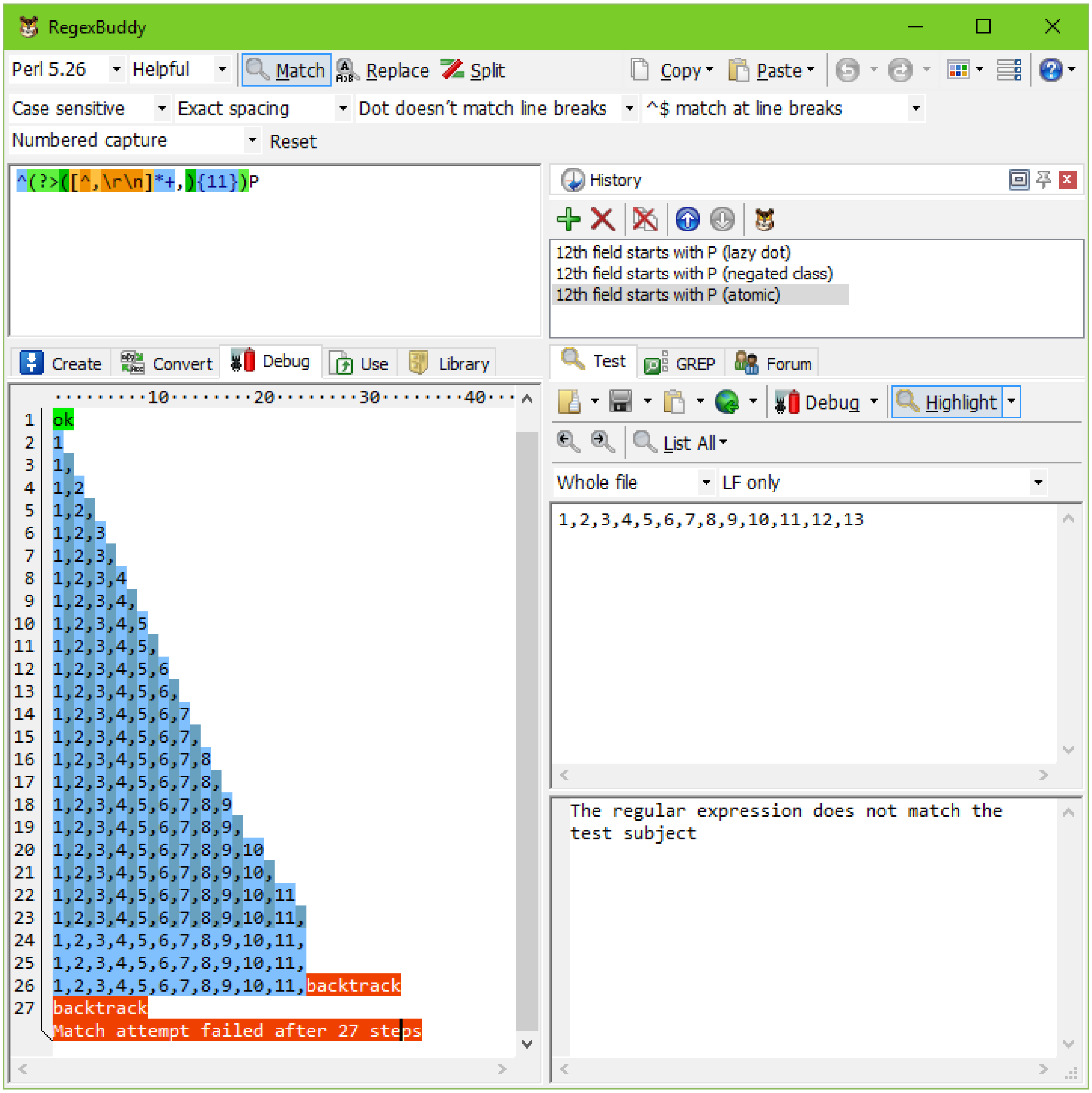
```
11 1,2,3,4,5,
12 1,2,3,4,5,6
13 1,2,3,4,5,6,
14 1,2,3,4,5,6,7
15 1,2,3,4,5,6,7,
16 1,2,3,4,5,6,7,8
17 1,2,3,4,5,6,7,8,
18 1,2,3,4,5,6,7,8,9
19 1,2,3,4,5,6,7,8,9,
20 1,2,3,4,5,6,7,8,9,10
21 1,2,3,4,5,6,7,8,9,10,
22 1,2,3,4,5,6,7,8,9,10,11
23 1,2,3,4,5,6,7,8,9,10,11,
24 1,2,3,4,5,6,7,8,9,10,11,
25 1,2,3,4,5,6,7,8,9,10,11,backtrack
26 1,2,3,4,5,6,7,8,9,10,1
27 1,2,3,4,5,6,7,8,9,10,1backtrack
28 1,2,3,4,5,6,7,8,9,10,ok
29 1,2,3,4,5,6,7,8,9,10,backtrack
30 1,2,3,4,5,6,7,8,9,1
31 1,2,3,4,5,6,7,8,9,1backtrack
32 1,2,3,4,5,6,7,8,9,ok
33 1,2,3,4,5,6,7,8,9,backtrack
34 1,2,3,4,5,6,7,8,ok
35 1,2,3,4,5,6,7,8,backtrack
36 1,2,3,4,5,6,7,ok
37 1,2,3,4,5,6,7,backtrack
38 1,2,3,4,5,6,ok
39 1,2,3,4,5,6,backtrack
```

```
The regular expression does not match the
test subject
```

The complexity of the improved regex is linear to the length of the first 11 fields. 24 steps are needed in our example to match the 11 fields. It takes only 1 step to discover that P can't be matched. Another 27 steps are then needed to to backtrack all the iterations of the two quantifiers. 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.



Perl 5.26 · Helpful · Case sensitive · Exact spacing · Dot doesn't match line breaks · ^$ match at line breaks · Numbered capture · Reset

```
^(?>([^,\r\n]*+,){11})P
```

History:
12th field starts with P (lazy dot)
12th field starts with P (negated class)
12th field starts with P (atomic)

```
·········10········20········30········40···
 1  ok
 2  1
 3  1,
 4  1,2
 5  1,2,
 6  1,2,3
 7  1,2,3,
 8  1,2,3,4
 9  1,2,3,4,
10  1,2,3,4,5
11  1,2,3,4,5,
12  1,2,3,4,5,6
13  1,2,3,4,5,6,
14  1,2,3,4,5,6,7
15  1,2,3,4,5,6,7,
16  1,2,3,4,5,6,7,8
17  1,2,3,4,5,6,7,8,
18  1,2,3,4,5,6,7,8,9
19  1,2,3,4,5,6,7,8,9,
20  1,2,3,4,5,6,7,8,9,10
21  1,2,3,4,5,6,7,8,9,10,
22  1,2,3,4,5,6,7,8,9,10,11
23  1,2,3,4,5,6,7,8,9,10,11,
24  1,2,3,4,5,6,7,8,9,10,11,
25  1,2,3,4,5,6,7,8,9,10,11,
26  1,2,3,4,5,6,7,8,9,10,11,backtrack
27  backtrack
    Match attempt failed after 27 steps
```

Whole file · LF only

```
1,2,3,4,5,6,7,8,9,10,11,12,13
```

```
The regular expression does not match the
test subject
```

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 parenthesis 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` fails to match, so the engine backtracks. But the atomic group made it forget all backtracking positions. The match attempt at the start of the string fails immediately.

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`.

If you test the final solution in RegexBuddy's debugger, you'll see that it needs the same 24 steps to match the 11 fields. Then it takes 1 step to exit the atomic group and throw away all the backtracking information. Discovering that P can't be matched still takes one step. But because of the atomic group, backtracking it all takes only a single step.

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, it's very straightforward to write the regular expression `<html>.*?<head>.*?<title>.*?</title>.*?</head>.*?<body[^>]*>.*?</body>.*?</html>`. 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.

You've no doubt 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.

14. Preventing Regular Expression Denial of Service (ReDoS)

The previous topic explains catastrophic backtracking with practical examples from the perspective of somebody trying to get their regular expressions to work and perform well on their own PC. You should understand those examples before reading this topic.

