PowerGREP

Manual

Version 3.2 — 3 October 2005

Table of Contents

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).

The tasks you can perform with PowerGREP broadly fit into three categories:

- Find files and information. Search using one or more regular expressions and/or words and phrases. PowerGREP will display file names, search matches and context, as you like. You can restrict the search to particular folders, files of certain types and even parts of files. E.g. you could search through the headings in the HTML files in the folder containing your web site's source files.
- Edit and convert text and data files. In addition to searching for text patterns, you can use regular expressions to substitute one pattern with another. E.g. you can search for dates in U.S. mm/dd/yy format and substitute them with the same dates in international yyyy/mm/dd format.
- Extract data and collect statistics. Extract useful information by searching through raw data files and save search matches, or regular expression substitution patterns into one or more new files. Group identical matches and count them to gather statistics from various kinds of log files.

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 JGsoft - Just Great Software.

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

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.

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.

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.

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. 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.

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 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.

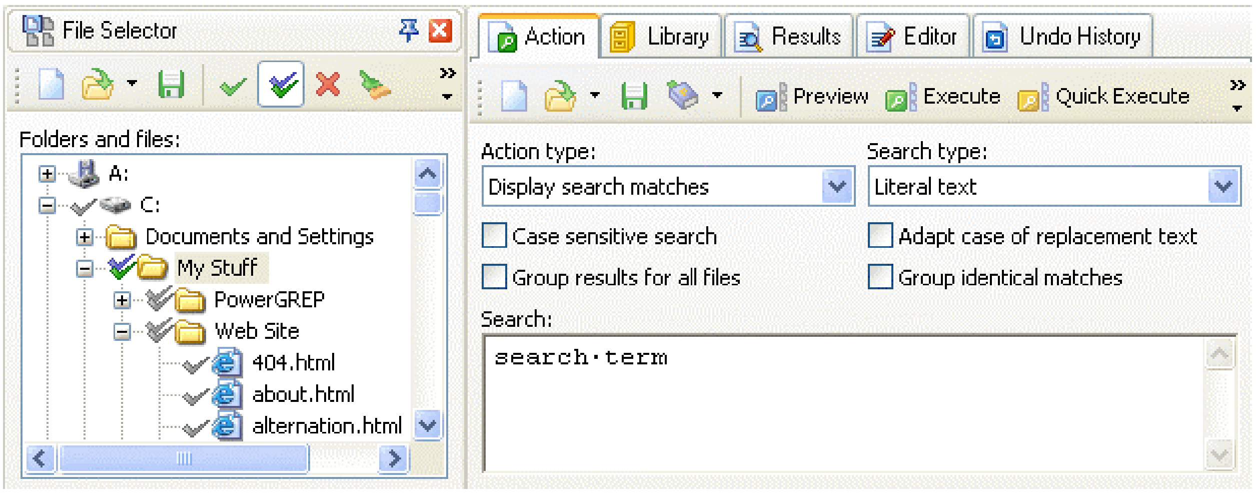
1. You start with telling PowerGREP which files you want to work with. Click on a file or folder in the File Selector. Then select Include File or Folder in the File Selector menu to mark the file or folder to be searched through. Marking a folder is a quick way to work with all the files in that folder. 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 pane. For a simple search, select the “display search matches” action type and the “literal text” search type at the top of the action pane. 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 pane. Double-click on a match to open the file in the editor and see its context.

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



Action type: Display search matches

Search type: Literal text

Case sensitive search

Adapt case of replacement text

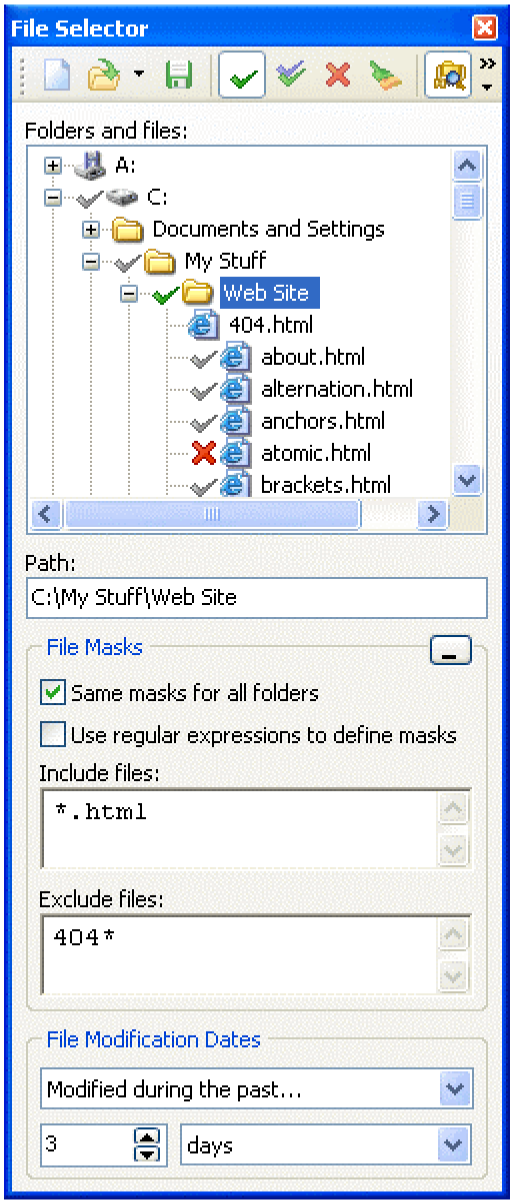
Group results for all files

Group identical matches

Search:

```
search·term
```

4. Mark Files for Searching



Path: C:\My Stuff\Web Site

[✓] Same masks for all folders

[ ] Use regular expressions to define masks

Include files: `*.html`

Exclude files: `404*`

Modified during the past...

3 days

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

1. 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.

2. 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.

3. 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.

4. 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.

5. 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.

6. 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.

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

8. Finally, 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.

The next step is to define the action you want to run on the files you just marked. After running the search, you can further narrow down the search results with the Mark Files with Search Results command or the Search Only through Files with Results option.

5. Define a Search Action

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

- Find files: Get a list of files matching the search terms. This is the fastest way to search. It is also the only way to get a list of files *not* matching your search terms, or to get a list of all marked files (by not entering any search terms).
- Display search matches: Get a list of all search matches in each file. Use this action type when you want to inspect the context of individual matches.
- Search-and-replace: Substitute search matches. Have PowerGREP replace all matches, or prepare a list of matches without replacing anything. You can inspect the context, and replace or revert individual matches.
- Collect data: Extract search matches or substitutions into one or more new files.

2. Select the kind of search term you want to use from the “search type” list. Again, the Action pane will rearrange itself so you can enter that kind of search term. PowerGREP supports three 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.
- 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 items: Display only files matching all search terms. Only available when the action type is "find files" or "display search matches", and the search type is a list.

4. When the action type is “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. 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.

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 "find files", this option has a different meaning, since the main action does not involve individual search matches. In "find 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. Specify target file options. If the action type is “find files”, PowerGREP can save the file names of the files that are found into a target file. If the action type is “search-and-replace”, the target settings determine if the replacements are made in the files being searched through, or in copies of those files. When the action type is “collect data”, PowerGREP will save search matches or collected text to into either a single target file for the whole action, or into one file for each file searched through.

9. 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.

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

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

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

6. Interpret Search Results

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

1. Set the display options you want. After changing one or more options, click the Update button on the toolbar.

- 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 (i.e. the section they were found in).
- Group search matches: Group matches per file to display the matches of each file, using the name of the file as a header. Group unique matches to see identical matches only once per file, or only once for the whole result set.
- Display totals: Show totals for the whole operation before or after the results. When grouping per file, show totals for each file before or after the matches in that file. When grouping unique matches, toggle indicating the number of times each match was found.
- Sort files: When grouping per files, sort the files alphabetically by full path, or by number of search matches found in each file.
- Sort matches: Display matches in the order they were found (when not grouping unique matches), or sort them alphabetically (grouping or not), or by the number of times each unique match was found (grouping or not).
- Display replacements: For "search-and-replace" and "collect data" actions, show either the original search match, or the replacement or collected text, or both. When showing both, you can show both on the same line, or on separate lines. When showing them separately, and showing context, the context is shown twice.

2. Use the Word Wrap item in the Results menu to toggle wrapping of long lines in the results.

3. Double-click on a match to open it in the file editor to inspect its contents. You can make or revert replacements in the file editor. When grouping unique matches, double-clicking a match shows the individual match results at the bottom of the Results pane. Double-click on an individual match to see its context in the file editor.

If you turned on “group identical matches” on the Action pane, 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.

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 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, unless you used Quick Execute, and unless the action type is “find files”, and unless 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 clicking on it, and pressing the Replace button on the Editor toolbar. The match is instantly replaced with the replacement text that you prepared in the action definition. To replace the match with something else, simply use the Backspace or Delete key on the keyboard, and type in the new text.

6. If you executed a search-and-replace, you can restore the original text by clicking on the match, and pressing the Revert button on the toolbar. You can also restore individually replaced matches (see step 5) this way.

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. E.g. F9 is only associated with the Action|Preview menu item. At any time, pressing F9 will start a preview of the action.

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

See the editor reference for a list of key combinations you can use to edit text in multi-line text boxes in PowerGREP. In addition to the editor box on the Editor pane, all boxes for search terms on the Action pane 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 pane by pressing the pane's keyboard shortcut as indicated in the View menu. E.g. press Ctrl+F2 to activate the File Selector, Ctrl+F3 for the Action pane, and Ctrl+F5 for the Results. These keyboard shortcuts, and the associated menu items, activate the pane 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 Ctrl+F1 through Ctrl+F7 key combinations will greatly speed up your work with PowerGREP.

Selecting Files and Folders with The Keyboard

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

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

9. Regular Expression Quick Start

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

Text Patterns and Matches

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

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

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

Literal Characters

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

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

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

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

Character Classes or Character Sets

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

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

Typing a caret after the opening square bracket will negate the character class. The result is that the character class will match any character that is *not* in the character class. «q[^x]» matches „qu" in "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. Usually, non-English letters and numbers are included.

Non-Printable Characters

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

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

If your regular expression engine supports Unicode, use «\uFFFF» to insert a Unicode character. E.g. «\u20A0» matches the euro currency sign.

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

The Dot Matches (Almost) Any Character

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

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

Anchors

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

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

Alternation

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

Repetition

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

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

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

Greedy and Lazy Repetition

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

Place a question mark after the quantifier to make it lazy. «`<.+?>`» will match „`<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 Backreferences

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

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

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

Unicode Properties

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

Lookaround

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

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

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

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

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

Part 2

PowerGREP Examples

1. Search Through File Names

The search terms you enter on the Action pane are always used to search through the contents of a file, and never to search through the name of a file. Use the File Selector to get list of files that have a certain search term in their name.

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

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

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

How to Search through Both Names and Contents

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

7. Leave the action type as "display search matches".
8. On the Action pane, enter the search terms you want to search for through the contents of the files.
9. Click the Preview button to run the action.

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

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

2. Find Email Addresses

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

Finding a Particular, Unspecified Email Address

1. Select the files you want to search through in the File Selector.
2. Start with a fresh action.
3. Leave the action type as "display search matches". 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]{2,4}\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 pane. 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 pane.
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 pane and repeat from step 8.

How to Get a List of all Email Addresses

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

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

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

3. 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 pane 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.

4. Boolean Operators “and” and “or”

Many search tools use the boolean operators "and“ and ”or". 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.

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 items" 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 Items

If you turn on the option "list only files matching all items", 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 items" is *on* to imply "and" between the search terms.
5. Enter the first search term. Click on the green plus button to add additional search terms to the list.
6. Click the Preview button to run the action.

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

List Only Lines Matching All Items

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 items" option that appears after choosing line by line sectioning. This implies "and" between the search terms.
7. Click the Preview button to run the action.

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

Combining “and” and “or”

The “list only files matching all items” 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 items" 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.

5. 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 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`»

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

6. 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 "display search matches".
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. In the "file sectioning" list, select "line by line".
6. Turn on the option "list only sections matching all items". This option only appears when you've entered multiple search terms. It tells PowerGREP to only display matches from sections (lines, in this case) in which all the words we're searching for can be found.
7. Click the Preview button to run the action.

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

7. 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 "collect data".
4. Enter one or more search terms. Do not enter any text to be collected.
5. In the "file sectioning" list, select "line by line".
6. Turn on the option "collect/replace whole sections". This makes sure lines will be extracted as a whole.
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 the “delimiter for collected items” to “Line break”. Then click the Collect button.

To delete the lines from the original files instead, set the action type to “search-and-replace” in step 3 above. Do not enter any replacement text.

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.

8. 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 pane, 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.

9. Add a Header and Footer to Files

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

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

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

10. 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. 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.
3. Select "Update copyright statements" in the library, and click the Use Action button.
4. Set the replacement to “`\1-`” (backslash, one, dash) followed by the current year.
5. Set the target and backup file options as you like them.
6. Click the Preview button to run a test.
7. If all looks well, click the Replace button to actually update the copyright statements.

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

How The Regular Expression Works

The regular expression we used is «`(copyright +(©|\(c\)|&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 +(©|\(c\)|&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.

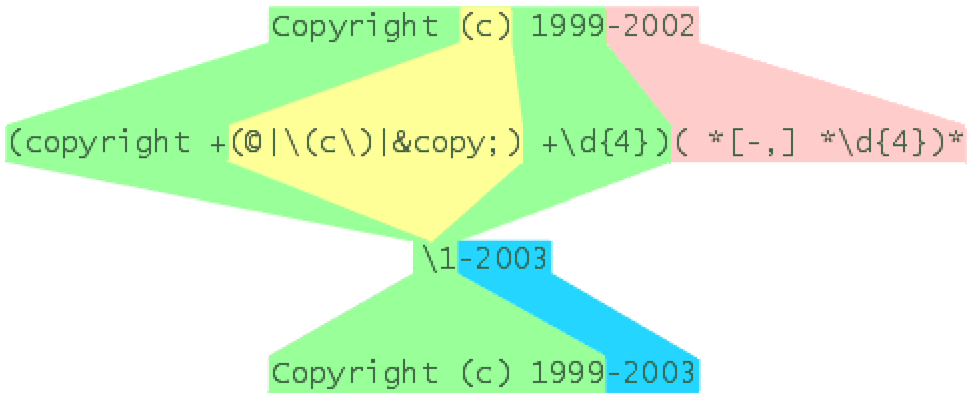
The first part of the regular expression will successfully match a copyright statement in the form of "Copyright (c) 1999".

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

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



11. 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.

It is obvious that we will need a regular expression to do this magic. We need to keep in mind that PowerGREP can only replace one piece of text with another, and not arbitrarily copy some text from one place to another. So, our regular expression will have to match the <TITLE> tag, the <H1> tag, and everything between the </TITLE> and <H1> tags. In the replacement, we will update the <TITLE> tag, and reinsert all the other stuff matched by the regex unmodified.

Note that I will be assuming here that a possibly empty <TITLE> tag is already present in the HTML code. Further below, I will relax this assumption step-by-step and expand the regular expression to support increasing levels of web author sloppiness. I am doing this in steps not only because that makes things easier to understand for you. It is also easier for me to expand my regular expression step-by-step.

The regular expression we need is actually quite easy:

«`<TITLE>.*?</TITLE>(.*?)<H1>(.*?)</H1>`»

The «`.*?`» part will match whatever is between the tags. It is important to use the question mark to make the star lazy. This will not make any functional difference unless there is more than one </TITLE>, <H1> or </H1> tag, but will speed up the regex by eliminating backtracking.

By turning on the option “dot matches newline”, our regular expression can easily span across multiple lines.

The replacement is easy too. We insert the part matched between the <H1> tags between the <TITLE> tags, and reinsert the rest unmodified. So we will replace with:

«`<TITLE>\2</TITLE>\1<H1>\2</H1>`»

That's it! You can easily run this action on your own HTML files:

1. Select the files you want to search through in the File Selector.
2.
3. 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.
4. Select "Update HTML title tags" in the library, and click the Use Action button.
5. Set the target and backup file options as you like them.
6. Click the Preview button to run a test.
7. If all looks well, click the Replace button to actually update the 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 instead:

«`(<TITLE>.*?</TITLE>|(</HEAD>))(.*?)<H1>(.*?)</H1>`»

Here we are exploiting the fact that the regular expression engine is eager. When the regex engine can match the first part of the alternation (i.e. it found a title tag), it will consider the entire alternation as matched. It will not attempt to match any further parts of the alternation, but continue right away with «`(.*?)<H1>(.*?)</H1>`».

An important point to remember about the regex engine and its eagerness, is that it will process the file from the first to the last byte and attempt to match the entire regular expression at each byte before continuing with the next byte. Our regular expression works because if «`<TITLE>.*?</TITLE>`» can be matched anywhere in a (more or less valid) HTML file, it will be matched before (i.e. to the left of) the place where "</HEAD>" could be matched. If </HEAD> would appear in a file before the title tag, our regular expression would match where the </HEAD> appears.

As you can see, I put the </HEAD> tag in the regular expression between parenthesis. This is to capture it in a backreference so I can insert it into the replacement (by typing in \2). We can specify only one replacement text in PowerGREP, so it has to work properly whether the “title” part of the option, or the “head” part of the option was matched. If the “title” part was matched, the backreference for “head” will be empty. If the “head” part was matched, the backreference will contain "</HEAD>". Note that it is perfectly valid to use empty backreferences in the replacement text.

Here is the replacement text:

“`<TITLE>\4</TITLE>\2\3<H1>\4</H1>`”

The part “`\3<H1>\4</H1>`” simply re-inserts the <H1> tags and everything matched between them and before them, like we did in the previous example. The part matched by «`(<TITLE>.*?</TITLE>|(</HEAD>))`» is replaced with “`<TITLE>\4</TITLE>\2`”.

If we matched the “title” part of the option, \2 will be empty and we are simply substituting the title tag. We do not need to worry about the </HEAD> tag in that case, because it's included in \3.

If we matched the “head” part of the option, the </HEAD> tag is re-inserted by \2 and the title tag will be added right before the </HEAD> tag.

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

12. Replace HTML Tags

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

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

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

13. Replace HTML Attributes

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

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

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

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.

14. 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 main part of the action as usual.

Search Through Comments and Strings Only

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

Don't Search Through Either Comments or Strings

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

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

Search Through Comments Only, or Strings Only

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

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

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

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

15. 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`”

16. 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 the delimiter for collected items as "Line break". PowerGREP will insert the delimiter, if any, 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.

17. 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 action definition. This makes it easy to collect both part of the section (e.g. its header), and part of the item, for each item found in each section.

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

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

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

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

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

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

1. Select the files you want to search through in the File Selector.
2. Start with a fresh action.
3. Select "search for sections" from the "file sectioning" list. Leave the section search type as "regular expression".
4. 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'.
5. Set the action type to "collect data". Leave the search type as "regular expression".
6. In the Search box, 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.

18. 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+\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) [^ "]+ HTTP/[0-9.]+"`»
5. Status code returned by the server (200): «`[0-9]+`»
6. Number of bytes served (8669): «`[-0-9]+`»
7. Referring URL, between double quotes: «`"[^"]*"`»
8. User agent, between double quotes: «`"[^"]*"`»

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

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

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

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

Collecting Referring URLs

1. Select the log files you want to search through in the File Selector.
2. Set the action type to "collect data". Leave the search type as "regular expression".

3. Search for the regular expression «`"GET [^ ?"]+?\.html?[^ "]* HTTP/[0-9.]+"\s+[0-9]+\s+[-0-9]\s+"([^"]*)"`» which captures the referring URL in backreference one.
4. Enter “`\1`” in the Collect box, to collect referring URLs.
5. Turn on "group identical matches" and "group results for all files".
6. Set "sort collected matches" to "by decreasing totals" so we can see which URLs are most important.
7. Set "minimum number of occurrences" to 10 or higher, to avoid collecting to many URLs that are of no importance.
8. 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 [^ ?"]+?\.html?[^ "]* HTTP/[0-9.]+`» to restrict matches to page hits only. This makes sure the statistics aren't skewed, since most browsers send the same referrer information when loading images as when loading the page containing those images.