It's annoying when catastrophic backtracking happens on your PC. But when it happens in a server application with multiple concurrent users, it can really be catastrophic. Too many users running regexes that exhibit catastrophic backtracking will bring down the whole server. And "too many" need only be as few as the number of CPU cores in the server.

If the server accepts regexes from the user, then the user can easily provide one that exhibits catastrophic backtracking on any subject. If the server accepts subject data from the user, then the user may be able to provide subjects that trigger catastrophic backtracking in regexes used by the server, if those regexes are predisposed to catastrophic backtracking. When the user can do either of those things, the server is susceptible to regular expression denial of service (ReDoS). When enough users (or one actor masquerading as many users) provide malicious regexes and/or subjects to match against, the server will be spending nearly all its CPU cycles on trying to match those regexes.

Handling Regexes Provided by The User

If your application allows the user to provide their own regexes, then your only real defense is to use a text-directed regex engine. Those engines don't backtrack. But they also don't support features like backreferences that depend on backtracking and that many users expect.

If your application uses a backtracking engine with user-provided regexes, then you can only mitigate the consequences of catastrophic backtracking. And you'll really need to do so. It's very easy for people with limited regex skills to accidentally craft one that degenerates into catastrophic backtracking.

You'll need to use a regex engine that aborts the match attempt when catastrophic backtracking occurs rather than running until the OS kills the script or crashing. You can easily test this. When the regex `(x\w{1,10})+y` is attempted on an ever growing string of `x`'s there should be a reasonable limit on how long it takes for the regex engine to give up. Ideally your engine will allow you to configure this limit for your purposes. The .NET engine, for example, allows you to pass a timeout to the Regex() constructor. The PCRE engine allows you to set recursion limits. The lower your limits the better the protection against ReDoS, but the more likely you'll be aborting regexes that would find a valid match given slightly more time.

Reviewing Regexes in The Application

If the server only uses regexes that are hard-coded in your application, then you can prevent regex-based denial of service attacks entirely. You need to make sure that your regexes won't exhibit catastrophic backtracking regardless of the subjects they're used on. This isn't particularly difficult for somebody with a solid grasp of regular expressions. But it does require care and attention. It's not enough to just test that the regex matches valid subjects. You need to make sure, by looking at the regex independently of any subject data, that it is not possible for multiple permutations of the same regex to match the same thing.

Permutations occur when you give the regular expression a choice. You can do this with alternation and with quantifiers. So these are the regex tokens you need to inspect. Possessive quantifiers are excepted, because they never backtrack.

Alternation

Alternatives must be mutually exclusive. If multiple alternatives can match the same text then the engine will try both if the remainder of the regex fails. If the alternatives are in a group that is repeated, you have catastrophic backtracking.

A classic example is `(.|\s)*` to match any amount of any text when the regex flavor does not have a "dot matches line breaks" mode. If this is part of a longer regex then a subject string with a sufficiently long run of spaces will break the regex. The engine will try every possible combination of the spaces being matched by `.` or `\s`. For example, 3 spaces could be matched as `...`, `..\s`, `.\s.`, `.\s\s`, `\s..`, `\s.\s`, `\s\s.`, or `\s\s\s`. That's 2^N permutations. The fix is to use `(.|\n)*` to make the alternatives mutually exclusive. Even better to be more specific about which characters are really allowed, such as `[\r\n\t\x20-\x7E]` for ASCII printables, tabs, and line breaks.

It is acceptable for two alternatives to partially match the same text. `[0-9]*\.[0-9]+|[0-9]+` is perfectly fine to match a floating point number with optional integer part and optional fraction. Though a subject that consists of only digits is initially matched by `[0-9]*` and does cause some backtracking when `\.` fails, this backtracking never becomes catastrophic. Even if you put this inside a group in a longer regex, the group only does a minimal amount of backtracking. (But the group mustn't have a quantifier or it will fall foul of the rule for nested quantifiers.)

Quantifiers in Sequence

Quantified tokens that are in sequence must either be mutually exclusive with each other or be mutually exclusive with what comes between them. Otherwise both can match the same text and all combinations of the two quantifiers will be tried when the remainder of the regex fails to match. A token inside a group with alternation is still in sequence with any token before or after the group.

A classic example is `a.*?b.*?c` to match 3 things with "anything" between them. When `c` can't be matched the first `.*?` expands character by character until the end of the line or file. For each expansion the second `.*?` expands character by character to match the remainder of the line or file. The fix is to realize that you can't have "anything" between them. The first run needs to stop at `b` and the second run needs to stop at `c`. With single characters `a[^b]*+b[^c]*+c` is an easy solution. Since we now stop at the delimiter, we can use possessive quantifiers to further increase performance.

For a more complex example and solution, see matching a complete HTML file in the previous topic. This explains how you can use atomic grouping to prevent backtracking in more complex situations.

Nested Quantifiers

A group that contains a token with a quantifier must not have a quantifier of its own unless the quantified token inside the group can only be matched with something else that is mutually exclusive with it. That ensures that there is no way that fewer iterations of the outer quantifier with more iterations of the inner quantifier can match the same text as more iterations of the outer quantifier with fewer iterations of the inner quantifier.

The regex `(x\w{1,10})+y` matches a sequence of one or more codes that start with an `x` followed by 1 to 10 word characters, all followed by a `y`. All is well as long as the `y` can be matched. When the `y` is missing, backtracking occurs. If the string doesn't have too many `x`'s then backtracking happens very quickly. Things only turn catastrophic when the subject contains a long sequence of `x`'s. `x` and `x` are not mutually exclusive. So the repeated group can match `xxxx` in one iteration as `x\w\w\w` or in two iterations as as `x\wx\w`.

To solve this, you first need to consider whether `x` and `y` should be allowed in the 1 to 10 characters that follow it. Excluding the `x` eliminates most backtracking. What's left won't be catastrophic. You could exclude it with character class subtraction as in `(x[\w-[x]]{1,10})+y` or with character class intersection as in `(x[\w&&[^x]]{1,10})+y`. If you don't have those features you'll need to spell out the characters you want to allow: `(x[a-wyz0-9_]{1,10})+y`.

If the `x` should be allowed then your only solution is to disallow the `y` in the same way. Then you can make the group atomic or the quantifier possessive to eliminate the backtracking.

If both `x` and `y` should be allowed in the sequences of 1 to 10 characters, then there is no regex-only solution. You can't make the group atomic or the quantifier possessive as then `\w{1,10}` matches the final `y` which causes `y` to fail.

Other Defensive Techniques

In addition to preventing catastrophic backtracking as explained above, you should make your regular expressions as strict as possible. The stricter the regex, the less backtracking it does and thus the better it performs. Even if you can't measure the performance difference because the regex is used infrequently on short strings, proper technique is a habit. It also reduces the chance that a less experienced developer introduces catastrophic backtracking when they extend your regex later.

Make groups that contain alternatives atomic as much as you can. Use `\b(?>one|two|three)\b` to match a list of words.

Make quantifiers possessive as much as you can. If a repeated token is mutually exclusive with what follows, enforce that with a possessive quantifier.

Use (negated) character classes instead of the dot. It's rare that you really want to allow "anything". A double-quoted string, for example, can't contain "anything". It can't contain unescaped double quotes. So use `"[^"\n]*+"` instead of `".*?"`. Though both find exactly the same matches when used on their own, the latter can lead to catastrophic backtracking when pasted into a longer regex. The former never backtracks regardless of anything else the regex needs to match.

Why Use Regexes at All?

Some would certainly argue that the above only shows that regexes are dangerous and that they should not be used. They'll then force developers to do the job with procedural code. Procedural code to match non-trivial patterns quickly becomes long and complicated, increasing the chance of bugs and the cost to develop and maintain the code. Many pattern matching problems are naturally solved with recursion. And when a large subject string can’t be matched, runaway recursion leads to stack overflows that crash the application.

Developers need to learn to correctly use their tools. This is no different for regular expressions than for anything else.

15. 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 spaces 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 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)+)!.

16. 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.

In PowerGREP, simply use the Unicode escape `\u20AC`. U+20AC is the Unicode code point for the euro symbol. It will always match the euro symbol, whether your file is encoded in UTF-8, UTF-16, UCS-2 or whatever. Even when your file is encoded with a legacy 8-bit code page, there's no confusion. `\u20AC` is always the euro symbol.

PowerGREP also supports 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.

Just Great Software applications 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-16LE 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 PowerGREP.

Part 6

Regular Expression Reference

1. Special and Non-Printable Characters

Feature: Literal character
Syntax: Any character except `[\^$.|?*+()`
Description: All characters except the listed special characters match a single instance of themselves
Example: `a` matches `a`

Feature: Literal curly braces
Syntax: `{` and `}`
Description: `{` and `}` are literal characters, unless they're part of a valid regular expression token such as a quantifier `{3}`
Example: `{` matches `{`

Feature: Backslash escapes a metacharacter
Syntax: `\` followed by any of `[\^$.|?*+(){}`
Description: A backslash escapes special characters to suppress their special meaning
Example: `\*` matches `*`

Feature: Escape sequence
Syntax: `\Q...\E`
Description: Matches the characters between `\Q` and `\E` literally, suppressing the meaning of special characters
Example: `\Q+-*/\E` matches `+-*/`

Feature: Hexadecimal escape
Syntax: `\xFF` where FF are 2 hexadecimal digits
Description: Matches the character at the specified position in the code page
Example: `\xA9` matches `©` when using the Latin-1 code page

Feature: Character escape
Syntax: `\n`, `\r` and `\t`
Description: Match an LF character, CR character and a tab character respectively
Example: `\r\n` matches a Windows CRLF line break

Feature: Line break
Syntax: `\R`
Description: Matches any line break, including CRLF as a pair, CR only, LF only, form feed, vertical tab, and any Unicode line break
Example:

Feature: Line break
Syntax: `\R`
Description: Matches the next line control character U+0085
Example:

Feature: Line break
Syntax: Literal CRLF, LF, or CR line break
Description: Matches CRLF as a pair, CR only, and LF only regardless of the line break style used in the regex
Example:

Feature: Character escape
Syntax: `\a`
Description: Match the “alert” or “bell” control character (ASCII 0x07)
Example:

Feature: Character escape
Syntax: `\e`
Description: Match the “escape” control character (ASCII 0x1A)
Example:

Feature: Character escape
Syntax: `\f`
Description: Match the “form feed” control character (ASCII 0x0C)
Example:

Feature: Octal escape
Syntax: `\o{7777}` where 7777 is any octal number
Description: Matches the character at the specified position in the active code page
Example: `\o{20254}` matches `€` when using Unicode

2. Basic Features

Feature: Dot
Syntax: `.` (dot)
Description: Matches any single character except line break characters. Most regex flavors have an option to make the dot match line break characters too.
Example: `.` matches `x` or (almost) any other character

Feature: Not a line break
Syntax: `\N`
Description: Matches any single character except line break characters, like the dot, but is not affected by any options that make the dot match all characters including line breaks.
Example: `\N` matches `x` or any other character that is not a line break

Feature: Alternation
Syntax: `|` (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 alternatives.
Example: `abc|def|xyz` matches `abc`, `def` or `xyz`

Feature: Alternation is eager
Syntax: `|`
Description: Alternation returns the first alternative that matches.
Example: `a|ab` matches `a` in `ab`

3. Character Classes

Feature: Character class
Syntax: `[`
Description: When used outside a character class, `[` begins a character class. Inside a character class, different rules apply. Unless otherwise noted, the syntax on this page is only valid inside character classes, while the syntax on all other reference pages is not valid inside character classes.
Example:

Feature: Literal character
Syntax: Any character except `^-]\`
Description: All characters except the listed special characters are literal characters that add themselves to the character class.
Example: `[abc]` matches `a`, `b` or `c`

Feature: Backslash escapes a metacharacter
Syntax: `\` (backslash) followed by any of `^-]\`
Description: A backslash escapes special characters to suppress their special meaning.
Example: `[\^\]]` matches `^` or `]`

Feature: Range
Syntax: `-` (hyphen) between two tokens that each specify a single character.
Description: Adds a range of characters to the character class.
Example: `[a-zA-Z0-9]` matches any ASCII letter or digit

Feature: Negated character class
Syntax: `^` (caret) immediately after the opening `[`
Description: Negates the character class, causing it to match a single character *not* listed in the character class.
Example: `[^a-d]` matches `x` (any character except a, b, c or d)

Feature: Literal opening bracket
Syntax: `[`
Description: An opening square bracket is a literal character that adds an opening square bracket to the character class.
Example: `[ab[cd]ef]` matches `aef]`, `bef]`, `[ef]`, `cef]`, and `def]`

Feature: Character class subtraction
Syntax: `[base-[subtract]]`
Description: Removes all characters in the "subtract" class from the "base" class.
Example: `[a-z-[aeiuo]]` matches a single letter that is not a vowel.

Feature: Character class intersection
Syntax: `[base&&[intersect]]`
Description: Reduces the character class to the characters present in both "base" and "intersect".
Example: `[a-z&&[^aeiuo]]` matches a single letter that is not a vowel.

Feature: Character escape
Syntax: `\n`, `\r` and `\t`
Description: Add an LF character, a CR character, or a tab character to the character class, respectively.
Example: `[\n\r\t]` a line feed, a carriage return, or a tab.

Feature: Character escape
Syntax: `\a`
Description: Add the “alert” or “bell” control character (ASCII 0x07) to the character class.
Example: `[\a\t]` matches a bell or a tab character.

Feature: Character escape
Syntax: `\b`
Description: Add the “backspace” control character (ASCII 0x08) to the character class.
Example: `[\b\t]` matches a backspace or a tab character.

Feature: Character escape
Syntax: `\e`
Description: Add the “escape” control character (ASCII 0x1A) to the character class.
Example: `[\e\t]` matches an escape or a tab character.

Feature: Character escape
Syntax: `\f`
Description: Add the “form feed” control character (ASCII 0x0C) to the character class.
Example: `[\f\t]` matches a form feed or a tab character.

Feature: POSIX class
Syntax: `[:alpha:]`
Description: Matches one character from a POSIX character class. Can only be used in a bracket expression.
Example: `[[:digit:][:lower:]]` matches one of `0` through `9` or `a` through `z`

Feature: POSIX shorthand class
Syntax: `[:d:]`, `[:s:]`, `[:w:]`
Description: Matches one character from the POSIX character classes “digit”, “space”, or “word”. Can only be used in a bracket expression.
Example: `[[:s:][:d:]]` matches a space, a tab, a line break, or one of `0` through `9`

Feature: POSIX shorthand class
Syntax: `[:l:]` and `[:u:]`
Description: Matches one character from the POSIX character classes “lower” or “upper”. Can only be used in a bracket expression.
Example: `[[:u:]][[:l:]]` matches `Aa` but not `aA`.

Feature: POSIX shorthand class
Syntax: `[:h:]`
Description: Matches one character from the POSIX character classes “blank”. Can only be used in a bracket expression.
Example: `[[:h:]]` matches a space.

Feature: POSIX shorthand class
Syntax: `[:V:]`
Description: Matches a vertical whitespace character. Can only be used in a bracket expression.
Example: `[[:v:]]` match any single vertical whitespace character.

Feature: POSIX class
Syntax: Any supported `\p{…}` syntax
Description: `\p{…}` syntax can be used inside character classes.
Example: `[\p{Digit}\p{Lower}]` matches one of `0` through `9` or `a` through `z`

Feature: POSIX class
Syntax: `\p{Alpha}`
Description: Matches one character from a POSIX character class.
Example: `\p{Digit}` matches any single digit.

Feature: POSIX class
Syntax: `\p{IsAlpha}`
Description: Matches one character from a POSIX character class.
Example: `\p{IsDigit}` matches any single digit.