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

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

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

19. Extract Google Search Terms from Web Logs

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

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

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

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

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

1. Select the log files you want to search through in the File Selector.
2. Open the PowerGREP.pgl library file included with PowerGREP. You can find it in the folder where PowerGREP is installed, c:\Program Files\JGsoft\PowerGREP3 by default.
3. Select "Inspect Apache web logs - Google search terms" in the library, and click the Use Action button. This sets up the regular expression and extra processing as I explained above.
4. Click the Preview button to run the action.

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

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

The regular expression has two capturing groups. The first one matches the file name in the HTTP request, which is the landing page. The second group matches the Google search terms. The text to be collected uses "`\l2`" (backslash ell two) to collect the search terms converted to lowercase, and «`\1`» to collect the landing page.

20. 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". Leave the search type as "regular expression".
3. 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.
4. 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.
5. 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.
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. Click the Collect button to run the action.

The file created by PowerGREP will not be a full HTML file. It will only contain link to all files. To turn it into a full HTML file, either edit it in a text editor, or run a second PowerGREP action to add a header and footer to the file.

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:

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

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

21. Make Sections and Their Contents Consistent

This example illustrates how you can use named capturing groups to carry over regex matches between the three parts of a PowerGREP action: the main action definition, the extra processing and the file sectioning. Named capturing groups in the file sectioning carry over to the main action and the extra processing. Named capturing groups in the main action carry over to the extra processing.

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

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

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

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

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

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

Updating the Heading Tags Themselves

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

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

1. Select the files you want to search through in the File Selector.
2. Start with a fresh action.
3. Set the action type to "search-and-replace". Leave the search type as "regular expression".
4. In the Search box, 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.
5. In the Replace box, enter “`<h\0>${tag}</h\0>`”
6. Select "search for sections" from the "file sectioning" list. Leave the section search type as "regular expression".
7. 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".
8. Turn on the option "collect/replace whole sections".
9. Set the target and backup file options as you like them.
10. Click the Preview button to run a test.
11. If all looks well, click the Replace button to update the headers.

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

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

22. 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. Click the Collect button to run the action. PowerGREP will do all the hard work in creating the navigation bar. You only need to make two minor edits.
14. Double-click on one of the search matches highlighted in the results. This is simply a quick way to open the target file in PowerGREP's built-in file editor.
15. At the start of the file, add the line
    `<? function navbar($mainpage) {`
16. At the end of the file, add the line
    `} ?>`
17. Save the file, and your navigation bar is ready for inclusion into your web site.

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.

Instead of manually editing the target file, you can use a second PowerGREP action to add the function header and footer to the file.

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.

23. 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.

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 will explain 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 will also explain 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 will be underlined in blue, like hyperlinks on a web site. If you click on such a link, the assistant will move 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 will only be able click on a link when the assistant is describing the control that has keyboard focus. After you click, the assistant will automatically describe the newly activated control.

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 will use the Assistant panel to deliver the message. If you closed the assistant, it will automatically pop up in the place it was last visible.

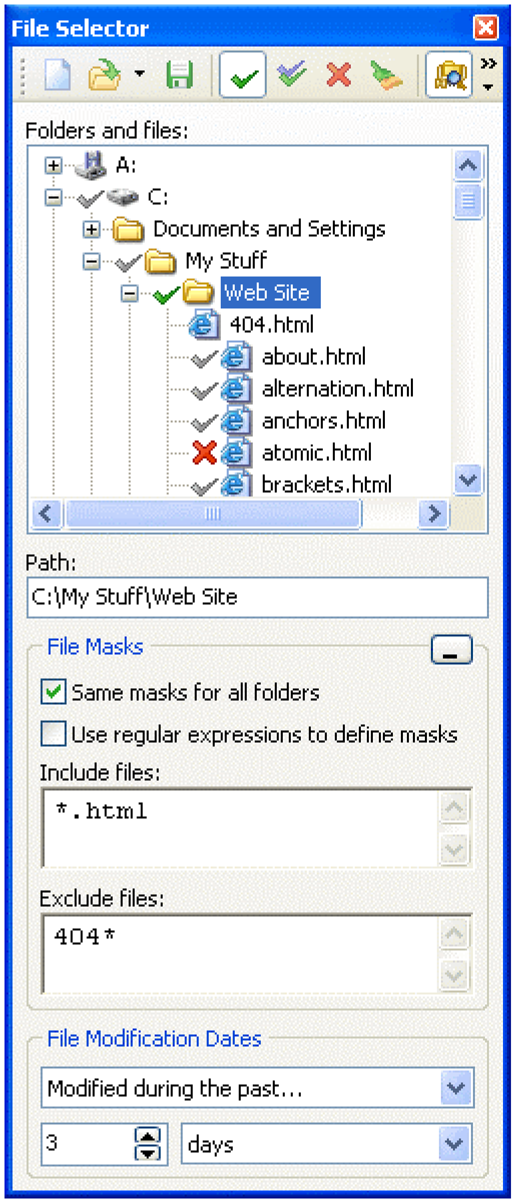
You can recognize error messages by their bold red headings. Hints have black headings. The assistant will continue to show the error message until you resolve the problem, or dismiss the error by clicking the Dismiss link below the error message. Meanwhile, the assistant will not display hints or descriptions.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

Folders and Files



File Selector

Folders and files:

- A:
- C:
  - Documents and Settings
  - My Stuff
    - Web Site
      - 404.html
      - about.html
      - alternation.html
      - anchors.html
      - atomic.html
      - brackets.html

Path:

C:\My Stuff\Web Site

File Masks

☑ Same masks for all folders

☐ Use regular expressions to define masks

Include files:

```
*.html
```

Exclude files:

```
404*
```

File Modification Dates

Modified during the past...

3 days

The "folders and files" tree view shows all drives, folders and files on your computer. The only files not visible in the tree are those you excluded from all actions in the file selection preferences. By default, only backup files are excluded.

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

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

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

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

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

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

When you change your mind about including or excluding a file or folder, select it and remove the mark with the Clear File or Folder command. To start from scratch, select Clear in the File Selector menu. Clearing a file or folder is not the same as excluding it. If you exclude a file or folder, it won't be 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.

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.

Entering Paths with The Keyboard

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

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

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

File Masks

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

In a traditional file mask, the asterisk (*) represents any number (including none) of any character, similar to «`.*`» in a regular expression. The question mark (?) represents one single character character, similar «`.`» in a regular expression. E.g. the file mask `*.txt` tells PowerGREP to include any file with a .txt extension. You can delimit multiple masks with semicolons. To search through all C source and header files, use `*.c;*.h`.

Traditional file masks also support a simple character class notation, which matches one character from a list or a range of characters. E.g. to search through all web logs from September 2003, use a file mask such as `www.200309[0123][0-9].log` or `www.200309??.log`.

If you choose to use regular expressions to define masks, you have the full regular expression syntax at your disposal. Semicolons, which cannot appear in file names, delimit multiple regular expressions.

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

When you do not specify any file masks for a folder, all files in that folder are included. If you specify an inclusion mask, only files that match the inclusion mask will be included. If you specify an exclusion mask, all files matching the exclusion mask will be excluded form the next action. The exclusion mask takes precedence. If you specify both, files matching both will not be searched through.

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

In the screen shot above, the folder “My Stuff” was marked for inclusion with Include File or Folder, as evidenced by the green tick next to it. The file "404.html" is not included, because it matches the exclusion mask for the folder “My Stuff”. The file "atomic.html" is not included either, even though it matches the inclusion mask, because it was excluded with the Exclude File or Folder command.

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

File Modification Dates

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

If a file with a green tick is excluded because of its modification date, the green tick will disappear. However, the File Selector will remember that you marked the file. If you cancel the file modification filter, the green tick will automatically reappear. PowerGREP can treat modification dates in several ways:

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

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

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

3. File Selector Menu

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

Clear

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

Open

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

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

Save

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

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

Favorites

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

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

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

Include File or Folder

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

Include Folder and Subfolders

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

Exclude File or Folder

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

Clear File or Folder

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

Clear Folder and its Files and Subfolders

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

Mark Files with Search Results

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

Search through Archives

This option is on by default. Toggle it to enable or disable searching through archives. When on, archives are treated as folders, and PowerGREP will search through the files inside the archive. When off, archives are skipped. The only exception is a find files action without a search text, which will list the archive files themselves when the option to search archives is off. Currently, PowerGREP can only search through zip archives.

Search through Binary Files

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

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

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

Search Only through Files with Results

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

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

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

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

Show All

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

Show Included Files

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

Show Files with Results

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

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.

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 under the JGsoft brand name. EditPad is available at http://www.editpadpro.com/.

4. Action Reference

The Action pane is the place where you define the task that PowerGREP will execute. The Action pane 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 pane to change its appearance.

All the options on the Action pane are laid out logically, from top to bottom. Changing an option will only show or hide other options below it, never above it. When defining an action, work your way through the Action pane from top to bottom.

The Action pane consists of five sections:

1. Main action definition: You will always use this part. It defines the core action to be executed. You can find files, display search matches, search-and-replace or collect data.
2. Extra processing: Only used for search-and-replace and collect data actions. You can apply an extra search-and-replace to the replacement text or the text to be collected in the main action.
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.
4. Target and backup files: Tell PowerGREP how to save results or modify files. Not available when displaying search matches.
5. 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 on the Action toolbar to tell PowerGREP to execute it. The Preview button is the safest one, since it will never modify any files, or do anything else you might regret.

5. Search Terms and Options

The three major parts of an action, the main action definition, the extra processing settings and the file sectioning all require search terms. All three parts provide the same search types and options, which you can specify independently. The only choice that must be consistent among the three is whether you are searching for literal text or a regular expression on one hand, or whether you are searching for binary data on the on the other hand. If you change one part's search type from text or regex to binary, or vice versa, PowerGREP will automatically change the other two.

Search Types: Text, Regex or Binary

PowerGREP can search for three 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.
- 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.

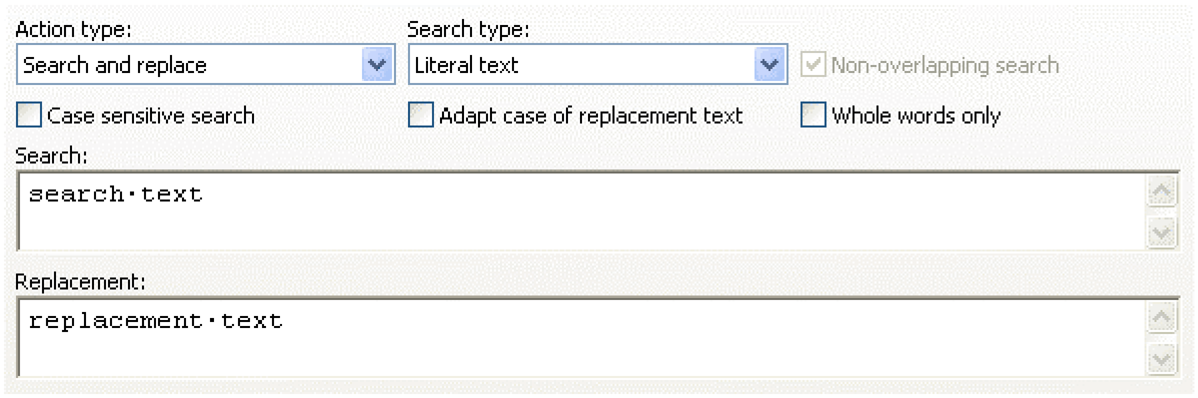
If you are new to regular expressions, the regular expression tutorial in this book will teach you everything you need to know. Given an hour or two of practice, you will soon be up to speed.

You probably also want to have a look at RegexBuddy (available separately), which makes it much easier to create 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.

All three search types allow you to use path placeholders. These 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 path placeholders in the action & results preferences if they conflict with text you're searching for.

Search Types: Single, List or Delimited

All three kinds of search items can be entered in three ways:



Action type: Search and replace

Search type: Literal text

[✓ Non-overlapping search]

Case sensitive search

Adapt case of replacement text

Whole words only

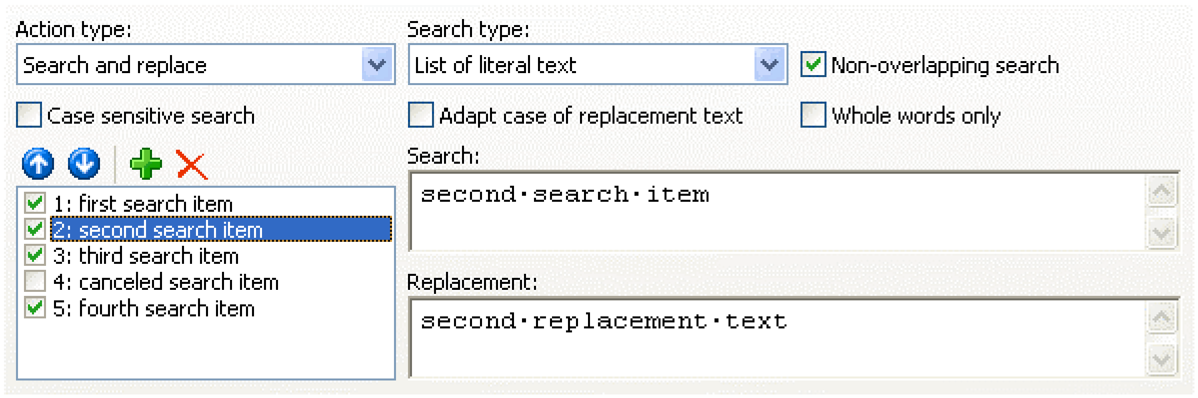
Search:

```
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).



Action type: Search and replace

Search type: List of literal text

[✓ Non-overlapping search]

Case sensitive search

Adapt case of replacement text

Whole words only

- [✓ 1: first search item]
- [✓ 2: second search item]
- [✓ 3: third search item]
- 4: canceled search item
- [✓ 5: fourth search item]

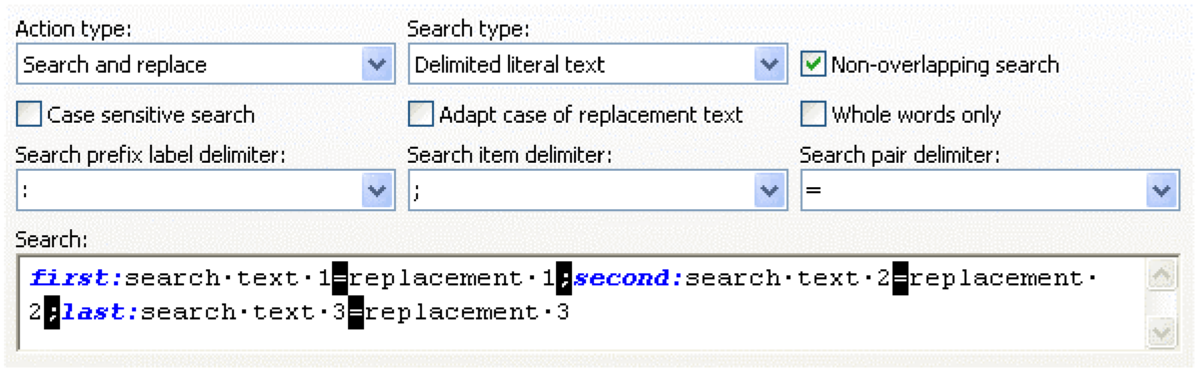
Search:

```
second·search·item
```

Replacement:

```
second·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.



Action type: Search and replace

Search type: Delimited literal text

[✓ Non-overlapping search]

Case sensitive search

Adapt case of replacement text

Whole words only

Search prefix label delimiter: :

Search item delimiter: ;

Search pair delimiter: =

Search:

```
first:search·text·1=replacement·1;second:search·text·2=replacement·
2;last:search·text·3=replacement·3
```

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. Simply paste them into the edit box, and select the delimiters. 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.

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” is “mammal”.`”, and that you entered the literal search terms «`category`» and «`cat`», in that order. Assuming “whole words only” is off, executing this search would find „`cat`” twice, and would not match «`category`» at all. The reason is that 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.

A non-overlapping search of literal text and binary data always finds the leftmost (in the text), shortest (in the list) match. The order of the search items does not matter.

If we searched for the regular expressions «`[Cc]ategory`» and «`[Cc]at`», it would match „`category`” once, and „`cat`” also once. The regular expression engine “walks” through the search text one character at a time. At each position, it tries all the regular expressions in the list, one after the other. As soon as one regex matches, the match is accepted. The remaining regexes are not tried at that position. If we swap both regexes in the list, PowerGREP would find „`cat`” twice, just like with the non-overlapping literal search.

A non-overlapping search of regular expressions always finds the leftmost (in the text), earliest (in the list) match. The order of the search items is significant.

Note that if you set the search type to regular expressions, but then enter plain text for all search terms, PowerGREP will automatically treat the search terms as literal text rather than regular expressions. This speeds up the search quite a bit, but does have the side effect of making the order of the search terms irrelevant for non-overlapping searches.

If you turn off the non-overlapping option, both the literal text example and the regular expression example yield the same results: „`cat`” is matched twice, and „`category`” matches once. The first match of „`cat`” overlaps entirely with the sole match of „`category`”.

In a search-and-replace action or extra processing search-and-replace, turning off non-overlapping search as 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”.`” (for both search pairs).

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. E.g. with searching for «`two cats`» case insensitively, and replacing with “`one dog`”, „`Two cats`” will be replaced 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.

Turn on "whole words only" to match only complete words. With this option on, searching for «`cat`» will *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

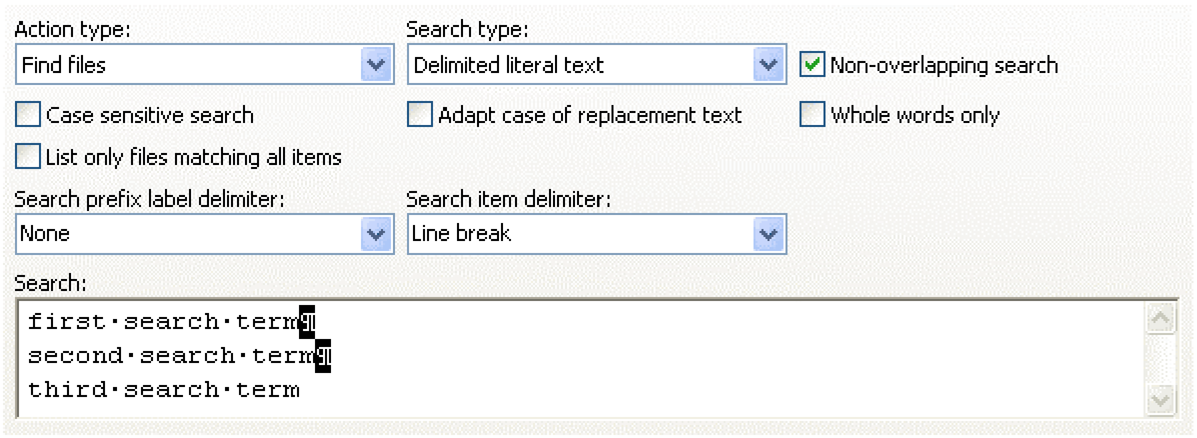
When using a list of search terms, the above options apply to all search terms. When using regular expressions, you can use mode modifiers to toggle the some of the options for individual regular expressions (or even parts of regular expressions). Put «`(?i)`» in front of a regular expression to make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

6. Action Definition

The top part of the Action pane is where you define the main action to be executed. Start with selecting the action type from the drop-down list in the upper left corner. All the other available options depend on the action type you've selected.

All action types require search terms. Before entering search terms, select the search type you want. See the Search Terms and Options chapter for more information.

Find Files



Action type: Find files
Search type: Delimited literal text
[✓] Non-overlapping search
[ ] Case sensitive search
[ ] Adapt case of replacement text
[ ] Whole words only
[ ] List only files matching all items
Search prefix label delimiter: None
Search item delimiter: Line break
Search:

```
first·search·term¶
second·search·term¶
third·search·term
```

Use the "find files" action type when you want to get a list of files that contain, or do not contain, one or more of the search terms, or all of the search terms. “Find files” is the fastest way to search. As soon as PowerGREP finds a match, it will continue with the next file. “Display search matches” on the other hand, requires PowerGREP to search through all files entirely to get a list of all matches in the files. If you just want to see which files match, and don't care about the individual matches, “find files” is the way to go.

The option "list only files matching all items“ is available for ”find files" actions when the search type is a list of search terms. If you turn on this option, PowerGREP will list only files in which it can find all of the search terms.

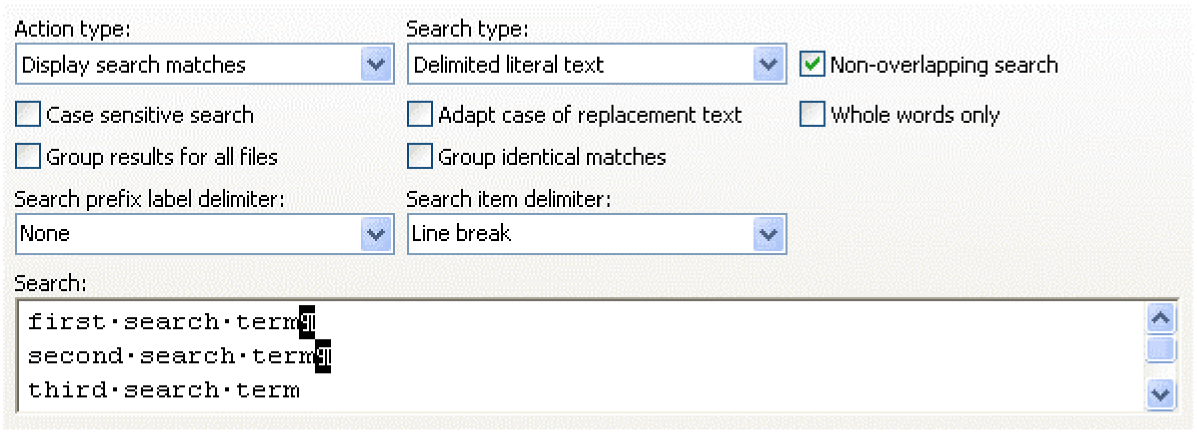
The "invert search results“ option in the file sectioning part of the action definition has a different meaning for a ”find files" action than for the other action types. With “find files”, inverting search results inverts the way entire files are matched, whether files are being sectioned or not. With the other action types, inverting search results inverts which sections are matched, and the option is only available when sectioning.

Turn on “invert search results” to list those files in which none of the search terms can be found. Turn on both “invert search results” and “list only files matching all items” to list all files in which some, but not all of the search items can be found, in addition to files in which none of the search items can be found. As the name of the option suggests, turning on “invert search results” makes PowerGREP list all files that would not be listed when the option is off, and vice versa.

“Find files” offers the following target types:

- Do not save results to file: The results are simply listed on the Results pane. No action is taken.
- Save list of matching files to file: The list of files displayed on the Results pane is saved into a text file of your choice. You can choose the encoding and line break style of the file, as well as the delimiter put between each file name.
- Copy matching files: Files listed in the results are copied into a folder or archive of your choice.
- Move matching files: Files listed in the results are copied into a folder or archive, and then removed from the original location.
- Delete matching files: Files listed in the results are permanently deleted. This action cannot be undone. Be careful!

Display Search Matches



Action type: Display search matches
Search type: Delimited literal text
[✓] Non-overlapping search
[ ] Case sensitive search
[ ] Adapt case of replacement text
[ ] Whole words only
[ ] Group results for all files
[ ] Group identical matches
Search prefix label delimiter: None
Search item delimiter: Line break
Search:

```
first·search·term¶
second·search·term¶
third·search·term
```

The "display search matches" searches all files completely, to find all search matches in the file. The Results pane will list all individual search matches in all files, with the option to sort and group matches. If you open a file that was searched through in PowerGREP's file editor, the editor will highlight all search matches in the file.

The option "list only files matching all items" is available when the search type is a list of search terms. If you turn on this option, PowerGREP will only show search matches for those files in which it can find all of the search terms. Any search matches found in partially matching files are discarded.

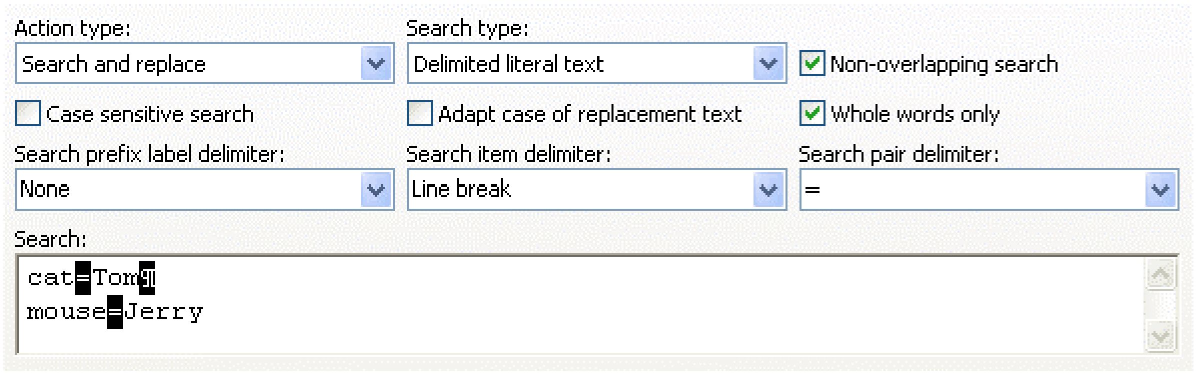
Use the “display search matches” action type when you want to inspect the context of individual search matches. Since storing information about each and every match requires a great deal of memory, PowerGREP will only collect the first 100,000 matches. The search will continue after the first 100,000 matches, but PowerGREP will only count the number of matches in each file. If you “quick search” a “display search matches” action, PowerGREP will only count the number of matches for all files. The editor will not highlight matches when PowerGREP only counts them.

A good way to deal with large numbers of search results is to turn on the option to group identical matches. Then PowerGREP does not keep track of each and every match, but only of each unique match in every file, counting how many times it occurs. If you also turn on "group results for all files", PowerGREP only keeps track of each unique match once for all files searched through. This reduces memory usage considerably. It also means the editor will not be able to highlight any matches, since individual match information was not saved.

There is an important difference between turning on “group identical matches” on the Action pane, versus grouping identical matches on the Results pane. Grouping them in the action makes PowerGREP discard individual match information. Memory usage is reduced, and the editor won't highlight matches. Grouping them after the fact on the Results pane only changes the display, not the match results. The editor will still highlight individual matches. Double-clicking on a grouped match in the results will give you a list of all occurrences of that match.

The “display search matches” action type does not offer any target settings. As the name suggests, it is for display only. If you want to save individual or grouped matches to a file, use the “collect data” action type instead.

Search-And-Replace


Action type: Search and replace
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 item delimiter: Line break
Search pair delimiter: =
Search:

```
cat=Tom
mouse=Jerry
```

A search-and-replace action searches to find all search matches, and then replaces each match with something else. When the search term is a regular expression, you can reuse part or all of the search match in the replacement text by using backreferences. E.g. search for «`http://\S+`» and replace with “`<a href="\0">\0</a>`” to convert all URLs in a file into HTML anchors.

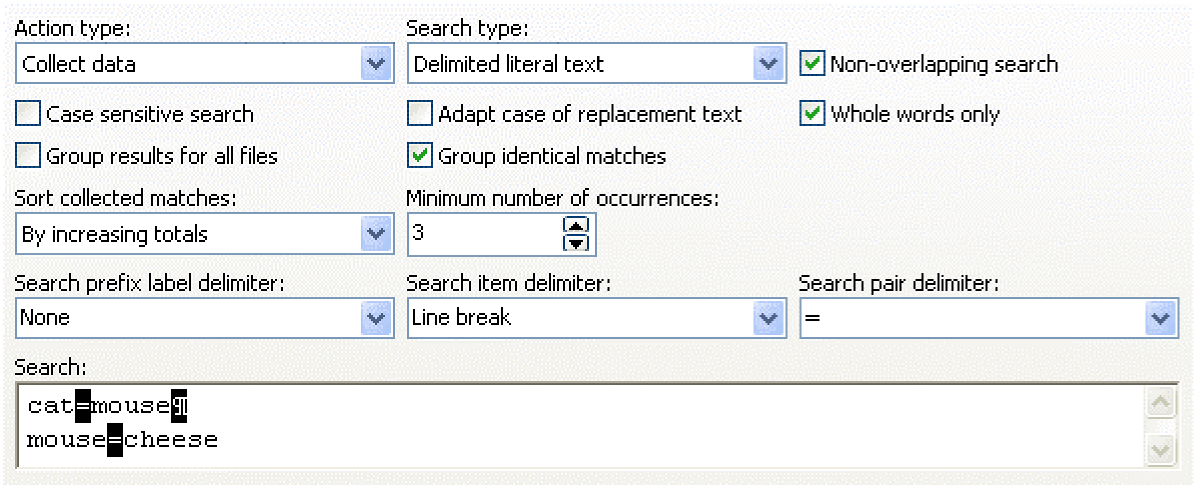
In addition to using backreferences, you can make the replacement text dynamic by applying extra processing. Extra processing consists of an additional search-and-replace run on the replacement text.

If you want to conditionally replace search matches, use the file sectioning feature to search for the matches you may want to replace. Turn on the option to replace whole sections. In the main part of the action, set the search terms to the conditions you want to use to decide to replace a section or not.

Three target types are available:

- 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.

Collect Data



```
cat=mouse
mouse=cheese
```

A "collect data“ action works much like a ”display search matches" action, except that the intention is not just to look at the matches in PowerGREP, but to save them to one or more files. As part of the search terms, you can specify the text to be collected, just like you would specify the replacement text in a search-and-replace action. You can make use of backreferences and extra processing just the same. You can use file sectioning to conditionally collect search matches (or rather: sections).

The "group results for all files" option determines whether all search matches will be saved into a single file, or whether a separate file is created for each file that was searched through. Toggling this option will change the target type and vice versa.

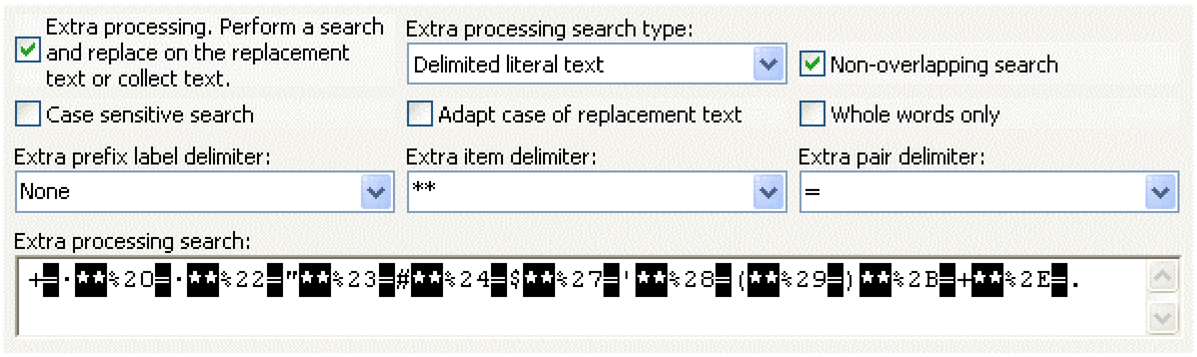
By default, the collected texts are saved in the order the corresponding search matches are found. Turn on "group identical matches" to save only unique items. Note that in the context of a “collect data” action, it is not the search match but the text to be collected that is checked for uniqueness. PowerGREP tracks uniqueness per output file. When grouping results for all files, identical texts are saved only once for the entire operation. Otherwise, the same text may be saved more than once during the whole operation, but only once per file searched through.

Two additional options become available when grouping identical matches. You can choose to sort collected matches either alphabetically, or by the number of times the collected text would have occurred when not grouping identical texts. You can also specify a minimum number of occurrences. Texts that are collected fewer times than the minimum are not saved into the output file at all. They won't appear in the Results pane either. Four target types are available:

- Do not save results to file: Display the results in PowerGREP only.
- Save results into a single file: Save the text collected from all the files into a single new file.
- 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.

Selecting the second target type automatically turns on “group results for all files”. The last two target types automatically turn that option off.

7. Extra Processing


[✓] Extra processing. Perform a search and replace on the replacement text or collect 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: None
Extra item delimiter: **
Extra pair delimiter: =
Extra processing search:

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

The “extra processing” part of the Action pane is only visible when you've set the action type to “search-and-replace” 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 will appear as plusses, and plusses will appear as %2B. With an extra processing step, you can search-and-replace the plusses back into spaces, etc. making the results a lot more readable.

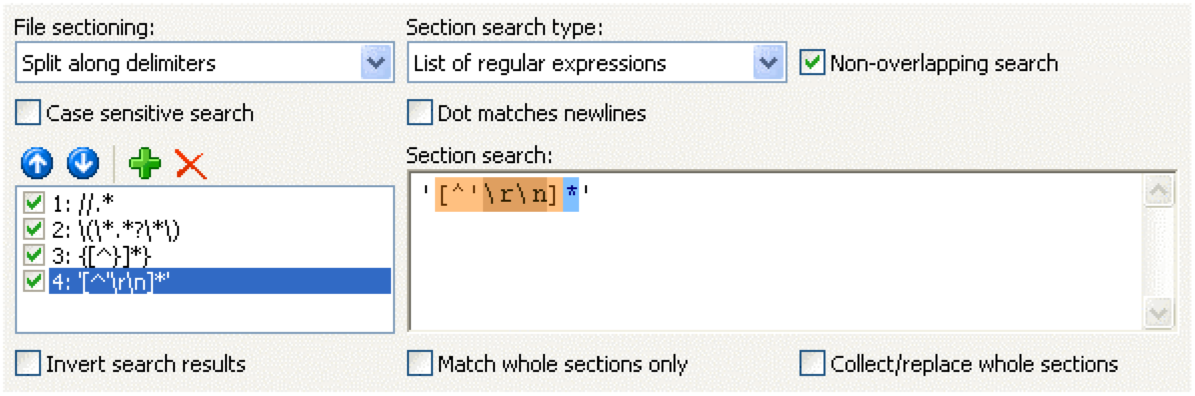
Do not confuse extra processing with entering a list of search terms in the main part of a search-and-replace action. Each search-and-replace in the main part of the action will be run on the entire file, or the entire section. The extra processing is only run on the replacement text, just before the 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.

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.

8. File Sectioning



File sectioning: Split along delimiters

Section search type: List of regular expressions

[✓] Non-overlapping search

[ ] Case sensitive search

[ ] Dot matches newlines

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

Section search:

```
'[^'\r\n]*'
```

[ ] Invert search results

[ ] Match whole sections only

[ ] Collect/replace whole sections

With the "file sectioning"part of the action definition, you can specify which parts of each file PowerGREP should process. Sections also determine what the context of a match is on the Results pane.

The default is "do not section files". PowerGREP will search the whole file without any boundaries. Search matches can span multiple lines, or even the entire file. The context for each match is the line that contains it. If a match spans across lines, all the lines it spans are the context of the match. The lines will be numbered sequentially (1, 2, 3, ...) in the results. The section numbers will not reflect actual line numbers. Not sectioning files is the fastest option.

"Do not section; count line numbers" does the same as the preceding option, except that the section numbers in the results will reflect actual line numbers. These line numbers will correspond with the line numbers shown when you open the file in PowerGREP's built-in file editor and turn on the line numbers.

The "line by line" option also counts line numbers, but actually divides the file into sections, one line each. The line breaks between the lines are not part of the section. This means that search terms and regular expressions will not be able to span multiple lines.

Use line by line sectioning when you want to match, collect or replace whole sections, or want to find lines in which the search terms are *not* present.

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.

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.

Testing File Sectioning

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

Sectioning Options

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

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

The option "invert search results" makes the main part of the action match sections in which the search terms can *not* be found. The entire section will be treated as the search match. When inverting search results, whole sections must be collected or replaced with a common replacement text.

Inverting search results has a different meaning in “find files” actions. The option will invert the whole list of files, not individual sections, since “find files” does not list individual matches or sections in the search results.

Finally, "list only sections matching all items" tells PowerGREP to 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. If you turn on this option for a search-and-replace action, whole sections must be replaced with a common replacement text.

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 will section the file only once, and sections will 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 and 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.

9. Target and Backup Files

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

Target Types

The available target types depend on the kind of action you have prepared. For a find files action, five target types are available:

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

A search-and-replace 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.

Four target types are available when collecting data:

- Do not save results to file: Display the results in PowerGREP only. No target and backup options need to be specified.
- Save results into a single file: Save the text collected from all the files into a single new file. Specify a single file as the target location.
- Modify original files: Save the text collected in each file to the file, overwriting the original file. The end result is a search and replace that deletes unmatched parts of the file.
- Save one file for each searched file: Create a new file for each file that has one or more search matches. Save the text collected from the file into the new file.

Target Destinations

When copying or moving files, and when creating a new file for each file searched through, you can choose the target destination type and corresponding target location.

- Single folder: Place all files into a single folder.
- Folder tree: Place all files into a specific folder. If you marked a folder in the File Selector to also include its subfolders, those subfolders will be recreated under the target folder.
- .zip archive: Place all files into a .zip archive. If you marked a folder in the File Selector 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. If the archive already contains a file with the same name, a backup copy of that file is created. If you set the backup type to "same folder as original", the backup copy is created inside the same .zip archive.
- Numbered .zip: Like the ".zip archive" option, except that if the .zip file 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. Click on the ellipsis (...) button to easily build the target path.

Backup Types

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.

First, select the naming convention backup files should use:

- 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. If two actions overwrite the same file, only the last action of the two can be undone.
- 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. If two actions overwrite the same file, only the last action of the two can be undone.
- 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. If two actions overwrite the same file, only the last action of the two can be undone.
- Path placeholders: Use path placeholders to dynamically build a full backup path for each file that needs to be backed up. Click on the ellipsis (...) button to easily build the target path. You should use %N% in the file name portion of the path to make sure existing backup files are not overwritten.
- 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 Destinations

Unless you disabled making backups, or selected the “path placeholders” or “hidden history options”, you can select the location backup files will be created in:

- Same folder as original: Place the backup file in the same folder as the file 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 Selector to also include its subfolders, those subfolders will be recreated under the target folder.
- .zip archive: Place all backup files into a .zip archive. If you marked a folder in the File Selector 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 .zip: Like the ".zip archive" option, except that if the .zip file already exists, a new archive will be created with a number added to its name.

10. Action Menu

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

Clear

Clears all settings on the Action pane. 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 pane 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.

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 pane.

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 pane.

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

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

Quick Execute

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

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

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

Abort

Aborts the action that is being preview or executed. The Results pane will show 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.

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.

11. Library Reference

A PowerGREP Library is a convenient place to store PowerGREP Actions for later reuse. You can open and save libraries on the Library pane in PowerGREP. In the default layout, you can access the library pane by clicking on its tab just below the menu bar.

To add an action to the library, use the Add to Library item in the Action menu or Action toolbar. 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 action definitions can be stored in PowerGREP libraries. They never contain file selections.