4. Anchors

Feature: String anchor
Syntax: `^` (caret)
Description: Matches at the start of the string the regex pattern is applied to.
Example: `^.` matches `a` in `abc\ndef`

Feature: String anchor
Syntax: `$` (dollar)
Description: Matches at the end of the string the regex pattern is applied to.
Example: `.$` matches `f` in `abc\ndef`

Feature: String anchor
Syntax: `$` (dollar)
Description: Matches before the final line break in the string (if any) in addition to matching at the very end of the string.
Example: `.$` matches `f` in `abc\ndef\n`

Feature: Line anchor
Syntax: `^` (caret)
Description: Matches after each line break in addition to matching at the start of the string, thus matching at the start of each line in the string.
Example: `^.` matches `a` and `d` in `abc\ndef`

Feature: Line anchor
Syntax: `$` (dollar)
Description: Matches before each line break in addition to matching at the end of the string, thus matching at the end of each line in the string.
Example: `.$` matches `c` and `f` in `abc\ndef`

Feature: String anchor
Syntax: `\A`
Description: Matches at the start of the string the regex pattern is applied to.
Example: `\A\w` matches only `a` in `abc`

Feature: Attempt anchor
Syntax: `\G`
Description: Matches at the start of the match attempt.
Example: `\G\w` matches `a`, `b`, and `c` when iterating over all matches in `abc def`

Feature: String anchor
Syntax: `\z`
Description: Matches at the end of the string the regex pattern is applied to.
Example: `\w\z` matches `f` in `abc\ndef` but fails to match `abc\ndef\n`

Feature: String anchor
Syntax: `\Z`
Description: Matches at the end of the string as well as before the final line break in the string (if any).
Example: `.\Z` matches `f` in `abc\ndef` and in `abc\ndef\n` but fails to match `abc\ndef\n\n`

5. Word Boundaries

Feature: Word boundary
Syntax: `\b`
Description: Matches at a position that is followed by a word character but not preceded by a word character, or that is preceded by a word character but not followed by a word character.
Example: `\b.` matches `a`, ` `, and `d` in `abc def`

Feature: Word boundary
Syntax: `\B`
Description: Matches at a position that is preceded and followed by a word character, or that is not preceded and not followed by a word character.
Example: `\B.` matches `b`, `c`, `e`, and `f` in `abc def`

Feature: Tcl word boundary
Syntax: `\y`
Description: Matches at a position that is followed by a word character but not preceded by a word character, or that is preceded by a word character but not followed by a word character.
Example: `\y.` matches `a`, ` `, and `d` in `abc def`

Feature: Tcl word boundary
Syntax: `\Y`
Description: Matches at a position that is preceded and followed by a word character, or that is not preceded and not followed by a word character.
Example: `\Y.` matches `b`, `c`, `e`, and `f` in `abc def`

Feature: Tcl word boundary
Syntax: `\m`
Description: Matches at a position that is followed by a word character but not preceded by a word character.
Example: `\m.` matches `a` and `d` in `abc def`

Feature: Tcl word boundary
Syntax: `\M`
Description: Matches at a position that is preceded by a word character but not followed by a word character.
Example: `.\M` matches `c` and `f` in `abc def`

6. Quantifiers

Feature: Greedy quantifier
Syntax: `?` (question mark)
Description: Makes the preceding item optional. Greedy, so the optional item is included in the match if possible.
Example: `abc?` matches `abc` or `ab`

Feature: Lazy quantifier
Syntax: `??`
Description: Makes the preceding item optional. Lazy, so the optional item is excluded in the match if possible.
Example: `abc??` matches `ab` or `abc`

Feature: Possessive quantifier
Syntax: `?+`
Description: Makes the preceding item optional. Possessive, so if the optional item can be matched, then the quantifier won't give up its match even if the remainder of the regex fails.
Example: `abc?+c` matches `abcc` but not `abc`

Feature: Greedy quantifier
Syntax: `*` (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`

Feature: Lazy quantifier
Syntax: `*?`
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"` and `"ghi"` in `abc "def" "ghi" jkl`

Feature: Possessive quantifier
Syntax: `*+`
Description: Repeats the previous item zero or more times. Possessive, so as many items as possible will be matched, without trying any permutations with less matches even if the remainder of the regex fails.
Example: `".*+"` can never match anything

Feature: Greedy quantifier
Syntax: `+` (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`

Feature: Lazy quantifier
Syntax: `+?`
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"` and `"ghi"` in `abc "def" "ghi" jkl`

Feature: Possessive quantifier
Syntax: `++`
Description: Repeats the previous item once or more. Possessive, so as many items as possible will be matched, without trying any permutations with less matches even if the remainder of the regex fails.
Example: `".++"` can never match anything

Feature: Fixed quantifier
Syntax: `{n}` where n is an integer >= 1
Description: Repeats the previous item exactly n times.
Example: `a{3}` matches `aaa`

Feature: Greedy quantifier
Syntax: `{n,m}` where n >= 0 and m >= 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: `a{2,4}` matches `aaaa`, `aaa` or `aa`

Feature: Greedy quantifier
Syntax: `{n,}` where n >= 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: `a{2,}` matches `aaaaa` in `aaaaa`

Feature: Greedy quantifier
Syntax: `{,m}` where m >= 1
Description: Repeats the previous item between zero and m times. Greedy, so repeating m times is tried before reducing the repetition to zero times.
Example: `a{,4}` matches `aaaa`, `aaa`, `aa`, `a`, or the empty string

Feature: Lazy quantifier
Syntax: `{n,m}?` where n >= 0 and m >= 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: `a{2,4}?` matches `aa`, `aaa` or `aaaa`

Feature: Lazy quantifier
Syntax: `{n,}?` where n >= 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: `a{2,}?` matches `aa` in `aaaaa`

Feature: Lazy quantifier
Syntax: `{,m}?` where m >= 1
Description: Repeats the previous item between zero and m times. Lazy, so repeating zero times is tried before increasing the repetition to m times.
Example: `a{,4}?` matches the empty string, `a`, `aa`, `aaa` or `aaaa`

Feature: Possessive quantifier
Syntax: `{n,m}+` where n >= 0 and m >= n
Description: Repeats the previous item between n and m times. Possessive, so as many items as possible up to m will be matched, without trying any permutations with less matches even if the remainder of the regex fails.
Example: `a{2,4}+a` matches `aaaaa` but not `aaaa`

Feature: Possessive quantifier
Syntax: `{n,}+` where n >= 0
Description: Repeats the previous item n or more times. Possessive, so as many items as possible will be matched, without trying any permutations with less matches even if the remainder of the regex fails.
Example: `a{2,}+a` never matches anything

7. Unicode Syntax Reference

This reference page explains what the Unicode tokens do when used outside character classes. All of these except `\X` can also be used inside character classes. Inside a character class, these tokens add the characters that they normally match to the character class.

Feature: Grapheme
Syntax: `\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.

Feature: Code point
Syntax: `\uFFFF` where FFFF are 4 hexadecimal digits
Description: Matches a specific Unicode code point.
Example: `\u00E0` matches `à` encoded as U+00E0 only. `\u00A9` matches `©`

Feature: Code point
Syntax: `\u{FFFF}` where FFFF are 1 to 4 hexadecimal digits
Description: Matches a specific Unicode code point.
Example: `\u{E0}` matches `à` encoded as U+00E0 only. `\u{A9}` matches `©`

Feature: Code point
Syntax: `\x{FFFF}` where FFFF are 1 to 4 hexadecimal digits
Description: Matches a specific Unicode code point.
Example: `\x{E0}` matches `à` encoded as U+00E0 only. `\x{A9}` matches `©`

Feature: Unicode category
Syntax: `\pL` where L is a Unicode category
Description: Matches a single Unicode code point in the specified Unicode category.
Example: `\pL` matches `à` encoded as U+00E0; `\pS` matches `©`

Feature: Unicode category
Syntax: `\PL` where L is a Unicode category
Description: Matches a single Unicode code point that is *not* in the specified Unicode category.
Example: `\PS` matches `à` encoded as U+00E0; `\PL` matches `©`

Feature: Unicode category
Syntax: `\p{L}` where L is a Unicode category
Description: Matches a single Unicode code point in the specified Unicode category.
Example: `\p{L}` matches `à` encoded as U+00E0; `\p{S}` matches `©`

Feature: Unicode category
Syntax: `\p{IsL}` where L is a Unicode category
Description: Matches a single Unicode code point in the specified Unicode category.
Example: `\p{IsL}` matches `à` encoded as U+00E0; `\p{IsS}` matches `©`

Feature: Unicode category
Syntax: `\p{Category}`
Description: Matches a single Unicode code point in the specified Unicode category.
Example: `\p{Letter}` matches à encoded as U+00E0; `\p{Symbol}` matches ©

Feature: Unicode category
Syntax: `\p{IsCategory}`
Description: Matches a single Unicode code point in the specified Unicode category.
Example: `\p{IsLetter}` matches à encoded as U+00E0; `\p{IsSymbol}` matches ©