When you open a library, you will see a list of one-line descriptions of all the actions in the library. Click on one, and PowerGREP will show you the complete description, along with the action definition itself. This is the main advantage of storing actions in a library rather than saving them into separate PowerGREP action files. You can easily seek out the action you want to use by comparing the descriptions of different actions.

To reuse an action from the library, click the Use Action button on the Library toolbar. The action you selected is then copied to the Action pane. All settings on the Action pane are completely replaced with those of the action you selected.

If you use an action from the library, and then edit that action on the Action pane, the action stored in the library is *not* automatically updated. You can adapt reused actions to the situation at hand without messing up your carefully collected library. If you do want to update the action in the library, simply use the Add to Library item in the Action menu again. If you didn't change the description, PowerGREP will ask you if you want to replace the action already in the library. If you did change the description, both the old and new action will be present in the library.

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 instance at the same time.

12. Library Menu

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

Copies the action you selected in the library to the Action pane. All settings on the Action pane are completely replaced with those of the action from the library.

Delete Action

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

13. Results Reference

While PowerGREP previews or executes an action, the Results pane shows a progress meter. 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.

By default, the results of the action will only be shown when it has run to completion. Click the Update Results button to see the results gathered so far, or toggle the Automatic Update button to make PowerGREP update the results regularly. Updating the results slows down the search, but does allow you to inspect the results before the action has completed.

Changing Display Options

The Results pane 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 file names, and file totals. Do not display matches.
- Matches without context: Display search matches. File names and file totals are shown when grouping per file.
- Matches with section numbers: Display search matches along with the number of the section or line the match was found on.
- Matches with context: Display the sections or lines on which matches were found, and highlight the matches.
- Matches with context and section numbers: Display the sections or lines on which matches were found, along with the number of the section or line.

Group search matches: If you selected to display files and/or matches, you can also select how the files and matches should be displayed. When grouping unique matches, the context choices determine how matches are shown when double-clicking a unique match.

- 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.

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:

- Alphabetically, A..Z: Sort files in ascending alphabetic order of their full path names.
- Alphabetically, Z..A: Sort files in descending alphabetic order of their full path names.
- By increasing totals: Sort files by the number of matches found in each file, from least to most.
- By decreasing totals: Sort files by the number of matches found in each file, from most to least.

Sort matches:

- Show in original order: Show matches in the order they have in the file they were found in.
- Alphabetically, A..Z: Sort matches alphabetically, from A to Z.
- Alphabetically, Z..A: Sort matches alphabetically, from Z to A.
- By increasing totals: When grouping unique matches, sort them from least occurrences to most occurrences.
- By decreasing totals: When grouping unique matches, sort them from most occurrences to least occurrences.

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.

14. Results Menu

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

Clear

Clears the Results pane, 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 pane 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 pane 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.

Favorites

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

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

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

Export

Saves the results as shown on the Results pane to a plain text file or HTML file. The file will contain exactly the same text as shown on the Results pane, including file headers, totals, etc. If you export to an HTML file, the HTML file 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 documentation in HTML format.

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 pane. 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 pane.

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 pane. Using automatic update is only recommended when working with small results sets.

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 pane are wrapped across multiple lines. When off, you will need to use the horizontal scroll bar to see the remainder of long lines.

Edit File

Click on a file name in the results, and select the Edit File command to open that file in PowerGREP's built-in file editor. The editor can edit both text and binary files.

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.

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 under the JGsoft brand name. EditPad is available at http://www.editpadpro.com/.

15. 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 under the JGsoft brand name, just like PowerGREP.

The key advantage of PowerGREP's built-in editor is that it highlights search matches with 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.

16. 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.

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.

Favorites

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

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

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

Print

Print the file you are editing. A print preview will appear. The print preview allows you to configure the printer and page layout before printing. You can limit the printout to the selected part of the file, and/or to a particular range of pages.

Next Match

Highlights the next search match in the file. If there are no further search matches after the cursor position, the cursor is moved to the end of the file.

Previous Match

Highlights the previous search match in the file. If there are no search matches before the cursor position, the cursor is moved to the start of the file.

Make Replacement

Replaces the search match that the text cursor is on with the replacement text. You can only make replacements after previewing or executing a search-and-replace action, since PowerGREP needs to know which replacement text to use. The color of the highlighted match will change to indicate it has been replaced.

Revert Replacement

Reverts the replaced search match the text cursor is on to the original text. You can only revert replacements after previewing or executing a search-and-replace action. Quick Replace does not allow you to replace individual matches, since Quick Replace discards information about individual matches. The color of the highlighted match will change to indicate it has been reverted.

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 pane 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.

17. Undo History Reference

Whenever you execute an action that creates or modifies one or more files, PowerGREP automatically adds the action to the undo history. In the default layout, you can access the undo history by clicking on the Undo History tab.

To be able to undo an action, backup copies must have been created of files that were overwritten. Be sure to set the backup options you want before executing an action. You can easily delete backup files that are no longer needed in the undo history.

The undo history lists all actions that modified files since you last cleaned the undo history. The most recent actions are listed at the top. Since the same file may have been modified by more than one action, you should always undo actions from top to bottom, when you want to undo multiple actions.

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. If you run more than one PowerGREP instance at the same time, the undo history is automatically synchronized between all instances.

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.

18. 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

Copies the action definition and file selection used to execute the action you selected in the undo history to the Action and File Selector, respectively. You can then use the Preview, Execute or Quick Execute command to execute the same action 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”.

19. Change PowerGREP’s Appearance

PowerGREP's interface is completely modular. The application consists of seven panes. You can activate each of the panes through the View menu, whether you closed the pane or not. You can freely arrange all seven panes to best suit the way you like to work with PowerGREP.

The default layout is optimized for a computer with a single monitor and average screen resolution. If your computer has a single large resolution monitor, you can make use of the additional space by docking side-by-side some of the panes that are tabbed by default. If your computer has more than one monitor, take advantage of both monitors by making one or more panes float. Then drag the floating panes off to the second monitor.

How to Rearrange PowerGREP’s Panes

To move a pane, use the mouse to drag and drop its caption bar (for a pane docked to the side, or a floating pane) or its tab (for a tabbed pane). While you drag the pane, a thick gray rectangle moves around while you move the mouse. This rectangle indicates where the pane will move to should you release the mouse button.

To dock a pane to the side, move the mouse pointer near the edge. You can dock panes to the side of the entire PowerGREP window. You can also dock panes to the side of other panes, and even to the side of a pane that is inside a tab. While dragging a pane, moving the mouse pointer a few pixels closer to or away from the edge of PowerGREP's window will make the difference between docking the pane to the side of PowerGREP itself, or to the side of the pane that touches that edge of PowerGREP's window.

To arrange two panes inside a tabbed container, drag one pane and move the mouse pointer to the center of the other pane. The gray rectangle's shape will change slightly, showing an extension at the top that looks like an invisible tab.

To make a pane float freely, drag it away from PowerGREP or simply double-click its caption or tab. Floating a pane is very useful if your computer has more than one monitor. Move the floating pane to your second monitor to take full advantage of your multi-monitor system. If you drag a second pane onto the floating pane, you can dock both panes together in a single floating container. This way you can conveniently display several panes 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 pane where you define the action that PowerGREP will execute. In the default view, you can access the Action pane by clicking on the Action tab near the top of the PowerGREP window.

View Library

Show the Library pane where you can store PowerGREP actions for later reuse. In the default view, you can access the Library pane by clicking on the Library tab near the top of the PowerGREP window.

View Results

Show the Results pane where PowerGREP displays detailed results after executing an action. In the default view, you can access the Results pane 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 pane by clicking on the Editor tab near the top of the PowerGREP window.

View Undo History

Show or hide 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.

Large Toolbar Icons

If you have a high resolution monitor, the icons on PowerGREP's various toolbars may be too small to discern properly. Select Large Toolbar Icons in the View menu to make them 50% larger. Select the same command to restore the toolbar icons to their default size.

Office 2003 Display Style

If you find PowerGREP's looks a bit bland, select Office 2003 Display Style from the View menu to make PowerGREP mimic the looks of Microsoft Office 2003. This will make PowerGREP rather colorful. Select the item again to restore the default looks. On Windows XP, the default looks will use the Windows XP theme you selected in your computer's display settings.

Restore Default Layout

Use the Restore Default Layout item in the View menu to quickly reset the PowerGREP window to its default layout, with the File Selector docked to the left, the Assistant docked to the bottom, and the other panes arranged in tabs.

This layout keeps each pane sufficiently large to be workable on a computer with a single medium resolution monitor.

Side by Side Layout

The side by side layout docks the File Selector, Assistant, Action, Results and Editor panes side by side. The Library and Undo History panes are tabbed with the Action pane.

If you have a very large resolution monitor, use this layout to work more comfortably by keeping the most commonly used panes permanently visible.

Widescreen Layout

The widescreen layout docks all panes side by side in three columns. The File Selector and Assistant share the leftmost column. The Action and Editor panes share the middle column. The Results, Library and Undo History share the rightmost column.

This layout makes it possible to keep the panes that you'll usually use together visible at the same time. Where the side by side layout is only useful on a high resolution screen, the vertical layout is perfectly usable on a medium resolution monitor. It is particularly useful if you have a widescreen monitor or laptop. The columns optimally take advantage of the extra screen width.

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 and Undo History pane in tabs, with the File Selector and Assistant docked to the left and the bottom. The Results and Editor panes 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. It arranges the Action, Library, File Selector and Assistant panes side by side. The Results and Editor panes 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).

20. File Selection Preferences

On the File Selection tab in the Preferences screen, you can configure PowerGREP's File Selector and the way PowerGREP handles files, folders and the network.

Remember File Selection and Action between PowerGREP Sessions

Turn on "remember search terms" 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 file selection and action files”. 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 search terms”, PowerGREP will reload the modified action. If you select “remember files”, PowerGREP will reload the original action file.

If you select “do not remember search terms”, PowerGREP will not automatically remember the file selection and action definition. New instances will start out with a blank file selection and action definition. Choose this option if you usually don't continue working with the last set of files or action. Then you don't have to manually clear the file selection and action definition each time.

File Selector

Turn on "search through hidden filesand folders" to make hidden files and folders visible in the File Selector, and to allow PowerGREP to search through them.

Turn on "automatically get a list of network servers and shares" if you want PowerGREP to show all network servers it can find when you expand the Network node in the File Selector. You may want to turn off this option if your computer is connected to a very large network. A long list of servers clutters the File Selector.

When you turn off the option to automatically scan the network, you can still access the network by directly typing in a UNC path in the Path field in the File Selector. E.g. to access the network share “share” on the server “server”, type \\server\share. That share will then appear under the Network node in the File Selector until you close PowerGREP.

When automatically scanning the network, PowerGREP can either search the whole network, or only the servers that are on the same Windows workgroup or domain as your computer. If you usually only access computers in your own workgroup or domain, you should turn on this option. You can still access servers outside your computer's domain by typing in a UNC path such as \\server\share.

Files Excluded from All Actions

In this section you can enter a semicolon-delimited list of file masks or regular expressions, just like you do in the File Selector. All files of which the names match one of these file masks will be completely invisible to PowerGREP. They will not appear in the File Selector, and will never be searched through. The purpose of making files invisible this way is to reduce clutter in the File Selector.

By default, the permanent exclusion masks are set to `*.bak;*.~*;Backup*;Working copy*` which match all backup files created by PowerGREP, as well as temporary working copies of files created by EditPad Pro. If you'd ever need to search through backup files or working copies, you will need to adjust the permanent exclusion file masks first.

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.

21. Action & Results Preferences

On the Action & Results tab in the Preferences screen, you can configure some aspects of the way PowerGREP executes actions and displays their results.

Action Execution Options

PowerGREP recognizes a number of path placeholders such as in the search terms. These 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. If the placeholders conflict with text you're searching for, you can disable them here. PowerGREP will then treat the placeholders as any other literal text.

When PowerGREP creates or modifies a file, PowerGREP will set the last modification date of the new or modified file to the current date and time. If you turn on the option “give target files the same time stamp as the source file”, each target file will be given the same last modification date as the source file it is based on. The source file is the file that was searched through by PowerGREP.

The option “delete files to the Windows Recycle Bin” controls how files are deleted when you set the target type of a “find 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.

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 pane 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.

22. Text Encoding Preferences

Computers deal with numbers, not with characters. When you save a text file, each character is mapped to a number, and the numbers are stored on disk. When you open a text file, the numbers are read and mapped back to characters. When saving a file in one application, and opening the that file in another application, both applications need to use the same character mappings.

Traditional character mappings or code pages use only 8 bits per character. This means that only 256 characters can be represented in any text file. As a result, different character mappings are used for different language and scripts. Since different computer manufacturers had different ideas about how to create character mappings, there's a wide variety of legacy character mappings. PowerGREP supports a wide range of these.

In addition to conversion problems, the main problem with using traditional character mappings is that it is impossible to create text files written in multiple languages using multiple scripts. You can't mix Chinese, Russian and French in a text file, unless you use Unicode. Unicode is a standard that aims to encompass all traditional character mappings, and all scripts used by current and historical human languages.

What This Means to You

If you only search through files created on your own Windows computer, or on other Windows computers using the same regional settings, PowerGREP's default settings will suit you perfectly. All your files will all use the same traditional Windows code page and/or Unicode.

In the section “default settings for files not configured below”, select the Windows code page that matches the language you work with. All Windows code pages are an extension of US ASCII, which supports the English alphabet. Code page 1252 is the default for the Americas and Western Europe.

Text files normally should not contain NULL characters. Binary files usually do contain NULL bytes. Turn on "treat files containing NULL characters as binary files" to be able to exclude binary files in the File Selector with the option Search through Binary Files. When you do search through binary files, PowerGREP will display the results in hexadecimal. The file editor edits binary files in hexadecimal mode.

If PowerGREP treats certain text files as binary files, that is because those text files contain spurious NULL characters. You can turn off “treat files containing NULL characters as binary files” to force PowerGREP to treat all files as text files.

On the Windows platform, Unicode files should start with a byte order marker. The byte order marker is a special code that indicates the Unicode encoding (UTF-8, UTF-16 or UTF-32) used by the file. PowerGREP will always detect the byte order marker, and treat the file with the corresponding Unicode encoding. However, 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. You should turn on the option “detect UTF-16 files without a byte order marker”, to prevent UTF-16 files from being treated as binary files.

Text Encoding Definitions

Creating additional text encoding definitions is only necessary when you work with files encoded with different traditional character mappings. This will be the case with files created on different Windows computers using different regional or language settings. It will also be the case with text files created on computers using operating systems other than Windows. Linux systems usually use UTF-8 without a byte order marker (meaning PowerGREP cannot auto-detect the UTF-8 format), or one of the ISO-8859 code pages. Old files created with MS-DOS will use one of the DOS code pages. Files created on IBM mainframes are likely to use one of the EBCDIC encodings.

Click the New button to create a new text encoding definition. Click the Delete button to remove the selected definition. Use the Move Up and Move Down buttons to change the order. PowerGREP looks through the definitions from top to bottom when looking up the encoding to use for a particular file. If the file matches more than one text encoding definition, the topmost definition is used.

Assign each definition a meaningful label. This label is only used in the text encoding preferences. It makes it easy to look up the encoding later in the list.

Select the character mapping to use for files of this type (i.e. files matching the definition's file mask) from the Encoding drop-down list. This list has one extra mapping not available for the default encoding. At the top of the list, you'll find the "binary file" encoding. This is not really an encoding, but tells PowerGREP to treat all files of this type as 8-bit binary files. These files can be excluded in the File Selector with the option Search through Binary Files. When you do search through binary files, PowerGREP will display the results in hexadecimal. The file editor edits binary files in hexadecimal mode.

The file mask is a semicolon-delimited list of file masks, just like the include files masks in the File Selector. If a file's name matches one of the masks in the list, the text encoding definition is used for that file.

If some files of this type are text files, and others are binary files, you can turn on “treat files containing NULL characters as binary files” to make PowerGREP auto-detect binary files. For most text encoding definitions though, you will leave this option off.

If some files of this type are UTF-16 Unicode files without a byte order marker, while others use another encoding, select the other encoding from the Encoding list, and turn on “detect UTF-16 Unicode files without a byte order marker”. If you're not sure, turn on the option.

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.

23. Appearance Preferences

On the Appearance tab in the Preferences screen, you can configure PowerGREP's appearance to your own tastes and eyesight.

Search Term Editors

The editors for search terms are all edit boxes on the Action pane, the file masks boxes on the File Selector, and the description and details boxes on the Library pane.

Turn on "visualize spaces and tabs" if you want spaces and tabs to be visualized. Spaces will be indicated by a vertically centered dot. Tabs are indicated by a » character. Since all whitespace is significant in search terms, it's a good idea to leave this option on. If you enter whitespace before or after a search term, PowerGREP will look for the term including the whitespace.

Click the Choose Font button to select the font to use for these edit boxes.

You can configure the colors on the Regex Colors tab.

Results and File Editors

Configure the edit boxes showing the results, and PowerGREP's built-in file editor.

Turn on "visualize spaces and tabs" if you want spaces and tabs to be visualized. Spaces will be indicated by a vertically centered dot. Tabs are indicated by a » character. This option is useful when working with files where extraneous whitespace can lead to problems, or where the difference between tabs and spaces matters.

Click the Choose Font button to select the font to use for the results and the file editor. Only fixed pitch fonts such as Courier New are available. These fonts keep columns lined up properly.

You can configure the colors for the results on the Results Colors tab, and those for the file editor on the Syntax Colors tab.

Text Cursor

The text cursor is the typical blinking vertical bar that indicates the position in the document where the text you type will be inserted. While the standard Windows text cursor is often barely visible, PowerGREP's cursor is fully adjustable to your own taste and eyesight.

The shape of the text cursor can be a vertical bar displayed to the left of the character the cursor currently points to, or a rectangle that sits on top of the character. You can choose to permanently use either mode, or to let the shape depend on insert/overwrite mode. You can toggle between insert and overwrite mode by pressing the Insert key on the keyboard. In insert mode, the text cursor will be a vertical bar, in overwrite mode it will be a rectangle.

You can also choose if and how the cursor blinks. If you select “do not blink”, then the cursor will be a solid bar using the “regular” color. “Blinking on-off” is the standard blinking style. The text cursor will alternate several times per second between being visible in the “regular” color, and being invisible. The final style, “alternating colors”, causes the text cursor to be permanently visible, but alternate between the “regular” and “alternate” colors several times per second.

The “regular” and “alternate” colors are used for the normal text cursor, depending on the blinking style you chose. The “dragging” color is used for the cursor that indicates the dropping position while you drag and drop text with the mouse. This cursor is always a steady vertical bar.

24. 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 on the Editors tab 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 pane.

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%.

PowerGREP cannot pass line or column numbers on the command line, since PowerGREP allows arbitrary file sectioning rather than always searching line by line.

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.

Finally, 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.

25. 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

On the Regex Colors tab, you can configure the colors for search term edit boxes. These are all edit boxes on the Action pane, the file masks boxes on the File Selector, and the description and details boxes on the Library pane.

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

On the Results Colors tab, 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

On the Syntax Colors tab, you can configure the colors used by 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.

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.

26. 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 path placeholders such as %FILE%" 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 a .zip archive, PowerGREP will combine the folder the zip file is in with the folder and file name of the file stored inside the .zip archive. When searching through a file zipped.txt in an archive c:\data\archive.zip, then %FILE% is set to c:\data\zipped.txt. If the archive does contain a folder structure, and PowerGREP is searching through zipfolder\zipped.txt, then %FILE% is set to c:\data\zipfolder\zipped.txt.

When using regular expressions, path placeholders are expanded after backreferences are expanded. So if the replacement text is “`\2`”, and the second backreference holds “`%FILE%`”, then the actual replacement will be the full path and filename to the file searched through. Since “`\%`” is not a valid backreference, you also do not need to escape backslashes in the replacement text when combining placeholders. “`%FOLDER<1%\%FILENAME%`” is a valid replacement text. The backslash will not interfere with the percentage sign.

For smarter handling of slashes in paths, you can rewrite that last example as the combined path placeholder %FOLDER<1\FILENAME%.

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
%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.

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\

27. Supported File Formats

PowerGREP is primarily designed to work with plain text files. Plain text files include plain text documents, HTML and XML files, software source code, comma-delimited and other data files, configuration files, etc. When searching through tagged formats such as HTML, PowerGREP does not make any difference between the HTML tags and the text that you would see when viewing the HTML file in a browser. It's all plain text.

Many applications store their data in binary files. These files often use file formats that are proprietary to the developer of the application. Such files generally cannot be processed by software other than the software that created them.

PowerGREP can decode a small number of proprietary file formats. Prior to searching, PowerGREP will create a textual representation of the file's contents. That textual representation is then searched through. Making replacements in files stored in proprietary formats is not possible. PowerGREP cannot convert the textual representation back into the original format.

PowerGREP can currently decode Microsoft Word documents (*.doc and *.dot files), Excel spreadsheets (*.xls files), and PDF files (*.pdf). PowerGREP also has partial support for Quattro Pro spreadsheets and Lotus 1-2-3 spreadsheets. “Partial support” means that not all versions of these file formats can be decoded by PowerGREP.

Unsupported Formats

You can search through the raw, non-decoded contents of files that PowerGREP cannot decode by turning on the option “search through binary files”. The search match will be listed in the results in both textual and hexadecimal representation. If you double-click on the search match, the file editor will open the file in hexadecimal mode. The view is very similar to that of a hex editor. Depending on the kind of file you are dealing with, you can still get meaningful search results.

To search for a sequence of bytes, rather than a text string, select the binary data search type. You can then enter the bytes into the search box as you would enter them into a hex editor.

28. 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 results file, one library file and one undo history file. The file will be loaded into the corresponding pane. In addition, you can specify one file of any other file. That file will be opened in the built-in file editor.

File selections are saved in file selection files, action files and results files. If you specify two or all three of these files on the command line, PowerGREP will use the file selection from the file selection file, or from the action file if you didn't specify a file selection file. Only when you don't specify either a file selection file or an action file, will PowerGREP read the file selection from the results file.

In similar vein, action definitions are saved in both action files and results files. If you specify both an action file and results file on the command line, PowerGREP will read the action definition from the action file.

If you specify options that affect the file selection or action, PowerGREP will load the file, and then use the options to modify the settings. In this situation, PowerGREP's caption bar will *not* indicate the name of the file selection file or action file.

File Selection and Action 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. `/find` sets the action type to “find files”.

2. `/search` sets the action type to “display search matches”.

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

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

5. `/searchtext <text>` sets the search term of the main part of the action to “text”.

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

7. `/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.

8. `/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.

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

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

11. `/delimitprefix <delimiter>` sets the search prefix label delimiter to “delimiter”. Also sets the search type to a delimited list.

12. `/delimitsearch <delimiter>` sets the search item delimiter to “delimiter”. Also sets the search type to a delimited list.

13. `/delimitreplace <delimiter>` sets the search pair delimiter to “delimiter”. Also sets the search type to a delimited list.

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

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

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

17. `/optcase <0|1>` Sets the option “case sensitive”.

18. `/optadaptive <0|1>` Sets the option “adaptive case”.

19. `/optinvert <0|1>` Sets the option “invert results”. Only has an effect when the action type is “find files”, or when you load an action definition that sections files.

20. `/target <0|1|2|3|4|5|6> <0|1|2|3|4> <location>` Sets the target options on the Replace and Sequence pages. Has no effect on the other pages. This parameter must be followed by one, two or three values. The number of values depends on the actual values you provide, as explained below.

The first value indicates how files are copied. When copying files, the original file will remain untouched.
0 = Do not copy files but change the file searched through. Do not specify any destination type or location.

(search-and-replace; collect data)
1 = Copy files in which replacements have been made. (search-and-replace; collect data)
2 = Copy all files searched through. (search-and-replace; collect data)
3 = Do not save results. (find files; collect data)
4 = Save results to single file. (find files; collect data)
5 = Move matching files (find files)
6 = Delete matching files (find files)

The second value indicates the destination type for copied files:
0 = Place all target files into a single folder.
1 = Place target files into a folder tree.
2 = Place target files into a .zip archive.
3 = Place target files into a numbered .zip archive.
4 = Use path placeholders

The third value indicates the actual location, either a folder, a .zip file or a path using path placeholders.

21. `/backup <0|1|2|3|4|5|6|7> <0|1|2|3|4> <location>` Sets the backup options on the Replace and Sequence pages. Has no effect on the other pages. This parameter must be followed by one, two or three values. The number of values depends on the actual values you provide, as explained below.

The first value indicates the type of backup to create:
0 = Do not create backup files. Do not specify any destination type or location.
1 = Single backup appending .bak extension
2 = Single backup with .~* extension
3 = Multi backup appending .bak, .bak2, ... extensions
4 = Multi backup prepending "Backup X of ..."
5 = Backup with same file name as original file (destination cannot be the same folder)
6 = Use path placeholders
7 = Hidden __history folder. Do not specify any destination type or location.

The second value indicates the destination type of the backup files:
0 = Same folder as original. Do not specify a location.
1 = Place all backup files into a single folder.
2 = Place backup files into a folder tree.
3 = Place backup files into a .zip archive.
4 = Place backup files into a numbered .zip archive.

The third value indicates the actual location, either a folder, a .zip file or a path using path placeholders.

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

23. `/optarchives <0|1>` Sets the option “search through archives”.

24. `/folder <folders>` The value must be a semicolon-delimited list of folders. Includes those folders, but not their subfolders, in the next action.

25. `/folderrecurse <folders>` The value must be a semicolon-delimited list of folders. Includes those folders and their subfolders in the next action.

26. `/masks <include> <exclude> <0|1>` Sets the file masks. The first value is the inclusion mask. The second value is the exclusion mask. The third value indicates if the masks are traditional file masks or regular expressions.

27. `/save <filename>` If the extension is .pgr, the results created by the /preview, /execute or /quick parameter will be saved into a PowerGREP 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.

When saving into a text file, you can specify an additional value after the file name to determine the encoding to be used for the text file. Valid values are `utf8` for UTF-8, `utf16le` for Unicode, or `utf16be` for big-endian Unicode. If you do not specify an encoding, the default ANSI code page will be used.

28. `/preview` Preview the action.

29. `/execute` Execute the action.

30. `/quick` Quick execute the action.

31. `/quit` This parameter causes PowerGREP to terminate after successfully executing an action.

29. 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 pane, 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/powergrep3.xsd. When creating file selection and action files by yourself, make sure to validate them against the schema. PowerGREP does not validate files against the schema, and makes little effort to display helpful error messages.

In order to avoid inconsistencies, there is no separate documentation for PowerGREP's file formats. The XML schema is annotated, and serves as both human-readable documentation and machine-readable specification.

The XML schema is laid out in a bottom-up fashion. The root element, which is always “powergrep”, is defined as the last element in the schema. If you collapse all nodes under xsd:schema you will get an overview of all the types the schema defines.

Part 4

Regular Expression Tutorial

1. Regular Expression Tutorial

In this tutorial, I will teach you all you need to know to be able to craft powerful time-saving regular expressions. I will start with the most basic concepts, so that you can follow this tutorial even if you know nothing at all about regular expressions yet.

But I will not stop there. I will also explain how a regular expression engine works on the inside, and alert you at the consequences. This will help you to understand quickly why a particular regex does not do what you initially expected. It will save you lots of guesswork and head scratching when you need to write more complex regexes.

What Regular Expressions Are Exactly - Terminology

Basically, a regular expression is a pattern describing a certain amount of text. Their name comes from the mathematical theory on which they are based. But we will not dig into that. Since most people including myself are lazy to type, you will usually find the name abbreviated to regex or regexp. I prefer regex, because it is easy to pronounce the plural "regexes". In this book, regular expressions are printed guillemots: «`regex`». They clearly separate the pattern from the surrounding text and punctuation.

This first example is actually a perfectly valid regex. It is the most basic pattern, simply matching the literal text „`regex`”. A "match" is the piece of text, or sequence of bytes or characters that pattern was found to correspond to by the regex processing software. Matches are indicated by double quotation marks, with the left one at the base of the line.

«`\b[A-Z0-9._%-]+@[A-Z0-9._%-]+\.[A-Z]{2,4}\b`» is a more complex pattern. It describes a series of letters, digits, dots, percentage signs and underscores, followed by an at sign, followed by another series of letters, digits, dots, percentage signs and underscores, finally followed by a single dot and between two and four letters. In other words: this pattern describes an email address.

With the above regular expression pattern, you can search through a text file to find email addresses, or verify if a given string looks like an email address. In this tutorial, I will use the term "string" to indicate the text that I am applying the regular expression to. I will indicate strings using regular double quotes. The term “string” or “character string” is used by programmers to indicate a sequence of characters. In practice, you can use regular expressions with whatever data you can access using the application or programming language you are working with.

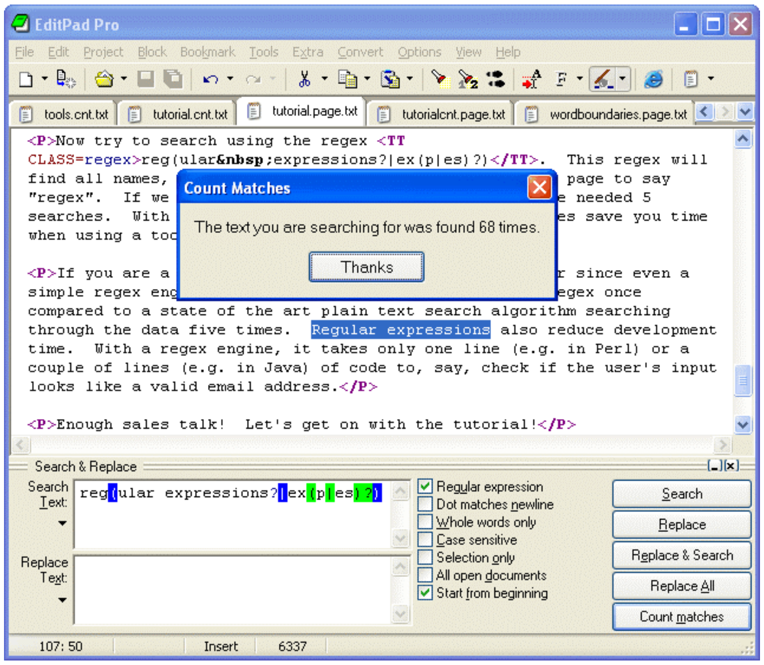
Different Regular Expression Engines

A regular expression “engine” is a piece of software that can process regular expressions, trying to match the pattern to the given string. Usually, the engine is part of a larger application and you do not access the engine directly. Rather, the application will invoke it for you when needed, making sure the right regular expression is applied to the right file or data.

As usual in the software world, different regular expression engines are not fully compatible with each other. It is not possible to describe every kind of engine and regular expression syntax (or “flavor”) in this tutorial. I will focus on the regex flavor used by Perl 5, for the simple reason that this regex flavor is the most popular one, and deservedly so. Many more recent regex engines are very similar, but not identical, to the one of Perl 5. Examples are the open source PCRE engine (used in many tools and languages like PHP), the .NET regular expression library, and the regular expression package included with version 1.4 and later of the Java JDK. I will point out to you whenever differences in regex flavors are important, and which features are specific to the Perl-derivatives mentioned above.

Give Regexes a First Try

You can easily try the following yourself in a text editor that supports regular expressions, such as EditPad Pro. If you do not have such an editor, you can download the free evaluation version of EditPad Pro to try this out. EditPad Pro's regex engine is fully functional in the demo version. As a quick test, copy and paste the text of this page into EditPad Pro. Then select Edit|Search and Replace from the menu. In the search pane that appears near the bottom, type in «regex» in the box labeled “Search Text”. Mark the “Regular expression” checkbox, unmark “All open documents” and mark “Start from beginning”. Then click the Search button and see how EditPad Pro's regex engine finds the first match. When “Start from beginning” is checked, EditPad Pro uses the entire file as the string to try to match the regex to.



```
<P>Now try to search using the regex <TT
CLASS=regex>reg(ular&nbsp;expressions?|ex(p|es)?)</TT>.  This regex will
find all names,                                          page to say
"regex".  If we                                          e needed 5
searches.  With                                          es save you time
when using a toc

<P>If you are a                                          r since even a
simple regex eng                                         egex once
compared to a state of the art plain text search algorithm searching
through the data five times.  Regular expressions also reduce development
time.  With a regex engine, it takes only one line (e.g. in Perl) or a
couple of lines (e.g. in Java) of code to, say, check if the user's input
looks like a valid email address.</P>

<P>Enough sales talk!  Let's get on with the tutorial!</P>
```

Count Matches

The text you are searching for was found 68 times.

Thanks

Search & Replace

Search Text: `reg(ular expressions?|ex(p|es)?)`

Replace Text:

[✓] Regular expression
[ ] Dot matches newline
[ ] Whole words only
[ ] Case sensitive
[ ] Selection only
[ ] All open documents
[✓] Start from beginning

107: 50 Insert 6337

When the regex has been matched, EditPad Pro will automatically turn off “Start from beginning”. When you click the Search button again, the remainder of the file, after the highlighted match, is used as the string. When the regex can no longer match the remaining text, you will be notified, and “Start from beginning” is automatically turned on again.

Now try to search using the regex «reg(ular expressions?|ex(p|es)?)» . This regex will find all names, singular and plural, I have used on this page to say “regex”. If we only had plain text search, we would have needed 5 searches. With regexes, we need just one search. Regexes save you time when using a tool like EditPad Pro.

If you are a programmer, your software will run faster since even a simple regex engine applying the above regex once will outperform a state of the art plain text search algorithm searching through the data five times. Regular expressions also reduce development time. With a regex engine, it takes only one line (e.g. in Perl, PHP, Java or .NET) or a couple of lines (e.g. in C using PCRE) of code to, say, check if the user's input looks like a valid email address.

2. Literal Characters

The most basic regular expression consists of a single literal character, e.g.: «a». It will match the first occurrence of that character in the string. If the string is "Jack is a boy", it will match the „a" after the "J". The fact that this "a" is in the middle of the word does not matter to the regex engine. If it matters to you, you will need to tell that to the regex engine by using word boundaries. We will get to that later.

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

Similarly, the regex «cat» will match „cat" in "About cats and dogs". This regular expression consists of a series of three literal characters. This is like saying to the regex engine: find a «c», immediately followed by an «a», immediately followed by a «t».

Note that regex engines are case sensitive by default. «cat» does not match "Cat", unless you tell the regex engine to ignore differences in case.

Special Characters

Because we want to do more than simply search for literal pieces of text, we need to reserve certain characters for special use. In the regex flavors discussed in this tutorial, there are 11 characters with special meanings: the opening square bracket «[», the backslash «\», the caret «^», the dollar sign «$», the period or dot «.», the vertical bar or pipe symbol «|», the question mark «?», the asterisk or star «*», the plus sign «+», the opening round bracket «(» and the closing round bracket «)». These special characters are often called "metacharacters".

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

Note that «1+1=2», with the backslash omitted, is a valid regex. So you will not get an error message. But it will not match "1+1=2". It would match „111=2" in "123+111=234", due to the special meaning of the plus character.

If you forget to escape a special character where its use is not allowed, such as in «+1», then you will get an error message.

All other characters should not be escaped with a backslash. That is because the backslash is also a special character. The backslash in combination with a literal character can create a regex token with a special meaning. E.g. «\d» will match a single digit from 0 to 9.

Special Characters and Programming Languages

If you are a programmer, you may be surprised that characters like the single quote and double quote are not special characters. That is correct. When using a regular expression or grep tool like PowerGREP or the search function of a text editor like EditPad Pro, you should not escape or repeat the quote characters like you do in a programming language.

In your source code, you have to keep in mind which characters get special treatment inside strings by your programming language. That is because those characters will be processed by the compiler, before the regex library sees the string. So the regex «`1\+1=2`» must be written as `"1\\+1=2"` in C++ code. The C++ compiler will turn the escaped backslash in the source code into a single backslash in the string that is passed on to the regex library. To match „`c:\temp`", you need to use the regex «`c:\\temp`». As a string in C++ source code, this regex becomes `"c:\\\\temp"`. Four backslashes to match a single one indeed.

See the tools and languages section in this book for more information on how to use regular expressions in various programming languages.

Non-Printable Characters

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

You can include any character in your regular expression if you know its hexadecimal ASCII or ANSI code for the character set that you are working with. In the Latin-1 character set, the copyright symbol is character 0xA9. So to search for the copyright symbol, you can use «`\xA9`». Another way to search for a tab is to use «`\x09`». Note that the leading zero is required.

If your regular expression engine supports Unicode, use «`\uFFFF`» rather than «`\xFF`» to insert a Unicode character. The euro currency sign occupies code point 0x20A0. If you cannot type it on your keyboard, you can insert it into a regular expression with «`\u20A0`».

3. First Look at How a Regex Engine Works Internally

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

There are two kinds of regular expression engines: text-directed engines, and regex-directed engines. Jeffrey Friedl calls them DFA and NFA engines, respectively. All the regex flavors treated in this tutorial are based on regex-directed engines. This is because certain very useful features, such as lazy quantifiers and backreferences, can only be implemented in regex-directed engines. No surprise that this kind of engine is more popular.

Notable tools that use text-directed engines are awk, egrep, flex, lex, MySQL and Procmail. For awk and egrep, there are a few versions of these tools that use a regex-directed engine.

You can easily find out whether the regex flavor you intend to use has a text-directed or regex-directed engine. If backreferences and/or lazy quantifiers are available, you can be certain the engine is regex-directed. You can do the test by applying the regex «`regex|regex not`» to the string “`regex not`”. If the resulting match is only „`regex`”, the engine is regex-directed. If the result is „`regex not`”, then it is text-directed. The reason behind this is that the regex-directed engine is “eager”.

In this tutorial, after introducing a new regex token, I will explain step by step how the regex engine actually processes that token. This inside look may seem a bit long-winded at certain times. But understanding how the regex engine works will enable you to use its full power and help you avoid common mistakes.

The Regex-Directed Engine Always Returns the Leftmost Match

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

When applying «`cat`» to “`He captured a catfish for his cat.`”, the engine will try to match the first token in the regex «`c`» to the first character in the match “`H`”. This fails. There are no other possible permutations of this regex, because it merely consists of a sequence of literal characters. So the regex engine tries to match the «`c`» with the “`e`”. This fails too, as does matching the «`c`» with the space. Arriving at the 4th character in the match, «`c`» matches „`c`”. The engine will then try to match the second token «`a`» to the 5th character, „`a`”. This succeeds too. But then, «`t`» fails to match “`p`”. At that point, the engine knows the regex cannot be matched starting at the 4th character in the match. So it will continue with the 5th: “`a`”. Again, «`c`» fails to match here and the engine carries on. At the 15th character in the match, «`c`» again matches „`c`”. The engine then proceeds to attempt to match the remainder of the regex at character 15 and finds that «`a`» matches „`a`” and «`t`» matches „`t`”.

The entire regular expression could be matched starting at character 15. The engine is "eager" to report a match. It will therefore report the first three letters of catfish as a valid match. The engine never proceeds beyond this point to see if there are any “better” matches. The first match is considered good enough.

In this first example of the engine's internals, our regex engine simply appears to work like a regular text search routine. A text-directed engine would have returned the same result too. However, it is important that you can follow the steps the engine takes in your mind. In following examples, the way the engine works will have a profound impact on the matches it will find. Some of the results may be surprising. But they are always logical and predetermined, once you know how the engine works.

4. Character Classes or Character Sets

With a "character class", also called “character set”, you can tell the regex engine to match only one out of several characters. Simply place the characters you want to match between square brackets. If you want to match an a or an e, use «`[ae]`». You could use this in «`gr[ae]y`» to match either „`gray`” or „`grey`”. Very useful if you do not know whether the document you are searching through is written in American or British English.