Feature: Unicode script
Syntax: `\p{Script}`
Description: Matches a single Unicode code point that is part of the specified Unicode script. Each Unicode code point is part of exactly one script. Scripts never contain unassigned code points.
Example: `\p{Greek}` matches Ω

Feature: Unicode script
Syntax: `\p{IsScript}`
Description: Matches a single Unicode code point that is part of the specified Unicode script. Each Unicode code point is part of exactly one script. Scripts never contain unassigned code points.
Example: `\p{IsGreek}` matches Ω

Feature: Unicode block
Syntax: `\p{Block}`
Description: Matches a single Unicode code point that is part of the specified Unicode block. Each Unicode code point is part of exactly one block. Blocks may contain unassigned code points.
Example: `\p{Arrows}` matches any of the code points from U+2190 until U+21FF (← until ⇿)

Feature: Unicode block
Syntax: `\p{InBlock}`
Description: Matches a single Unicode code point that is part of the specified Unicode block. Each Unicode code point is part of exactly one block. Blocks may contain unassigned code points.
Example: `\p{InArrows}` matches any of the code points from U+2190 until U+21FF (← until ⇿)

Feature: Unicode block
Syntax: `\p{IsBlock}`
Description: Matches a single Unicode code point that is part of the specified Unicode block. Each Unicode code point is part of exactly one block. Blocks may contain unassigned code points.
Example: `\p{IsArrows}` matches any of the code points from U+2190 until U+21FF (← until ⇿)

Feature: Negated Unicode property
Syntax: `\P{Property}`
Description: Matches a single Unicode code point that does *not* have the specified property (category, script, or block).
Example: `\P{L}` matches ©

Feature: Negated Unicode property
Syntax: `\p{^Property}`
Description: Matches a single Unicode code point that does *not* have the specified property (category, script, or block).
Example: `\p{^L}` matches `©`

Feature: Unicode property
Syntax: `\P{^Property}`
Description: Matches a single Unicode code point that *does have* the specified property (category, script, or block). Double negative is taken as positive.
Example: `\P{^L}` matches `q`

8. Capturing Groups and Backreferences

Feature: Capturing group
Syntax: `(regex)`
Description: Parentheses group the regex between them. They capture the text matched by the regex inside them into a numbered group that can be reused with a numbered backreference. They allow you to apply regex operators to the entire grouped regex.
Example: `(abc){3}` matches `abcabcabc`. First group matches `abc`.

Feature: Non-capturing group
Syntax: `(?:regex)`
Description: Non-capturing parentheses group the regex so you can apply regex operators, but do not capture anything.
Example: `(?:abc){3}` matches `abcabcabc`. No groups.

Feature: Backreference
Syntax: `\1` through `\9`
Description: Substituted with the text matched between the 1st through 9th numbered capturing group.
Example: `(abc|def)=\1` matches `abc=abc` or `def=def`, but not `abc=def` or `def=abc`.

Feature: Backreference
Syntax: `\10` through `\99`
Description: Substituted with the text matched between the 10th through 99th numbered capturing group.
Example:

Feature: Backreference
Syntax: `\k<1>` through `\k<99>`
Description: Substituted with the text matched between the 1st through 99th numbered capturing group.
Example: `(abc|def)=\k<1>` matches `abc=abc` or `def=def`, but not `abc=def` or `def=abc`.

Feature: Backreference
Syntax: `\k'1'` through `\k'99'`
Description: Substituted with the text matched between the 1st through 99th numbered capturing group.
Example: `(abc|def)=\k'1'` matches `abc=abc` or `def=def`, but not `abc=def` or `def=abc`.

Feature: Backreference
Syntax: `(?P=1)` through `(?P=99)`
Description: Substituted with the text matched between the 1st through 99th numbered capturing group.
Example: `(abc|def)=(?P=1)` matches `abc=abc` or `def=def`, but not `abc=def` or `def=abc`.

Feature: Relative Backreference
Syntax: `\k<-1>`, `\k<-2>`, etc.
Description: Substituted with the text matched by the capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from right to left starting at the backreference.
Example: `(a)(b)(c)(d)\k<-3>` matches `abcdb`.

Feature: Relative Backreference
Syntax: `\k'-1'`, `\k'-2'`, etc.
Description: Substituted with the text matched by the capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from right to left starting at the backreference.
Example: `(a)(b)(c)(d)\k'-3'` matches `abcdb`.

Feature: Failed backreference
Syntax: Any numbered backreference
Description: Backreferences to groups that did not participate in the match attempt fail to match.
Example: `(a)?\1` matches `aa` but fails to match `b`.

Feature: Nested backreference
Syntax: Any numbered backreference
Description: Backreferences can be used inside the group they reference.
Example: `(a\1?){3}` matches `aaaaaa`.

Feature: Forward reference
Syntax: Any numbered backreference
Description: Backreferences can be used before the group they reference.
Example: `(\2?(a)){3}` matches `aaaaaa`.

9. Named Groups and Backreferences

Feature: Named capturing group
Syntax: `(?<name>regex)`
Description: Captures the text matched by "regex" into the group "name". The name can contain letters and numbers but must start with a letter.
Example: `(?<x>abc){3}` matches `abcabcabc`. The group x matches `abc`.

Feature: Named capturing group
Syntax: `(?'name'regex)`
Description: Captures the text matched by "regex" into the group "name". The name can contain letters and numbers but must start with a letter.
Example: `(?'x'abc){3}` matches `abcabcabc`. The group x matches `abc`.

Feature: Named capturing group
Syntax: `(?P<name>regex)`
Description: Captures the text matched by "regex" into the group "name". The name can contain letters and numbers but must start with a letter.
Example: `(?P<x>abc){3}` matches `abcabcabc`. The group x matches `abc`.

Feature: Duplicate named group
Syntax: Any named group
Description: Two named groups can share the same name.
Example: `(?<x>a)|(?<x>b)` matches `a` or `b`.

Feature: Duplicate named group
Syntax: Any named group
Description: Named groups that share the same name are treated as one an the same group, so there are no pitfalls when using backreferences to that name.
Example:

Feature: Named backreference
Syntax: `\k<name>`
Description: Substituted with the text matched by the named group "name".
Example: `(?<x>abc|def)=\k<x>` matches `abc=abc` or `def=def`, but not `abc=def` or `def=abc`.

Feature: Named backreference
Syntax: `\k'name'`
Description: Substituted with the text matched by the named group "name".
Example: `(?'x'abc|def)=\k'x'` matches `abc=abc` or `def=def`, but not `abc=def` or `def=abc`.

Feature: Named backreference
Syntax: `(?P=name)`
Description: Substituted with the text matched by the named group "name".
Example: `(?P<x>abc|def)=(?P=x)` matches `abc=abc` or `def=def`, but not `abc=def` or `def=abc`.

Feature: Failed backreference
Syntax: Any named backreference
Description: Backreferences to groups that did not participate in the match attempt fail to match.
Example: `(?<x>a)?\k<x>` matches `aa` but fails to match `b`.

Feature: Nested backreference
Syntax: Any named backreference
Description: Backreferences can be used inside the group they reference.
Example: `(?<x>a\k<x>?){3}` matches `aaaaaa`.

Feature: Forward reference
Syntax: Any named backreference
Description: Backreferences can be used before the group they reference.
Example: `(\k<x>?(?<x>a)){3}` matches `aaaaaa`.

Feature: Named capturing group
Syntax: Any named capturing group
Description: A number is a valid name for a capturing group.
Example: `(?<17>abc){3}` matches `abcabcabc`. The group named "17" matches `abc`.

Feature: Named backreference
Syntax: Any named backreference
Description: A number is a valid name for a backreference which then points to a group with that number as its name.
Example: `(?<17>abc|def)=\k<17>` matches `abc=abc` or `def=def`, but not `abc=def` or `def=abc`.