A character class matches only a single character. «`gr[ae]y`» will not match “`graay`”, “`graey`” or any such thing. The order of the characters inside a character class does not matter. The results are identical.

You can use a hyphen inside a character class to specify a range of characters. «`[0-9]`» matches a *single* digit between 0 and 9. You can use more than one range. «`[0-9a-fA-F]`» matches a single hexadecimal digit, case insensitively. You can combine ranges and single characters. «`[0-9a-fxA-FX]`» matches a hexadecimal digit or the letter X. Again, the order of the characters and the ranges does not matter.

Useful Applications

Find a word, even if it is misspelled, such as «`sep[ae]r[ae]te`» or «`li[cs]en[cs]e`».

Find an identifier in a programming language with «`[A-Za-z_][A-Za-z_0-9]*`».

Find a C-style hexadecimal number with «`0[xX][A-Fa-f0-9]+`».

Negated Character Classes

Typing a caret after the opening square bracket will negate the character class. The result is that the character class will match any character that is *not* in the character class. Unlike the dot, negated character classes also match (invisible) line break characters.

It is important to remember that a negated character class still must match a character. «`q[^u]`» does *not* mean: “a q not followed by a u”. It means: “a q followed by a character that is not a u”. It will not match the q in the string “`Iraq`”. It will match the q and the space after the q in “`Iraq is a country`”. Indeed: the space will be part of the overall match, because it is the “character that is not a u” that is matched by the negated character class in the above regexp. If you want the regex to match the q, and only the q, in both strings, you need to use negative lookahead: «`q(?!u)`». But we will get to that later.

Metacharacters Inside Character Classes

Note that the only special characters or metacharacters inside a character class are the closing bracket (]), the backslash (\), the caret (^) and the hyphen (-). The usual metacharacters are normal characters inside a character class, and do not need to be escaped by a backslash. To search for a star or plus, use «`[+*]`». Your regex will work fine if you escape the regular metacharacters inside a character class, but doing so significantly reduces readability.

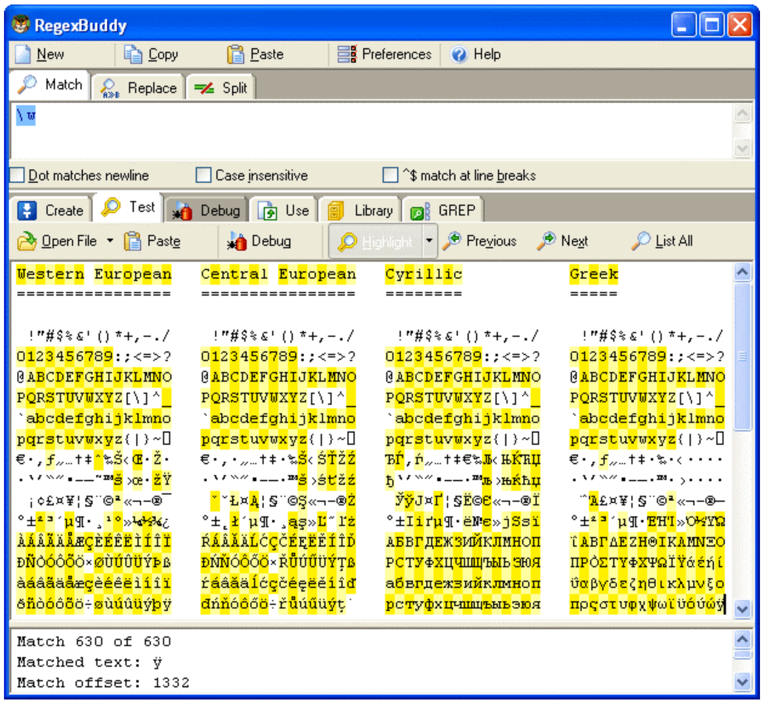
To include a backslash as a character without any special meaning inside a character class, you have to escape it with another backslash. «`[\\x]`» matches a backslash or an x. The closing bracket (]), the caret (^) and the hyphen (-) can be included by escaping them with a backslash, or by placing them in a position where they do not take on their special meaning. I recommend the latter method, since it improves readability. To include a caret, place it anywhere except right after the opening bracket. «`[x^]`» matches an x or a caret. You can put the closing bracket right after the opening bracket, or the negating caret. «`[]x]`» matches a closing bracket or an x. «`[^]x]`» matches any character that is not a closing bracket or an x. The hyphen can be included right after the opening bracket, or right before the closing bracket, or right after the negating caret. Both «`[-x]`» and «`[x-]`» match an x or a hyphen.

You can use non-printable characters in character classes just like you can use them outside of character classes. E.g. «`[$\u20A0]`» matches a dollar or euro sign, assuming your regex flavor supports Unicode.

Shorthand Character Classes

Since certain character classes are used often, a series of shorthand character classes are available. «`\d`» is short for «`[0-9]`».

«`\w`» stands for "word character". Exactly which characters it matches differs between regex flavors. In all flavors, it will include «`[A-Za-z]`». In most, the underscore and digits are also included. In some flavors, word characters from other languages may also match. The best way to find out is to do a couple of tests with the regex flavor you are using. In the screen shot, you can see the characters matched by «`\w`» in RegexBuddy using various scripts.



```
\w
```

Dot matches newline [ ]   Case insensitive [ ]   ^$ match at line breaks [ ]

```
Western European     Central European     Cyrillic             Greek
================     ================     ========             =====

 !"#$%&'()*+,-./      !"#$%&'()*+,-./      !"#$%&'()*+,-./      !"#$%&'()*+,-./
0123456789:;<=>?     0123456789:;<=>?     0123456789:;<=>?     0123456789:;<=>?
@ABCDEFGHIJKLMNO     @ABCDEFGHIJKLMNO     @ABCDEFGHIJKLMNO     @ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_     PQRSTUVWXYZ[\]^_     PQRSTUVWXYZ[\]^_     PQRSTUVWXYZ[\]^_
`abcdefghijklmno     `abcdefghijklmno     `abcdefghijklmno     `abcdefghijklmno
pqrstuvwxyz{|}~      pqrstuvwxyz{|}~      pqrstuvwxyz{|}~      pqrstuvwxyz{|}~
€·‚ƒ„…†‡ˆ‰Š‹Œ·Ž·     €·‚·„…†‡·‰Š‹ŚŤŽŹ     ЂЃ‚ѓ„…†‡€‰Љ‹ЊЌЋЏ     €·‚ƒ„…†‡·‰·‹····
·‘’“”•–—˜™š›œ·žŸ     ·‘’“”•–—·™š›śťžź     ђ‘’“”•–—·™љ›њќћџ      ·‘’“”•–—·™·›····
 ¡¢£¤¥¦§¨©ª«¬-®¯      ˇ˘Ł¤Ą¦§¨©Ş«¬-®Ż      ЎўЈ¤Ґ¦§Ё©Є«¬-®Ї      ΅Ά£¤¥¦§¨©ª«¬-®―
°±²³´µ¶·¸¹º»¼½¾¿     °±˛ł´µ¶·¸ąş»Ľ˝ľż     °±Ііґµ¶·ё№є»јЅѕї     °±²³΄µ¶·ΈΉΊ»Ό½ΎΏ
ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏ     ŔÁÂĂÄĹĆÇČÉĘËĚÍÎĎ     АБВГДЕЖЗИЙКЛМНОП     ΐΑΒΓΔΕΖΗΘΙΚΛΜΝΞΟ
ÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞß     ĐŃŇÓÔŐÖ×ŘŮÚŰÜÝŢß     РСТУФХЦЧШЩЪЫЬЭЮЯ     ΠΡ΢ΣΤΥΦΧΨΩΪΫάέήί
àáâãäåæçèéêëìíîï     ŕáâăäĺćçčéęëěíîď     абвгдежзийклмноп     ΰαβγδεζηθικλμνξο
ðñòóôõö÷øùúûüýþÿ     đńňóôőö÷řůúűüýţ˙     рстуфхцчшщъыьэюя     πρςστυφχψωϊϋόύώ
```

```
Match 630 of 630
Matched text: ÿ
Match offset: 1332
```

«`\s`» stands for "whitespace character". Again, which characters this actually includes, depends on the regex flavor. In all flavors discussed in this tutorial, it includes «`[ \t]`». That is: «`\s`» will match a space or a tab. In most flavors, it also includes a carriage return or a line feed as in «`[ \t\r\n]`». Some flavors include additional, rarely used non-printable characters such as vertical tab and form feed.

Shorthand character classes can be used both inside and outside the square brackets. «`\s\d`» matches a whitespace character followed by a digit. «`[\s\d]`» matches a single character that is either whitespace or a digit. When applied to "`1 + 2 = 3`", the former regex will match „ `2`" (space two), while the latter matches „`1`" (one). «`[\da-fA-F]`» matches a hexadecimal digit, and is equivalent to «`[0-9a-fA-F]`».

Negated Shorthand Character Classes

The above three shorthands also have negated versions. «`\D`» is the same as «`[^\d]`», «`\W`» is short for «`[^\w]`» and «`\S`» is the equivalent of «`[^\s]`».

Be careful when using the negated shorthands inside square brackets. «`[\D\S]`» is *not* the same as «`[^\d\s]`». The latter will match any character that is not a digit or whitespace. So it will match „`x`", but not "`8`". The former, however, will match any character that is either not a digit, or is not whitespace. Because a digit is not whitespace, and whitespace is not a digit, «`[\D\S]`» will match any character, digit, whitespace or otherwise.

Repeating Character Classes

If you repeat a character class by using the «`?`», «`*`» or «`+`» operators, you will repeat the entire character class, and not just the character that it matched. The regex «`[0-9]+`» can match „`837`" as well as „`222`".

If you want to repeat the matched character, rather than the class, you will need to use backreferences. «`([0-9])\1+`» will match „`222`" but not "`837`". When applied to the string "`833337`", it will match „`3333`" in the middle of this string. If you do not want that, you need to use lookahead and lookbehind.

But I digress. I did not yet explain how character classes work inside the regex engine. Let us take a look at that first.

Looking Inside The Regex Engine

As I already said: the order of the characters inside a character class does not matter. «`gr[ae]y`» will match „`grey`" in "`Is his hair grey or gray?`", because that is the *leftmost match*. We already saw how the engine applies a regex consisting only of literal characters. Below, I will explain how it applies a regex that has more than one permutation. That is: «`gr[ae]y`» can match both „`gray`" and „`grey`".

Nothing noteworthy happens for the first twelve characters in the string. The engine will fail to match «`g`» at every step, and continue with the next character in the string. When the engine arrives at the 13th character, „`g`" is matched. The engine will then try to match the remainder of the regex with the text. The next token in the regex is the literal «`r`», which matches the next character in the text. So the third token, «`[ae]`» is attempted at the next character in the text ("`e`"). The character class gives the engine two options: match «`a`» or match «`e`». It will first attempt to match «`a`», and fail.

But because we are using a regex-directed engine, it must continue trying to match all the other permutations of the regex pattern before deciding that the regex cannot be matched with the text starting at character 13. So it will continue with the other option, and find that «`e`» matches „`e`". The last regex token is «`y`», which can be matched with the following character as well. The engine has found a complete match with the text starting at character 13. It will return „`grey`" as the match result, and look no further. Again, the *leftmost match* was returned, even though we put the «`a`» first in the character class, and „`gray`" could have been matched in the string. But the engine simply did not get that far, because another equally valid match was found to the left of it.

5. 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 newlinecharacters. In all regex flavors discussed in this tutorial, the dot will *not* match a newline character by default. So by default, the dot is short for the negated character class «`[^\n]`» (UNIX regex flavors) or «`[^\r\n]`» (Windows regex flavors).

This exception exists mostly because of historic reasons. The first tools that used regular expressions were line-based. They would read a file line by line, and apply the regular expression separately to each line. The effect is that with these tools, the string could never contain newlines, so the dot could never match them.

Modern tools and languages can apply regular expressions to very large strings or even entire files. All regex flavors discussed here have an option to make the dot match all characters, including newlines. In RegexBuddy, EditPad Pro or PowerGREP, you simply tick the checkbox labeled “dot matches newline”.

In Perl, the mode where the dot also matches newlines is called "single-line mode". This is a bit unfortunate, because it is easy to mix up this term with “multi-line mode”. Multi-line mode only affects anchors, and single-line mode only affects the dot. You can activate single-line mode by adding an s after the regex code, like this: `m/^regex$/s;`.

Other languages and regex libraries have adopted Perl's terminology. When using the regex classes of the .NET framework, you activate this mode by specifying `RegexOptions.Singleline`, such as in `Regex.Match(“string”, “regex”, RegexOptions.Singleline)`.

In all programming languages and regex libraries I know, activating single-line mode has no effect other than making the dot match newlines. So if you expose this option to your users, please give it a clearer label like was done in RegexBuddy, EditPad Pro and PowerGREP.

Use The Dot Sparingly

The dot is a very powerful regex metacharacter. It allows you to be lazy. Put in a dot, and everything will match just fine when you test the regex on valid data. The problem is that the regex will also match in cases where it should not match. If you are new to regular expressions, some of these cases may not be so obvious at first.

I will illustrate this with a simple example. Let's say we want to match a date in mm/dd/yy format, but we want to leave the user the choice of date separators. The quick solution is «`\d\d.\d\d.\d\d`». Seems fine at first. It will match a date like „`02/12/03`” just fine. Trouble is: „`02512703`” is also considered a valid date by this regular expression. In this match, the first dot matched „`5`”, and the second matched „`7`”. Obviously not what we intended.

«`\d\d[- /.]\d\d[- /.]\d\d`» is a better solution. This regex allows a dash, space, dot and forward slash as date separators. Remember that the dot is not a metacharacter inside a character class, so we do not need to escape it with a backslash.

This regex is still far from perfect. It matches „`99/99/99`” as a valid date. «`[0-1]\d[- /.][0-3]\d[- /.]\d\d`» is a step ahead, though it will still match „`19/39/99`”. How perfect you want your regex to be depends on what you want to do with it. If you are validating user input, it has to be perfect. If you are parsing data files from a known source that generates its files in the same way every time, our last attempt is probably more than sufficient to parse the data without errors. You can find a better regex to match dates in the example section.

Use Negated Character Sets Instead of the Dot

I will explain this in depth when I present you the repeat operators star and plus, but the warning is important enough to mention it here as well. I will illustrate with an example.

Suppose you want to match a double-quoted string. Sounds easy. We can have any number of any character between the double quotes, so «`".*"`» seems to do the trick just fine. The dot matches any character, and the star allows the dot to be repeated any number of times, including zero. If you test this regex on “`Put a “string” between double quotes`”, it will match „`“string”`” just fine. Now go ahead and test it on “`Houston, we have a problem with “string one” and “string two”. Please respond.`”

Ouch. The regex matches „`“string one” and “string two”`”. Definitely not what we intended. The reason for this is that the star is *greedy*.

In the date-matching example, we improved our regex by replacing the dot with a character class. Here, we will do the same. Our original definition of a double-quoted string was faulty. We do not want any number of *any character* between the quotes. We want any number of characters that are not double quotes or newlines between the quotes. So the proper regex is «`"[^"\r\n]*"`».

6. Start of String and End of String Anchors

Thus far, I have explained literal characters and character classes. In both cases, putting one in a regex will cause the regex engine to try to match a single character.

Anchors are a different breed. They do not match any character at all. Instead, they match a position before, after or between characters. They can be used to "anchor" the regex match at a certain position. The caret «^» matches the position before the first character in the string. Applying «^a» to "abc" matches „a". «^b» will not match "abc" at all, because the «b» cannot be matched right after the start of the string, matched by «^». See below for the inside view of the regex engine.

Similarly, «$» matches right after the last character in the string. «c$» matches „c" in "abc", while «a$» does not match at all.

Useful Applications

When using regular expressions in a programming language to validate user input, using anchors is very important. If you use the code `if ($input =~ m/\d+/)` in a Perl script to see if the user entered an integer number, it will accept the input even if the user entered "qsdf4ghjk", because «\d+» matches the 4. The correct regex to use is «^\d+$». Because "start of string" must be matched before the match of «\d+», and "end of string" must be matched right after it, the entire string must consist of digits for «^\d+$» to be able to match.

It is easy for the user to accidentally type in a space. When Perl reads from a line from a text file, the line break will also be stored in the variable. So before validating input, it is good practice to trim leading and trailing whitespace. «^\s+» matches leading whitespace and «\s+$» matches trailing whitespace. In Perl, you could use `$input =~ s/^\s+|\s+$//g`. Handy use of alternation and /g allows us to do this in a single line of code.

Using ^ and $ as Start of Line and End of Line Anchors

If you have a string consisting of multiple lines, like "`first line\nsecond line`" (where \n indicates a line break), it is often desirable to work with lines, rather than the entire string. Therefore, all the regex engines discussed in this tutorial have the option to expand the meaning of both anchors. «^» can then match at the start of the string (before the "f" in the above string), as well as after each line break (between "\n" and "s"). Likewise, «$» will still match at the end of the string (after the last "e"), and also before every line break (between "e" and "\n").

In text editors like EditPad Pro or GNU Emacs, and regex tools like PowerGREP, the caret and dollar always match at the start and end of each line. This makes sense because those applications are designed to work with entire files, rather than short strings.

In all programming languages and libraries discussed in this book , except Ruby, you have to explicitly activate this extended functionality. It is traditionally called "multi-line mode". In Perl, you do this by adding an m after the regex code, like this: `m/^regex$/m;`. In .NET, the anchors match before and after newlines when you specify `RegexOptions.Multiline`, such as in `Regex.Match("string", "regex", RegexOptions.Multiline)`.

Permanent Start of String and End of String Anchors

«`\A`» only ever matches at the start of the string. Likewise, «`\Z`» only ever matches at the end of the string. These two tokens never match at line breaks. This is true in all regex flavors discussed in this tutorial, even when you turn on "multiline mode". In EditPad Pro and PowerGREP, where the caret and dollar always match at the start and end of lines, «`\A`» and «`\Z`» only match at the start and the end of the entire file.

Zero-Length Matches

We saw that the anchors match at a position, rather than matching a character. This means that when a regex only consists of one or more anchors, it can result in a zero-length match. Depending on the situation, this can be very useful or undesirable. Using «`^\d*$`» to test if the user entered a number (notice the use of the star instead of the plus), would cause the script to accept an empty string as a valid input. See below.

However, matching only a position can be very useful. In email, for example, it is common to prepend a "greater than" symbol and a space to each line of the quoted message. In VB.NET, we can easily do this with `Dim Quoted as String = Regex.Replace(Original, "^", "> ", RegexOptions.Multiline)`. We are using multi-line mode, so the regex «`^`» matches at the start of the quoted message, and after each newline. The Regex.Replace method will remove the regex match from the string, and insert the replacement string (greater than symbol and a space). Since the match does not include any characters, nothing is deleted. However, the match does include a starting position, and the replacement string is inserted there, just like we want it.

Strings Ending with a Line Break

Even though «`\Z`» and «`$`» only match at the end of the string (when the option for the caret and dollar to match at embedded line breaks is off), there is one exception. If the string ends with a line break, then «`\Z`» and «`$`» will match at the position before that line break, rather than at the very end of the string. This "enhancement" was introduced by Perl, and is copied by many regex flavors, including Java, .NET and PCRE. In Perl, when reading a line from a file, the resulting string will end with a line break. Reading a line from a file with the text "joe" results in the string "`joe\n`". When applied to this string, both «`^[a-z]+$`» and «`\A[a-z]+\Z`» will match „`joe`".

If you only want a match at the absolute very end of the string, use «`\z`» (lower case z instead of upper case Z). «`\A[a-z]+\z`» does not match "`joe\n`". «`\z`» matches after the line break, which is not matched by the character class.

Looking Inside the Regex Engine

Let's see what happens when we try to match «`^4$`» to "`749\n486\n4`" (where `\n` represents a newline character) in multi-line mode. As usual, the regex engine starts at the first character: "`7`". The first token in the regular expression is «`^`». Since this token is a zero-width token, the engine does not try to match it with the character, but rather with the position before the character that the regex engine has reached so far. «`^`» indeed matches the position before "`7`". The engine then advances to the next regex token: «`4`». Since the previous token was zero-width, the regex engine does *not* advance to the next character in the string. It remains at "`7`". «`4`» is a literal character, which does not match "`7`". There are no other permutations of the regex, so the engine starts again with the first regex token, at the next character: “4”. This time, «^» cannot match at the position before the 4. This position is preceded by a character, and that character is not a newline. The engine continues at “9”, and fails again. The next attempt, at “\n”, also fails. Again, the position before “\n” is preceded by a character, “9”, and that character is not a newline.

Then, the regex engine arrives at the second “4” in the string. The «^» can match at the position before the “4”, because it is preceded by a newline character. Again, the regex engine advances to the next regex token, «4», but does not advance the character position in the string. «4» matches „4”, and the engine advances both the regex token and the string character. Now the engine attempts to match «$» at the position before (indeed: before) the “8”. The dollar cannot match here, because this position is followed by a character, and that character is not a newline.

Yet again, the engine must try to match the first token again. Previously, it was successfully matched at the second “4”, so the engine continues at the next character, “8”, where the caret does not match. Same at the six and the newline.

Finally, the regex engine tries to match the first token at the third “4” in the string. With success. After that, the engine successfully matches «4» with „4”. The current regex token is advanced to «$», and the current character is advanced to the very last position in the string: the void after the string. No regex token that needs a character to match can match here. Not even a negated character class. However, we are trying to match a dollar sign, and the mighty dollar is a strange beast. It is zero-width, so it will try to match the position before the current character. It does not matter that this “character” is the void after the string. In fact, the dollar will check the current character. It must be either a newline, or the void after the string, for «$» to match the position before the current character. Since that is the case after the example, the dollar matches successfully. Since «$» was the last token in the regex, the engine has found a successful match: the last „4” in the string.

Another Inside Look

Earlier I mentioned that «^\d*$» would successfully match an empty string. Let's see why. There is only one “character” position in an empty string: the void after the string. The first token in the regex is «^». It matches the position before the void after the string, because it is preceded by the void before the string. The next token is «\d*». As we will see later, one of the star's effects is that it makes the «\d», in this case, optional. The engine will try to match «\d» with the void after the string. That fails, but the star turns the failure of the «\d» into a zero-width success. The engine will proceed with the next regex token, without advancing the position in the string. So the engine arrives at «$», and the void after the string. We already saw that those match. At this point, the entire regex has matched the empty string, and the engine reports success.

Caution for Programmers

A regular expression such as «$» all by itself can indeed match after the string. If you would query the engine for the character position, it would return the length of the string if string indices are zero-based, or the length+1 if string indices are one-based in your programming language. If you would query the engine for the length of the match, it would return zero.

What you have to watch out for is that String[Regex.MatchPosition] may cause an access violation or segmentation fault, because MatchPosition can point to the void after the string. This can also happen with «^» and «^$» if the last character in the string is a newline.

7. 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 four 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 a word character and a non-word character following right after the word character.
- Between a non-word character and a word character following right after the non-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". The exact list of characters is different for each regex flavor, but all word characters are always matched by the short-hand character class «\w». All non-word characters are always matched by «\W».

In Perl and the other regex flavors discussed in this tutorial, there is only one metacharacter that matches both before a word and after a word. This is because any position between characters can never be both at the start and at the end of a word. Using only one operator makes things easier for you.

Note that «\w» usually also matches digits. So «\b4\b» can be used to match a 4 that is not part of a larger number. This regex will not match "`44 sheets of a4`". So saying "«\b» matches before and after an alphanumeric sequence“ is more exact than saying ”before and after a word".

Negated Word Boundary

«\B» is the negated version of «\b». «\B» matches at every position where «\b» does not. Effectively, «\B» matches at any position between two word characters as well as at any position between two non-word characters.

Looking Inside the Regex Engine

Let's see what happens when we apply the regex «\bis\b» to the string "`This island is beautiful`". The engine starts with the first token «\b» at the first character "`T`". Since this token is zero-length, the position before the character is inspected. «\b» matches here, because the T is a word character and the character before it is the void before the start of the string. The engine continues with the next token: the literal «i». The engine does not advance to the next character in the string, because the previous regex token was zero-width. «i» does not match "`T`", so the engine retries the first token at the next character position.

«\b» cannot match at the position between the "`T`" and the "`h`". It cannot match between the "`h`" and the "`i`" either, and neither between the "`i`" and the "`s`".

The next character in the string is a space. «\b» matches here because the space is not a word character, and the preceding character is. Again, the engine continues with the «i» which does not match with the space.

Advancing a character and restarting with the first regex token, «`\b`» matches between the space and the second “`i`” in the string. Continuing, the regex engine finds that «`i`» matches „`i`” and «`s`» matches „`s`”. Now, the engine tries to match the second «`\b`» at the position before the “`l`”. This fails because this position is between two word characters. The engine reverts to the start of the regex and advances one character to the “`s`” in “`island`”. Again, the «`\b`» fails to match and continues to do so until the second space is reached. It matches there, but matching the «`i`» fails.

But «`\b`» matches at the position before the third “`i`” in the string. The engine continues, and finds that «`i`» matches „`i`” and «`s`» matches «`s`». The last token in the regex, «`\b`», also matches at the position before the second 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`”.

8. Alternation with The Vertical Bar or Pipe Symbol

I already explained how you can use character classes to match a single character out of several possible characters. Alternation is similar. You can use alternation to match a single regular expression out of several possible regular expressions.

If you want to search for the literal text «`cat`» or «`dog`», separate both options with a vertical bar or pipe symbol: «`cat|dog`». If you want more options, simply expand the list: «`cat|dog|mouse|fish`» .

The alternation operator has the lowest precedence of all regex operators. That is, it tells the regex engine to match either everything to the left of the vertical bar, or everything to the right of the vertical bar. If you want to limit the reach of the alternation, you will need to use round brackets for grouping. If we want to improve the first example to match whole words only, we would need to use «`\b(cat|dog)\b`». This tells the regex engine to find a word boundary, then either "cat" or "dog", and then another word boundary. If we had omitted the round brackets, the regex engine would have searched for "a word boundary followed by cat", or, "dog followed by a word boundary.

Remember That The Regex Engine Is Eager

I already explained that the regex engine is eager. It will stop searching as soon as it finds a valid match. The consequence is that in certain situations, the order of the alternatives matters. Suppose you want to use a regex to match a list of function names in a programming language: Get, GetValue, Set or SetValue. The obvious solution is «`Get|GetValue|Set|SetValue`». Let's see how this works out when the string is "`SetValue`".

The regex engine starts at the first token in the regex, «`G`», and at the first character in the string, "`S`". The match fails. However, the regex engine studied the entire regular expression before starting. So it knows that this regular expression uses alternation, and that the entire regex has not failed yet. So it continues with the second option, being the second «`G`» in the regex. The match fails again. The next token is the first «`S`» in the regex. The match succeeds, and the engine continues with the next character in the string, as well as the next token in the regex. The next token in the regex is the «`e`» after the «`S`» that just successfully matched. «`e`» matches „`e`". The next token, «`t`» matches „`t`".

At this point, the third option in the alternation has been successfully matched. Because the regex engine is eager, it considers the entire alternation to have been successfully matched as soon as one of the options has. In this example, there are no other tokens in the regex outside the alternation, so the entire regex has successfully matched „`Set`" in "`SetValue`".

Contrary to what we intended, the regex did not match the entire string. There are several solutions. One option is to take into account that the regex engine is eager, and change the order of the options. If we use «`GetValue|Get|SetValue|Set`», «`SetValue`» will be attempted before «`Set`», and the engine will match the entire string. We could also combine the four options into two and use the question mark to make part of them optional: «`Get(Value)?|Set(Value)?`». Because the question mark is greedy, «`SetValue`» will be attempted before «`Set`». The best option is probably to express the fact that we only want to match complete words. We do not want to match Set or SetValue if the string is "`SetValueFunction`". So the solution is «`\b(Get|GetValue|Set|SetValue)\b`» or «`\b(Get(Value)?|Set(Value)?)\b`». Since all options have the same end, we can optimize this further to «`\b(Get|Set)(Value)?\b`» .

9. Optional Items

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

You can make several tokens optional by grouping them together using round brackets, and placing the question mark after the closing bracket. E.g.: «Nov(ember)?» will match „Nov” and „November”.

You can write a regular expression that matches many alternatives by including more than one question mark. «Feb(ruary)? 23(rd)?» matches „February 23rd”, „February 23”, „Feb 23rd” and „Feb 23”.

Important Regex Concept: Greediness

With the question mark, I have introduced the first metacharacter that is *greedy*. The question mark gives the regex engine two choices: try to match the part the question mark applies to, or do not try to match it. The engine will always try to match that part. Only if this causes the entire regular expression to fail, will the engine try ignoring the part the question mark applies to.

The effect is that if you apply the regex «Feb 23(rd)?» to the string “Today is Feb 23rd, 2003”, the match will always be „Feb 23rd” and not „Feb 23”. You can make the question mark *lazy* (i.e. turn off the greediness) by putting a second question mark after the first.

I will say a lot more about greediness when discussing the other repetition operators.

Looking Inside The Regex Engine

Let's apply the regular expression «colou?r» to the string “The colonel likes the color green”.

The first token in the regex is the literal «c». The first position where it matches successfully is the „c” in “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 will skip ahead to the next regex token: «r». But this fails to match “n” as well. Now, the engine can only conclude that the entire regular expression cannot be matched starting at the „c” in “colonel”. Therefore, the engine starts again trying to match «c» to the first o in “colonel”.

After a series of failures, «c» will match with the „c” in “color”, and «o», «l» and «o» match the following characters. Now the engine checks whether «u» matches “r”. This fails. Again: no problem. The question mark allows the engine to continue with «r». This matches „r” and the engine reports that the regex successfully matched „color” in our string.

10. Repetition with Star and Plus

I already introduced one repetition operator or quantifier: the question mark. It tells the engine to attempt match the preceding token zero times or once, in effect making it optional.

The asterisk or star tells the engine to attempt to match the preceding token zero or more times. The plus tells the engine to attempt to match the preceding token once or more. «`<[A-Za-z][A-Za-z0-9]*>`» matches an HTML tag without any attributes. The sharp brackets are literals. The first character class matches a letter. The second character class matches a letter or digit. The star repeats the second character class. Because we used the star, it's OK if the second character class matches nothing. So our regex will match a tag like „`<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

Modern regex flavors, like those discussed in this tutorial, have an additional repetition operator that allows you to specify how many times a token can be repeated. The syntax is `{`*`min`*`,`*`max`*`}`, where *min* is a positive integer number indicating the minimum number of matches, and *max* is an integer equal to or greater than *min* indicating the maximum number of matches. If the comma is present but *max* is omitted, the maximum number of matches is infinite. So «`{0,}`» is the same as «`*`», and «`{1,}`» is the same as «`+`». Omitting both the comma and *max* tells the engine to repeat the token exactly *min* times.

You could use «`\b[1-9][0-9]{3}\b`» to match a number between 1000 and 9999. «`\b[1-9][0-9]{2,4}\b`» matches a number between 100 and 99999. Notice the use of the word boundaries.

Watch Out for The Greediness!

Suppose you want to use a regex to match an HTML tag. You know that the input will be a valid HTML file, so the regular expression does not need to exclude any invalid use of sharp brackets. If it sits between sharp brackets, it is an HTML tag.

Most people new to regular expressions will attempt to use «`<.+>`». They will be surprised when they test it on a string like “`This is a <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. You can do that by putting a question markbehind 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 is used repeatedly in a tight loop in a script that you are writing, or perhaps in a custom syntax coloring scheme for EditPad Pro.

Finally, remember that this tutorial only talks about regex-directed engines. Text-directed engines do not backtrack. They do not get the speed penalty, but they also do not support lazy repetition operators.

11. Use Round Brackets for Grouping

By placing part of a regular expression inside round brackets or parentheses, you can group that part of the regular expression together. This allows you to apply a regex operator, e.g. a repetition operator, to the entire group. I have already used round brackets for this purpose in previous topics throughout this tutorial.

Note that only round brackets can be used for grouping. Square brackets define a character class, and curly braces are used by a special repetition operator.

Round Brackets Create a Backreference

Besides grouping part of a regular expression together, round brackets also create a "backreference". A backreference stores the part of the string matched by the part of the regular expression inside the parentheses.

That is, unless you use non-capturing parentheses. Remembering part of the regex match in a backreference, slows down the regex engine because it has more work to do. If you do not use the backreference, you can speed things up by using non-capturing parentheses, at the expense of making your regular expression slightly harder to read.

The regex «`Set(Value)?`» matches „`Set`" or „`SetValue`". In the first case, the first backreference will be empty, because it did not match anything. In the second case, the first backreference will contain „`Value`".

If you do not use the backreference, you can optimize this regular expression into «`Set(?:Value)?`». The question mark and the colon after the opening round bracket are the special syntax that you can use to tell the regex engine that this pair of brackets should not create a backreference. Note the question mark after the opening bracket is unrelated to the question mark at the end of the regex. That question mark is the regex operator that makes the previous token optional. This operator cannot appear after an opening round bracket, because an opening bracket by itself is not a valid regex token. Therefore, there is no confusion between the question mark as an operator to make a token optional, and the question mark as a character to change the properties of a pair of round brackets. The colon indicates that the change we want to make is to turn off capturing the backreference.

How to Use Backreferences

Backreferences allow you to reuse part of the regex match. You can reuse it inside the regular expression (see below), or afterwards. What you can do with it afterwards, depends on the tool you are using. In EditPad Pro or PowerGREP, you can use the backreference in the replacement text during a search-and-replace operation by typing `\1` (backslash one) into the replacement text. If you searched for «`EditPad (Lite|Pro)`» and use "`\1 version`" as the replacement, the actual replacement will be "`Lite version`" in case „`EditPad Lite`" was matched, and "`Pro version`" in case „`EditPad Pro`" was matched.

EditPad Pro and PowerGREP have a unique feature that allows you to change the case of the backreference. `\U1` inserts the first backreference in uppercase, `\L1` in lowercase and `\F1` with the first character in uppercase and the remainder in lowercase. Finally, `\I1` inserts it with the first letter of each word capitalized, and the other letters in lowercase.

Regex libraries in programming languages also provide access to the backreference. In Perl, you can use the magic variables `$1`, `$2`, etc. to access the part of the string matched by the backreference. In .NET (dot net), you can use the `Match` object that is returned by the `Match` method of the `Regex` class. This object has a property called `Groups`, which is a collection of Group objects. To get the string matched by the third backreference in C#, you can use `MyMatch.Groups[3].Value`.

The .NET (dot net) Regex class also has a method `Replace` that can do a regex-based search-and-replace on a string. In the replacement text, you can use `$1`, `$2`, etc. to insert backreferences.

To figure out the number of a particular backreference, scan the regular expression from left to right and count the opening round brackets. The first bracket starts backreference number one, the second number two, etc. Non-capturing parentheses are not counted. This fact means that non-capturing parentheses have another benefit: you can insert them into a regular expression without changing the numbers assigned to the backreferences. This can be very useful when modifying a complex regular expression.

The Entire Regex Match As Backreference Zero

Certain tools make the entire regex match available as backreference zero. In EditPad Pro or PowerGREP, you can use the entire regex match in the replacement text during a search and replace operation by typing `\0` (backslash zero) into the replacement text. In Perl, the magic variable `$&` holds the entire regex match. Libraries like .NET (dot net) where backreferences are made available as an array or numbered list, the item with index zero holds the entire regex match. Using backreference zero is more efficient than putting an extra pair of round brackets around the entire regex, because that would force the engine to continuously keep an extra copy of the entire regex match.

Using Backreferences in The Regular Expression

Backreferences can not only be used after a match has been found, but also during the match. Suppose you want to match a pair of opening and closing HTML tags, and the text in between. By putting the opening tag into a backreference, we can reuse the name of the tag for the closing tag. Here's how: «`<([A-Z][A-Z0-9]*)[^>]*>.*?</\1>`» . This regex contains only one pair of parentheses, which capture the string matched by «`[A-Z][A-Z0-9]*`» into the first backreference. This backreference is reused with «`\1`» (backslash one). The «`/`» before it is simply the forward slash in the closing HTML tag that we are trying to match.

You can reuse the same backreference more than once. «`([a-c])x\1x\1`» will match „`axaxa`”, „`bxbxb`” and „`cxcxc`”. If a backreference was not used in a particular match attempt (such as in the first example where the question mark made the first backreference optional), it is simply empty. Using an empty backreference in the regex is perfectly fine. It will simply be replaced with nothingness.

A backreference cannot be used inside itself. «`([abc]\1)`» will not work. Depending on your regex flavor, it will either give an error message, or it will fail to match anything without an error message. Therefore, \0 cannot be used inside a regex, only in the replacement.

Looking Inside The Regex Engine

Let's see how the regex engine applies the above regex to the string "`Testing <B><I>bold italic</I></B> text`". The first token in the regex is the literal «`<`». The regex engine will traverse the string until it can match at the first „`<`" in the string. The next token is «`[A-Z]`». The regex engine also takes note that it is now inside the first pair of capturing parentheses. «`[A-Z]`» matches „`B`". The engine advances to «`[A-Z0-9]`» and "`>`". This match fails. However, because of the star, that's perfectly fine. The position in the string remains at "`>`". The position in the regex is advanced to «`[^>]`».

This step crosses the closing bracket of the first pair of capturing parentheses. This prompts the regex engine to store what was matched inside them into the first backreference. In this case, „`B`" is stored.

After storing the backreference, the engine proceeds with the match attempt. «`[^>]`» does not match „`>`". Again, because of another star, this is not a problem. The position in the string remains at "`>`", and position in the regex is advanced to «`>`». These obviously match. The next token is a dot, repeated by a lazy star. Because of the laziness, the regex engine will initially skip this token, taking note that it should backtrack in case the remainder of the regex fails.

The engine has now arrived at the second «`<`» in the regex, and the second "`<`" in the string. These match. The next token is «`/`». This does not match "`I`", and the engine is forced to backtrack to the dot. The dot matches the second „`<`" in the string. The star is still lazy, so the engine again takes note of the available backtracking position and advances to «`<`» and "`I`". These do not match, so the engine again backtracks.

The backtracking continues until the dot has consumed „`<I>bold italic`". At this point, «`<`» matches the third „`<`" in the string, and the next token is «`/`» which matches "`/`". The next token is «`\1`». Note that the token the backreference, and not «`B`». The engine does not substitute the backreference in the regular expression. Every time the engine arrives at the backreference, it will read the value that was stored. This means that if the engine had backtracked beyond the first pair of capturing parentheses before arriving the second time at «`\1`», the new value stored in the first backreference would be used. But this did not happen here, so „`B`" it is. This fails to match at "`I`", so the engine backtracks again, and the dot consumes the third "`<`" in the string.

Backtracking continues again until the dot has consumed „`<I>bold italic</I>`". At this point, «`<`» matches „`<`" and «`/`» matches „`/`". The engine arrives again at «`\1`». The backreference still holds „`B`". «`B`» matches „`B`". The last token in the regex, «`>`» matches „`>`". A complete match has been found: „`<B><I>bold italic</I></B>`".