10. Special Groups

Feature: Comment
Syntax: `(?#comment)`
Description: Everything between `(?#` and `)` is ignored by the regex engine.
Example: `a(?#foobar)b` matches `ab`

Feature: Branch reset group
Syntax: `(?|regex)`
Description: If the regex inside the branch reset group has multiple alternatives with capturing groups, then the capturing group numbers are the same in all the alternatives.
Example: `(x)(?|(a)|(bc)|(def))\2` matches `xaa`, `xbcbc`, or `xdefdef` with the first group capturing `x` and the second group capturing `a`, `bc`, or `def`

Feature: Atomic group
Syntax: `(?>regex)`
Description: Atomic groups prevent the regex engine from backtracking back into the group after a match has been found for the group. If the remainder of the regex fails, the engine may backtrack over the group if a quantifier or alternation makes it optional. But it will not backtrack into the group to try other permutations of the group.
Example: `a(?>bc|b)c` matches `abcc` but not `abc`

Feature: Positive lookahead
Syntax: `(?=regex)`
Description: 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 `one(?=two)three`, both `two` and `three` have to match at the position where the match of `one` ends.
Example: `t(?=s)` matches the second `t` in `streets`.

Feature: Negative lookahead
Syntax: `(?!regex)`
Description: Similar to positive lookahead, except that negative lookahead only succeeds if the regex inside the lookahead fails to match.
Example: `t(?!s)` matches the first `t` in `streets`.

Feature: Positive lookbehind
Syntax: `(?<=regex)`
Description: Matches at a position if the pattern inside the lookbehind can be matched ending at that position.
Example: `(?<=s)t` matches the first `t` in `streets`.

Feature: Negative lookbehind
Syntax: `(?<!regex)`
Description: Matches at a position if the pattern inside the lookbehind cannot be matched ending at that position.
Example: `(?<!s)t` matches the second `t` in `streets`.

Feature: Lookbehind
Syntax: `(?<=regex|longer regex)`
Description: Alternatives inside lookbehind can differ in length.
Example: `(?<=is|e)t` matches the second and fourth `t` in `twisty streets`.

Feature: Lookbehind
Syntax: `(?<=x{n,m})`
Description: Quantifiers with a finite maximum number of repetitions can be used inside lookbehind.
Example: `(?<=s\w{1,7})t` matches only the fourth `t` in `twisty streets`.

Feature: Lookbehind
Syntax: `(?<=regex)`
Description: The full regular expression syntax can be used inside lookbehind.
Example: `(?<=s\w+)t` matches only the fourth `t` in `twisty streets`.

Feature: Lookbehind
Syntax: `(group)(?<=\1)`
Description: Backreferences can be used inside lookbehind. Syntax prohibited in lookbehind is also prohibited in the referenced capturing group.
Example: `(\w).+(?<=\1)` matches `twisty street` in `twisty streets`.

Feature: Keep text out of the regex match
Syntax: `\K`
Description: The text matched by the part of the regex to the left of the `\K` is omitted from the overall regex match. Other than that the regex is matched normally from left to right. Capturing groups to the left of the `\K` capture as usual.
Example: `s\Kt` matches only the first `t` in `streets`.

Feature: Lookaround conditional
Syntax: `(?(?=regex)then|else)` where `(?=regex)` is any valid lookaround and `then` and `else` are any valid regexes
Description: If the lookaround succeeds, the “then” part must match for the overall regex to match. If the lookaround fails, the “else” part must match for the overall regex to match. The lookaround is zero-length. The “then” and “else” parts consume their matches like normal regexes.
Example: `(?(?<=a)b|c)` matches the second `b` and the first `c` in `babxcac`

Feature: Named conditional
Syntax: `(?(name)then|else)` where `name` is the name of a capturing group and `then` and `else` are any valid regexes
Description: If the capturing group with the given name took part in the match attempt thus far, the “then” part must match for the overall regex to match. If the capturing group did not take part in the match thus far, the “else” part must match for the overall regex to match.
Example: `(?<one>a)?(?(one)b|c)` matches `ab`, the first `c`, and the second `c` in `babxcac`

Feature: Named conditional
Syntax: `(?(<name>)then|else)` where `name` is the name of a capturing group and `then` and `else` are any valid regexes
Description: If the capturing group with the given name took part in the match attempt thus far, the “then” part must match for the overall regex to match. If the capturing group did not take part in the match thus far, the “else” part must match for the overall regex to match.
Example: `(?<one>a)?(?(<one>)b|c)` matches `ab`, the first `c`, and the second `c` in `babxcac`

Feature: Named conditional
Syntax: `(?('name')then|else)` where `name` is the name of a capturing group and `then` and `else` are any valid regexes
Description: If the capturing group with the given name took part in the match attempt thus far, the “then” part must match for the overall regex to match. If the capturing group did not take part in the match thus far, the “else” part must match for the overall regex to match.
Example: `(?'one'a)?(?('one')b|c)` matches `ab`, the first `c`, and the second `c` in `babxcac`

Feature: Conditional
Syntax: `(?(1)then|else)` where `1` is the number of a capturing group and `then` and `else` are any valid regexes
Description: If the referenced capturing group took part in the match attempt thus far, the “then” part must match for the overall regex to match. If the capturing group did not take part in the match thus far, 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`

Feature: Relative conditional
Syntax: `(?(-1)then|else)` where `-1` is a negative integer and `then` and `else` are any valid regexes
Description: Conditional that tests the capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from right to left starting immediately before the conditional. If the referenced capturing group took part in the match attempt thus far, the “then” part must match for the overall regex to match. If the capturing group did not take part in the match thus far, 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`

Feature: Forward conditional
Syntax: `(?(+1)then|else)` where `+1` is a positive integer and `then` and `else` are any valid regexes
Description: Conditional that tests the capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from left to right starting at the “then” part of conditional. If the referenced capturing group took part in the match attempt thus far, the “then” part must match for the overall regex to match. If the capturing group did not take part in the match thus far, the “else” part must match for the overall regex to match.
Example: `((?(+1)b|c)(d)?){2}` matches `cc` and `cdb` in `bdbdccxcdcxcdb`

11. Balancing Groups, Recursion, and Subroutines

Feature: Balancing group
Syntax: `(?<capture-subtract>regex)` where “capture” and “subtract” are group names and “regex” is any regex
Description: The name “subtract” must be used as the name of a capturing group elsewhere in the regex. If this group has captured matches that haven’t been subtracted yet, then the balancing group subtracts one capture from “subtract”, attempts to match “regex”, and stores its match into the group “capture”. If “capture” is omitted, the same happens without storing the match. If “regex” is omitted, the balancing group succeeds without advancing through the string. If the group “subtract” has no matches to subtract, then the balancing group fails to match, regardless of whether “regex” is specified or not.
Example: `^(?<l>\w)+\w?`
`(\k<l>(?<-l>))+`
`(?(l)(?!))$` matches any palindrome word

Feature: Balancing group
Syntax: `(?'capture-subtract'regex)` where “capture” and “subtract” are group names and “regex” is any regex
Description: The name “subtract” must be used as the name of a capturing group elsewhere in the regex. If this group has captured matches that haven’t been subtracted yet, then the balancing group subtracts one capture from “subtract”, attempts to match “regex”, and stores its match into the group “capture”. If “capture” is omitted, the same happens without storing the match. If “regex” is omitted, the balancing group succeeds without advancing through the string. If the group “subtract” has no matches to subtract, then the balancing group fails to match, regardless of whether “regex” is specified or not.
Example: `^(?'l'\w)+\w?`
`(\k'l'(?'-l'))+`
`(?(l)(?!))$` matches any palindrome word

Feature: Recursion
Syntax: `(?R)`
Description: Recursion of the entire regular expression.
Example: `a(?R)?z` matches `az`, `aazz`, `aaazzz`, etc.

Feature: Recursion
Syntax: `(?0)`
Description: Recursion of the entire regular expression.
Example: `a(?0)?z` matches `az`, `aazz`, `aaazzz`, etc.

Feature: Recursion
Syntax: `\g<0>`
Description: Recursion of the entire regular expression.
Example: `a\g<0>?z` matches `az`, `aazz`, `aaazzz`, etc.

Feature: Recursion
Syntax: `\g'0'`
Description: Recursion of the entire regular expression.
Example: `a\g'0'?z` matches `az`, `aazz`, `aaazzz`, etc.

Feature: Subroutine call
Syntax: `(?1)` where 1 is the number of a capturing group
Description: Recursion of a capturing group or subroutine call to a capturing group.
Example: `a(b(?1)?y)z` matches `abyz`, `abbyyz`, `abbbyyyz`, etc.

Feature: Subroutine call
Syntax: `\g<1>` where 1 is the number of a capturing group
Description: Recursion of a capturing group or subroutine call to a capturing group.
Example: `a(b\g<1>?y)z` matches `abyz`, `abbyyz`, `abbbyyyz`, etc.

Feature: Subroutine call
Syntax: `\g'1'` where 1 is the number of a capturing group
Description: Recursion of a capturing group or subroutine call to a capturing group.
Example: `a(b\g'1'?y)z` matches `abyz`, `abbyyz`, `abbbyyyz`, etc.

Feature: Relative subroutine call
Syntax: `(?-1)` where -1 is is a negative integer
Description: Recursion of or subroutine call to a capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from right to left starting at the subroutine call.
Example: `a(b(?-1)?y)z` matches `abyz`, `abbyyz`, `abbbyyyz`, etc.

Feature: Relative subroutine call
Syntax: `\g<-1>` where -1 is a negative integer
Description: Recursion of or subroutine call to a capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from right to left starting at the subroutine call.
Example: `a(b\g<-1>?y)z` matches `abyz`, `abbyyz`, `abbbyyyz`, etc.

Feature: Relative subroutine call
Syntax: `\g'-1'` where -1 is is a negative integer
Description: Recursion of or subroutine call to a capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from right to left starting at the subroutine call.
Example: `a(b\g'-1'?y)z` matches `abyz`, `abbyyz`, `abbbyyyz`, etc.

Feature: Forward subroutine call
Syntax: `(?+1)` where +1 is is a positive integer
Description: Recursion of or subroutine call to a capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from left to right starting at the subroutine call.
Example: `(?+1)x([ab])` matches `axa`, `axb`, `bxa`, and `bxb`

Feature: Forward subroutine call
Syntax: `\g<+1>` where +1 is a positive integer
Description: Recursion of or subroutine call to a capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from left to right starting at the subroutine call.
Example: `\g<+1>x([ab])` matches `axa`, `axb`, `bxa`, and `bxb`

Feature: Forward subroutine call
Syntax: `\g'+1'` where +1 is is a positive integer
Description: Recursion of or subroutine call to a capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from left to right starting at the subroutine call.
Example: `\g'+1'x([ab])` matches `axa`, `axb`, `bxa`, and `bxb`

Feature: Named subroutine call
Syntax: `(?&name)` where "name" is the name of a capturing group
Description: Recursion of a capturing group or subroutine call to a capturing group.
Example: `a(?<x>b(?&x)?y)z` matches `abyz`, `abbyyz`, `abbbyyyz`, etc.

Feature: Named subroutine call
Syntax: `(?P>name)` where "name" is the name of a capturing group
Description: Recursion of a capturing group or subroutine call to a capturing group.
Example: `a(?P<x>b(?P>x)?y)z` matches `abyz`, `abbyyz`, `abbbyyyz`, etc.

Feature: Named subroutine call
Syntax: `\g<name>` where "name" is the name of a capturing group
Description: Recursion of a capturing group or subroutine call to a capturing group.
Example: `a(?<x>b\g<x>?y)z` matches `abyz`, `abbyyz`, `abbbyyyz`, etc.

Feature: Named subroutine call
Syntax: `\g'name'` where "name" is the name of a capturing group
Description: Recursion of a capturing group or subroutine call to a capturing group.
Example: `a(?'x'b\g'x'?y)z` matches `abyz`, `abbyyz`, `abbbyyyz`, etc.

Feature: Subroutine definitions
Syntax: `(?(DEFINE)regex)` where "regex" is any regex
Description: The DEFINE group does not take part in the matching process. Subroutine calls can be made to capturing groups inside the DEFINE group.
Example: `(?(DEFINE)([ab]))`
`x(?1)y(?1)z` matches `xayaz`, `xaybz`, `xbyaz`, and `xbybz`

Feature: Subroutine calls capture
Syntax: Subroutine call using Ruby-style `\g` syntax
Description: A subroutine call to a capturing group makes that capturing group store the text matched during the subroutine call.
Example: When `([ab])\g'1'` matches `ab` the first capturing group holds `b` after the match.

Feature: Recursion reverts capturing groups
Syntax: Recursion or subroutine call using syntax other than `\g`
Description: When the regex engine exits from recursion or a subroutine call, it reverts all capturing groups to the text they had matched prior to entering the recursion or subroutine call.
Example: When `(a)(([bc])\1)(?2)` matches `abaca` the third group stores `b` after the match

Feature: Recursion does not isolate or revert capturing groups
Syntax: Recursion or subroutine call using Ruby-style `\g` syntax
Description: Capturing groups are not given any special treatment by recursion and subroutine calls, except perhaps that subroutine calls capture. Backreferences always see the text most recently matched by each capturing group, regardless of whether they are inside the same level of recursion or not.
Example: When `(a)(([bc])\1)\g'2'` matches `abaca` the third group stores `c` after the match

Feature: Recursion is atomic
Syntax: Recursion or subroutine call using `(?P>…)`
Description: Recursion and subroutine calls are atomic. Once the regex engine exits from them, it will not backtrack into it to try different permutations of the recursion or subroutine call.
Example: `(a+)(?P>1)(?P>1)` can never match anything because the first `(?P>1)` matches all remaining a's and the regex engine won't backtrack into the first `(?P>1)` when the second one fails

12. Replacement String Characters

Feature: Backslash
Syntax: A backslash that does not form a token
Description: A backslash that is not part of a replacement string token is a literal backslash.
Example: Replacing with `\!` yields `\!`

Feature: Backslash
Syntax: Trailing backslash
Description: A backslash at the end of the replacement string is a literal backslash.
Example: Replacing with `\` yields `\`

Feature: Backslash
Syntax: `\\`
Description: A backslash escapes itself.
Example: Replacing with `\\` yields `\`

Feature: Dollar
Syntax: A dollar that does not form a token
Description: A dollar sign that does not form a replacement string token is a literal dollar sign.
Example: Replacing with `$!` yields `$!`

Feature: Dollar
Syntax: Trailing dollar
Description: A dollar sign at the end of the replacement string is a literal dollar sign.
Example: Replacing with `$` yields `$`

Feature: Dollar
Syntax: `$$`
Description: A dollar sign escapes itself.
Example: Replacing with `$$` yields `$`

Feature: Dollar
Syntax: `\$`
Description: A backslash escapes a dollar sign.
Example: Replacing with `\$` yields `$`

Feature: Hexadecimal escape
Syntax: `\xFF` where FF are 2 hexadecimal digits
Description: Inserts the character at the specified position in the code page
Example: `\xA9` inserts `©` when using the Latin-1 code page

Feature: Unicode escape
Syntax: `\uFFFF` where FFFF are 4 hexadecimal digits
Description: Inserts a specific Unicode code point.
Example: `\u00E0` inserts `à` encoded as U+00E0 only. `\u00A9` inserts `©`

Feature: Unicode escape
Syntax: `\u{FFFF}` where FFFF are 1 to 4 hexadecimal digits
Description: Inserts a specific Unicode code point.
Example: `\u{E0}` inserts à encoded as U+00E0 only. `\u{A9}` inserts ©

Feature: Unicode escape
Syntax: `\x{FFFF}` where FFFF are 1 to 4 hexadecimal digits
Description: Inserts a specific Unicode code point.
Example: `\x{E0}` inserts à encoded as U+00E0 only. `\x{A9}` inserts ©

Feature: Character escape
Syntax: `\n`, `\r` and `\t`
Description: Insert an LF character, CR character and a tab character respectively
Example: `\r\n` inserts a Windows CRLF line break

Feature: Octal escape
Syntax: `\o{7777}` where 7777 is any octal number
Description: Inserts the character at the specified position in the active code page
Example: `\o{20254}` inserts € when using Unicode

13. Matched Text and Backreferences in Replacement Strings

Feature: Whole match
Syntax: `\&`
Description: Insert the whole regex match.
Example: Replacing `\d+` with `[\&]` in `1a2b` yields `[1]a[2]b`

Feature: Whole match
Syntax: `$&`
Description: Insert the whole regex match.
Example: Replacing `\d+` with `[$&]` in `1a2b` yields `[1]a[2]b`

Feature: Whole match
Syntax: `\0`
Description: Insert the whole regex match.
Example: Replacing `\d+` with `[\0]` in `1a2b` yields `[1]a[2]b`

Feature: Whole match
Syntax: `$0`
Description: Insert the whole regex match.
Example: Replacing `\d+` with `[$0]` in `1a2b` yields `[1]a[2]b`

Feature: Whole match
Syntax: `\g<0>`
Description: Insert the whole regex match.
Example: Replacing `\d+` with `[\g<0>]` in `1a2b` yields `[1]a[2]b`

Feature: Backreference
Syntax: `\1` through `\9`
Description: Insert the text matched by one of the first 9 capturing groups.
Example: Replacing `(a)(b)(c)` with `\3\3\1` in `abc` yields `cca`

Feature: Backreference
Syntax: `\10` through `\99`
Description: Insert the text matched by capturing groups 10 through 99.
Example:

Feature: Backreference and literal
Syntax: `\10` through `\99`
Description: When there are fewer capturing groups than the 2-digit number, treat this as a single-digit backreference followed by a literal number instead of as an invalid backreference.
Example: Replacing `(a)(b)(c)` with `\39\38\17` in `abc` yields `c9c8a7`

Feature: Backreference
Syntax: `$1` through `$9`
Description: Insert the text matched by one of the first 9 capturing groups.
Example: Replacing `(a)(b)(c)` with `$3$3$1` in `abc` yields `cca`

Feature: Backreference
Syntax: `$10` through `$99`
Description: Insert the text matched by capturing groups 10 through 99.
Example:

Feature: Backreference and literal
Syntax: `$10` through `$99`
Description: When there are fewer capturing groups than the 2-digit number, treat this as a single-digit backreference followed by a literal number instead of as an invalid backreference.
Example: Replacing `(a)(b)(c)` with `$39$38$17` in `abc` yields `c9c8a7`

Feature: Backreference
Syntax: `${1}` through `${99}`
Description: Insert the text matched by capturing groups 1 through 99.
Example: Replacing `(a)(b)(c)` with `${3}${3}${1}` in `abc` yields `cca`

Feature: Backreference
Syntax: `\g<1>` through `\g<99>`
Description: Insert the text matched by capturing groups 1 through 99.
Example: Replacing `(a)(b)(c)` with `\g<3>\g<3>\g<1>` in `abc` yields `cca`

Feature: Named backreference
Syntax: `${name}`
Description: Insert the text matched by the named capturing group "name".
Example: Replacing `(?'one'a)(?'two'b)` with `${two}${one}` in `ab` yields `ba`

Feature: Named backreference
Syntax: `\g<name>`
Description: Insert the text matched by the named capturing group "name".
Example: Replacing `(?P<one>a)(?P<two>b)` with `\g<two>\g<one>` in `ab` yields `ba`

Feature: Backreference to non-participating group
Syntax: Any supported backreference syntax
Description: A backreference to a non-participating capturing group is replaced with the empty string.
Example:

Feature: Last backreference
Syntax: `\+`
Description: Insert the text matched by the highest-numbered capturing group that actually participated in the match.
Example: Replacing `(a)(z)?` with `[\+]` in `ab` yields `[a]b`

Feature: Last backreference
Syntax: `$+`
Description: Insert the text matched by the highest-numbered capturing group that actually participated in the match.
Example: Replacing `(a)(z)?` with `[$+]` in `ab` yields `[a]b`

14. Context and Case Conversion in Replacement Strings

Feature: Match Context
Syntax: `` \` ``
Description: Insert the part of the subject string to the left of the regex match
Example: Replacing `b` with `` \` `` in `abc` yields `aac`

Feature: Match Context
Syntax: `` $` ``
Description: Insert the part of the subject string to the left of the regex match
Example: Replacing `b` with `` $` `` in `abc` yields `aac`

Feature: Match Context
Syntax: `\'`
Description: Insert the part of the subject string to the right of the regex match
Example: Replacing `b` with `\'` in `abc` yields `acc`

Feature: Match Context
Syntax: `$'`
Description: Insert the part of the subject string to the right of the regex match
Example: Replacing `b` with `$'` in `abc` yields `acc`

Feature: Match Context
Syntax: `$_`
Description: Insert the whole subject string
Example: Replacing `b` with `$_` in `abc` yields `aabcc`

Feature: Case Conversion
Syntax: `\U0` and `\U1` through `\U99`
Description: Insert the whole regex match or the 1st through 99th backreference with all letters in the matched text converted to uppercase.
Example: Replacing `.+` with `\U0` in `HeLlO WoRlD` yields `HELLO WORLD`

Feature: Case Conversion
Syntax: `\L0` and `\L1` through `\L99`
Description: Insert the whole regex match or the 1st through 99th backreference with all letters in the matched text converted to lowercase.
Example: Replacing `.+` with `\L0` in `HeLlO WoRlD` yields `hello world`

Feature: Case Conversion
Syntax: `\F0` and `\F1` through `\F99`
Description: Insert the whole regex match or the 1st through 99th backreference with the first letter in the matched text converted to uppercase and the remaining letters converted to lowercase.
Example: Replacing `.+` with `\F0` in `HeLlO WoRlD` yields `Hello world`

Feature: Case Conversion
Syntax: `\I0` and `\I1` through `\I99`
Description: Insert the whole regex match or the 1st through 99th backreference with the first letter of each word in the matched text converted to uppercase and the remaining letters converted to lowercase.
Example: Replacing `.+` with `\I0` in `HeLlO WoRlD` yields `Hello World`

15. Conditionals in Replacement Strings

Feature: Conditional
Syntax: `(?1yes:no)` through `(?99yes:no)`
Description: Conditional referencing a numbered capturing group. Inserts the “yes” part if the group participated or the “no” part if it didn’t.
Example: Replacing all matches of `(y)?|n` in `yyn!` with `(?1yes:no)` yields `yesyesno!`

Feature: Conditional
Syntax: `(?10yes:no)` through `(?99yes:no)`
Description: When there are fewer capturing groups than the 2-digit number, treat this as a single-digit conditional with the “yes” part starting with a literal number instead of as an invalid conditional.
Example: Replacing all matches of `(y)?|n` in `yyn!` with `(?19yes:no)` yields `9yes9yesno!`

Feature: Conditional
Syntax: `(?{1}yes:no)` through `(?{99}yes:no)`
Description: Conditional referencing a numbered capturing group. Inserts the “yes” part if the group participated or the “no” part if it didn’t.
Example: Replacing all matches of `(y)?|n` in `yyn!` with `(?{1}yes:no)` yields `yesyesno!`

Feature: Conditional
Syntax: `${1:+yes:no}` through `${99:+yes:no}`
Description: Conditional referencing a numbered capturing group. Inserts the “yes” part if the group participated or the “no” part if it didn’t.
Example: Replacing all matches of `(y)?|n` in `yyn!` with `${1:+yes:no}` yields `yesyesno!`

Feature: Conditional
Syntax: `${1:-no}` through `${99:-no}`
Description: Conditional referencing a numbered capturing group. Inserts the text captured by the group if it participated or the contents of the conditional if it didn’t.
Example: Replacing all matches of `(y)?|n` in `yyn!` with `${1:-no}` yields `yyno!`

Feature: Conditional
Syntax: `(?{name}yes:no)`
Description: Conditional referencing a named capturing group. Inserts the “yes” part if the group participated or the “no” part if it didn’t.
Example: Replacing all matches of `(?'one'y)?|n` in `yyn!` with `(?{one}yes:no)` yields `yesyesno!`

Feature: Conditional
Syntax: `${name:+yes:no}`
Description: Conditional referencing a named capturing group. Inserts the “yes” part if the group participated or the “no” part if it didn’t.
Example: Replacing all matches of `(?'one'y)?|n` in `yyn!` with `${one:+yes:no}` yields `yesyesno!`

Feature: Conditional
Syntax: `${name:-no}`
Description: Conditional referencing a named capturing group. Inserts the text captured by the group if it participated or the contents of the conditional if it didn’t.
Example: Replacing all matches of `(?'one'y)?|n` in `yyn!` with `${one:-no}` yields `yyno!`

Index

  - question mark, 319
  - reluctant, 321
  - specific amount, 320
  - star, 320
  - ungreedy, 321
  - zero or more, 320
  - zero or once, 319