Repetition and Backreferences

As I mentioned in the above inside look, the regex engine does not permanently substitute backreferences in the regular expression. It will use the last match saved into the backreference each time it needs to be used. If a new match is found by capturing parentheses, the previously saved match is overwritten. There is a clear difference between «`([abc]+)`» and «`([abc])+`». Though both successfully match „`cab`", the first regex will put „`cab`" into the first backreference, while the second regex will only store „`b`". That is because in the second regex, the plus caused the pair of parentheses to repeat three times. The first time, „`c`" was stored. The second time „`a`" and the third time „`b`". Each time, the previous value was overwritten, so „`b`" remains.

This also means that «`([abc]+)=\1`» will match „`cab=cab`", and that «`([abc])+=\1`» will not. The reason is that when the engine arrives at «`\1`», it holds «`b`» which fails to match "`c`". Obvious when you look at a simple example like this one, but a common cause of difficulty with regular expressions nonetheless. When using backreferences, always double check that you are really capturing what you want.

Useful Example: Checking for Doubled Words

When editing text, doubled words such as “the the” easily creep in. Using the regex «`\b(\w+)\s+\1\b`» in your text editor, you can easily find them. To delete the second word, simply type in “`\1`” as the replacement text and click the Replace button.

Parentheses and Backreferences Cannot Be Used Inside Character Classes

Round brackets cannot be used inside character classes, at least not as metacharacters. When you put a round bracket in a character class, it is treated as a literal character. So the regex «`[(a)b]`» matches „`a`”, „`b`”, „`(`” and „`)`”.

Backreferences also cannot be used inside a character class. The `\1` in regex like «`(a)[\1b]`» will be interpreted as an octal escape in most regex flavors. So this regex will match an „`a`” followed by either «`\x01`» or a «`b`».

12. Named Capturing Groups

All modern regular expression engines support capturing groups, which are numbered from left to right, starting with one. The numbers can then be used in backreferences to match the same text again in the regular expression, or to use part of the regex match for further processing. In a complex regular expression with many capturing groups, the numbering can get a little confusing.

Named Capture with Python, PCRE and PHP

Python's regex module was the first to offer a solution: named capture. By assigning a name to a capturing group, you can easily reference it by name. «`(?P<name>group)`» captures the match of «`group`» into the backreference "name". You can reference the contents of the group with the numbered backreference «`\1`» or the named backreference «`(?P=name)`».

The open source PCRE library has followed Python's example, and offers named capture using the same syntax. The PHP preg functions offer the same functionality, since they are based on PCRE.

Python's `sub()` function allows you to reference a named group as "`\1`" or "`\g<name>`". This does *not* work in PHP. In PHP, you can use double-quoted string interpolation with the `$regs` parameter you passed to `pcre_match()`: "`$regs['name']`".

Named Capture with .NET's System.Text.RegularExpressions

The regular expression classes of the .NET framework also support named capture. Unfortunately, the Microsoft developers decided to invent their own syntax, rather than follow the one pioneered by Python. Currently, no other regex flavor supports Microsoft's version of named capture.

Here is an example with two capturing groups in .NET style: «`(?<first>group)(?'second'group)`». As you can see, .NET offers two syntaxes to create a capturing group: one using sharp brackets, and the other using single quotes. The first syntax is preferable in strings, where single quotes may need to be escaped. The second syntax is preferable in ASP code, where the sharp brackets are used for HTML tags. You can use the pointy bracket flavor and the quoted flavors interchangeably.

To reference a capturing group inside the regex, use «`\k<name>`» or «`\k'name'`». Again, you can use the two syntactic variations interchangeably.

When doing a search-and-replace, you can reference the named group with the familiar dollar sign syntax: "`${name}`". Simply use a name instead of a number between the curly braces.

Names and Numbers for Capturing Groups

Here is where things get a bit ugly. Python and PCRE treat named capturing groups just like unnamed capturing groups, and number both kinds from left to right, starting with one. The regex «`(a)(?P<x>b)(c)(?P<y>d)`» matches „`abcd`" as expected. If you do a search-and-replace with this regex and the replacement “\1\2\3\4”, you will get “abcd”. All four groups were numbered from left to right, from one till four. Easy and logical.

Things are quite a bit more complicated with the .NET framework. The regex «(a)(?<x>b)(c)(?<y>d)» again matches „abcd”. However, if you do a search-and-replace with “$1$2$3$4” as the replacement, you will get “acbd”. Probably not what you expected.

The .NET framework *does* number named capturing groups from left to right, but numbers them *after* all the unnamed groups have been numbered. So the unnamed groups «(a)» and «(c)» get numbered first, from left to right, starting at one. Then the named groups «(?<x>b)» and «(?<y>d)» get their numbers, continuing from the unnamed groups, in this case: three.

To make things simple, when using .NET's regex support, just assume that named groups do not get numbered at all, and reference them by name exclusively. To keep things compatible across regex flavors, I strongly recommend that you do not mix named and unnamed capturing groups at all. Either give a group a name, or make it non-capturing as in «(?:nocapture)». Non-capturing groups are more efficient, since the regex engine does not need to keep track of their matches.

Other Regex Flavors

EditPad Pro and PowerGREP support both the Python syntax and the .NET syntax for named capture. However, they will number named groups along with unnamed capturing groups, just like Python does.

RegexBuddy also supports both Python's and Microsoft's style. RegexBuddy will convert one flavor of named capture into the other when generating source code snippets for Python, PHP/preg, PHP, or one of the .NET languages.

None of the other regex flavors discussed in this book support named capture.

13. Unicode Regular Expressions

Unicode is a character set that aims to define all characters and glyphs from all human languages, living and dead. With more and more software being required to support multiple languages, or even just *any* language, Unicode has been strongly gaining popularity in recent years. Using different character sets for different languages is simply too cumbersome for programmers and users.

Unfortunately, Unicode brings its own requirements and pitfalls when it comes to regular expressions. Of the regex flavors discussed in this tutorial, Java and the .NET framework use Unicode-based regex engines. Perl supports Unicode starting with version 5.6.

RegexBuddy's regex engine is fully Unicode-based starting with version 2.0.0. RegexBuddy 1.x.x did not support Unicode at all. PowerGREP uses the same Unicode regex engine starting with version 3.0.0. Earlier versions would convert Unicode files to ANSI prior to grepping with an 8-bit (i.e. non-Unicode) regex engine.

Characters, Code Points and Graphemes or How Unicode Makes a Mess of Things

Most people would consider "à" a single character. Unfortunately, it need not be depending on the meaning of the word "character".

All regex engines discussed in this tutorial treat any single Unicode *code point* as a single character. When this tutorial tells you that the dot matches any single character, this translates into Unicode parlance as "the dot matches any single Unicode code point". In Unicode, "à" can be encoded as two code points: U+0061 (a) followed by U+0300 (grave accent). In this situation, «.» applied to "à" will match „a" without the accent. «^.$» will fail to match, since the string consists of two code points. «^..$» matches „à".

The Unicode code point U+0300 (grave accent) is a *combining mark*. Any code point that is not a combining mark can be followed by any number of combining marks. This sequence, like U+0061 U+0300 above, is displayed as a single *grapheme*on the screen.

Unfortunately, "à" can also be encoded with the single Unicode code point U+00E0 (a with grave accent). The reason for this duality is that many historical character sets encode "a with grave accent" as a single character. Unicode's designers thought it would be useful to have a one-on-one mapping with popular legacy character sets, in addition to the Unicode way of separating marks and base letters (which makes arbitrary combinations not supported by legacy character sets possible).

How to Match a Single Unicode Grapheme

Matching a single grapheme, whether it's encoded as a single code point, or as multiple code points using combining marks, is easy in Perl, RegexBuddy and PowerGREP: simply use «\X». You can consider «\X» the Unicode version of the dot in regex engines that use plain ASCII. There is one difference, though: «\X» always matches line break characters, whereas the dot does not match line break characters unless you enable the dot matches newline matching mode.

Java and .NET unfortunately do not support «`\X`» (yet). Use «`\P{M}\p{M}*`» as a substitute. To match any number of graphemes, use «`(?:\P{M}\p{M}*)+`» instead of «`\X+`».

Unicode Character Properties

In addition to complications, Unicode also brings new possibilities. One is that each Unicode character belongs to a certain category. You can match a single character belonging to a particular category with «`\p{}`». You can match a single character *not* belonging to a particular category with «`\P{}`».

Again, "character" really means "Unicode code point". «`\p{L}`» matches a single code point in the category "letter". If your input string is "à" encoded as U+0061 U+0300, it matches „a" without the accent. If the input is "à" encoded as U+00E0, it matches „à" with the accent. The reason is that both the code points U+0061 (a) and U+00E0 (à) are in the category "letter", while U+0300 is in the category "mark".

You should now understand why «`\P{M}\p{M}*`» is the equivalent of «`\X`». «`\P{M}`» matches a code point that is not a combining mark, while «`\p{M}*`» matches zero or more code points that are combining marks. To match a letter including any diacritics, use «`\p{L}\p{M}*`». This last regex will always match „à", regardless of how it is encoded.

In addition to the standard notation, «`\p{L}`», Java, Perl, RegexBuddy and PowerGREP allow you to use the shorthand «`\pL`». In addition to that, Perl, RegexBuddy and PowerGREP also support the longhand «`\p{Letter}`».

- «`\p{L}`» or «`\p{Letter}`»: any kind of letter from any language.
    - «`\p{Ll}`» or «`\p{Lowercase_Letter}`»: a lowercase letter that has an uppercase variant.
    - «`\p{Lu}`» or «`\p{Uppercase_Letter}`»: an uppercase letter that has a lowercase variant.
    - «`\p{Lt}`» or «`\p{Titlecase_Letter}`»: a letter that appears at the start of a word when only the first letter of the word is capitalized.
    - «`\p{L&}`» or «`\p{Letter&}`»: a letter that exists in lowercase and uppercase variants (combination of Ll, Lu and Lt).
    - «`\p{Lm}`» or «`\p{Modifier_Letter}`»: a special character that is used like a letter.
    - «`\p{Lo}`» or «`\p{Other_Letter}`»: a letter or ideograph that does not have lowercase and uppercase variants.
- «`\p{M}`» or «`\p{Mark}`»: a character intended to be combined with another character (e.g. accents, umlauts, enclosing boxes, etc.).
    - «`\p{Mn}`» or «`\p{Non_Spacing_Mark}`»: a character intended to be combined with another character that does not take up extra space (e.g. accents, umlauts, etc.).
    - «`\p{Mc}`» or «`\p{Spacing_Combining_Mark}`»: a character intended to be combined with another character that takes up extra space (vowel signs in many Eastern languages).
    - «`\p{Me}`» or «`\p{Enclosing_Mark}`»: a character that encloses the character is is combined with (circle, square, keycap, etc.).
- «`\p{Z}`» or «`\p{Separator}`»: any kind of whitespace or invisible separator.
    - «`\p{Zs}`» or «`\p{Space_Separator}`»: a whitespace character that is invisible, but does take up space.
    - «`\p{Zl}`» or «`\p{Line_Separator}`»: line separator character U+2028.
    - «`\p{Zp}`» or «`\p{Paragraph_Separator}`»: paragraph separator character U+2029.
- «`\p{S}`» or «`\p{Symbol}`»: math symbols, currency signs, dingbats, box-drawing characters, etc..
    - «`\p{Sm}`» or «`\p{Math_Symbol}`»: any mathematical symbol.
    - «`\p{Sc}`» or «`\p{Currency_Symbol}`»: any currency sign.

  - «`\p{Sk}`» or «`\p{Modifier_Symbol}`»: a combining character (mark) as a full character on its own.
  - «`\p{So}`» or «`\p{Other_Symbol}`»: various symbols that are not math symbols, currency signs, or combining characters.
- «`\p{N}`» or «`\p{Number}`»: any kind of numeric character in any script.
  - «`\p{Nd}`» or «`\p{Decimal_Digit_Number}`»: a digit zero through nine in any script except ideographic scripts.
  - «`\p{Nl}`» or «`\p{Letter_Number}`»: a number that looks like a letter, such as a Roman numeral.
  - «`\p{No}`» or «`\p{Other_Number}`»: a superscript or subscript digit, or a number that is not a digit 0..9 (excluding numbers from ideographic scripts).
- «`\p{P}`» or «`\p{Punctuation}`»: any kind of punctuation character.
  - «`\p{Pd}`» or «`\p{Dash_Punctuation}`»: any kind of hyphen or dash.
  - «`\p{Ps}`» or «`\p{Open_Punctuation}`»: any kind of opening bracket.
  - «`\p{Pe}`» or «`\p{Close_Punctuation}`»: any kind of closing bracket.
  - «`\p{Pi}`» or «`\p{Initial_Punctuation}`»: any kind of opening quote.
  - «`\p{Pf}`» or «`\p{Final_Punctuation}`»: any kind of closing quote.
  - «`\p{Pc}`» or «`\p{Connector_Punctuation}`»: a punctuation character such as an underscore that connects words.
  - «`\p{Po}`» or «`\p{Other_Punctuation}`»: any kind of punctuation character that is not a dash, bracket, quote or connector.
- «`\p{C}`» or «`\p{Other}`»: invisible control characters and unused code points.
  - «`\p{Cc}`» or «`\p{Control}`»: an ASCII 0x00..0x1F or Latin-1 0x80..0x9F control character.
  - «`\p{Cf}`» or «`\p{Format}`»: invisible formatting indicator.
  - «`\p{Co}`» or «`\p{Private_Use}`»: any code point reserved for private use.
  - «`\p{Cs}`» or «`\p{Surrogate}`»: one half of a surrogate pair in UTF-16 encoding.
  - «`\p{Cn}`» or «`\p{Unassigned}`»: any code point to which no character has been assigned.

Do You Need To Worry About Different Encodings?

While you should always keep in mind the pitfalls created by the different ways in which accented characters can be encoded, you don't always have to worry about them. If you know that your input string and your regex use the same style, then you don't have to worry about it at all. This process is called Unicode *normalization*. All programming languages with native Unicode support, such as Java, C# and VB.NET, have library routines for normalizing strings. If you normalize both the subject and regex before attempting the match, there won't be any inconsistencies.

If you are using Java, you can pass the CANON_EQ flag as the second parameter to Pattern.compile(). This tells the Java regex engine to consider *canonically equivalent* characters as identical. E.g. the regex «à» encoded as U+00E0 will match „à" encoded as U+0061 U+0300, and vice versa. None of the other regex engines currently support canonical equivalence while matching.

If you type the à key on the keyboard, all word processors that I know of will insert the code point U+00E0 into the file. So if you're working with text that you typed in yourself, any regex that you type in yourself will match in the same way.

Finally, if you're using PowerGREP to search through text files encoded using a traditional Windows (often called "ANSI") or ISO-8859 code page, PowerGREP will always use the one-on-one substitution. Since all the Windows or ISO-8859 code pages encode accented characters as a single code point, all software that I know of will use a single Unicode code point for each character when converting the file to Unicode.

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. E.g. «`\u00E0`» matches „à”, but only when encoded as a single code point U+00E0.

In Java, the regex token «`\uFFFF`» only matches the specified code point, even when you turned on canonical equivalence. However, the same syntax `\uFFFF` is also used to insert Unicode characters into literal strings in the Java source code. `Pattern.compile("\u00E0")` will match both the single-code-point and double-code-point encodings of „à”, while `Pattern.compile("\\u00E0")` matches only the single-code-point version. Remember that when writing a regex as a Java string literal, backslashes must be escaped. The former Java code compiles the regex «à», while the latter compiles «`\u00E0`». Depending on what you're doing, the difference may be significant.

14. Regex Matching Modes

All regular expression engines discussed in this tutorial support the following three matching modes:

- `/i` makes the regex match case insensitive.
- `/s` enables "single-line mode". In this mode, the dot matches newlines.
- `/m` enables "multi-line mode". In this mode, the caret and dollar match before and after newlines in the subject string.

Many regex flavors have additional modes or options that have single letter equivalents, but these differ widely.

Most tools that support regular expressions have checkboxes or similar controls that you can use to turn these modes on or off. Most programming languages allow you to pass option flags when constructing the regex object. E.g. in Perl, `m/regex/i` turns on case insensitivity, while `Pattern.compile(“regex”, Pattern.CASE_INSENSITIVE)` does the same in Java.

Specifying Modes Inside The Regular Expression

Sometimes, the tool or language does not provide the ability to specify matching options. E.g. the handy `String.matches()` method in Java does not take a parameter for matching options like `Pattern.compile()` does. In that situation, you can add a mode modifier to the start of the regex. E.g. `(?i)` turns on case insensitivity, while `(?ism)` turns on all three options.

Turning Modes On and Off for Only Part of The Regular Expression

Modern regex flavors allow you to apply modifiers to only part of the regular expression. If you insert the modifier `(?ism)` in the middle of the regex, the modifier only applies to the part of the regex to the right of the modifier. You can turn off modes by preceding them with a minus sign. All modes after the minus sign will be turned off. E.g. `(?i-sm)` turns on case insensitivity, and turns off both single-line mode and multi-line mode.

Not all regex flavors support this. The latest versions of all tools and languages discussed in this book do. Older regex flavors usually apply the option to the entire regular expression, no matter where you placed it. You can quickly test this. The regex «`(?i)te(?-i)st`» should match „`test`” and „`TEst`”, but not “`teST`” or “`TEST`”.

Modifier Spans

Instead of using two modifiers, one to turn an option on, and one to turn it off, you use a modifier span. «`(?i)ignorecase(?-i)casesensitive(?i)ignorecase`» is equivalent to «`(?i)ignorecase(?-i:casesensitive)ignorecase`». You have probably noticed the resemblance between the modifier span and the non-capturing group «`(?:group)`». Technically, the non-capturing group is a modifier span that does not change any modifiers. It is obvious that the modifier span does not create a backreference.

15. Atomic Grouping and Possessive Quantifiers

When discussing the repetition operators or quantifiers, I explained the difference between greedy and lazy repetition. Greediness and laziness determine the order in which the regex engine tries the possible permutations of the regex pattern. A greedy quantifier will first try to repeat the token as many times as possible, and gradually give up matches as the engine backtracks to find an overall match. A lazy quantifier will first repeat the token as few times as required, and gradually expand the match as the engine backtracks through the regex to find an overall match.

Because greediness and laziness change the order in which permutations are tried, they can change the overall regex match. However, they do not change the fact that the regex engine will backtrack to try all possible permutations of the regular expression in case no match can be found. First, let's see why backtracking can lead to problems.

Catastrophic Backtracking

Recently I got a complaint from a customer that EditPad Pro hung (i.e. it stopped responding) when trying to find lines in a comma-delimited text file where the 12th item on a line started with a "`P`". The customer was using the 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 possiblities 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. This causes software like EditPad Pro to stop responding longer than your patience lasts. Other applications may even crash as the regex engine runs out of memory trying to remember all backtracking positions.

If you try this example with RegexBuddy's debugger, you will see that it needs 48,096 steps to conclude there regex cannot match. If the string is “`1,2,3,4,5,6,7,8,9,10,11,12,13,14`”, just 3 characters more, the number of steps jumps to 96,857. 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. RegexBuddy's debugger will abort the attempt after 100,000 steps, to prevent it from running out of memory.

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.

Atomic Grouping and Possessive Quantifiers

Recent regex flavors have introduced two additional solutions to this problem: atomic grouping and possessive quantifiers. Their purpose is to prevent backtracking, allowing the regex engine to fail faster.

In the above example, we could easily reduce the amount of backtracking to a very low level by better specifying what we wanted. But that is not always possible in such a straightforward manner. In that case, you should use atomic grouping to prevent the regex engine from backtracking.

Using atomic grouping, the above regex becomes «`^(?>(.*?,){11})P`». Everything between `(?>)` is treated as one single token by the regex engine, once the regex engine leaves the group. Because the entire group is one token, no backtracking can take place once the regex engine has found a match for the group. If backtracking is required, the engine has to backtrack to the regex token before the group (the caret in our example). If there is no token before the group, the regex must retry the entire regex at the next position in the string.

Possessive quantifiers are a limited form of atomic grouping with a cleaner notation. To make a quantifier possessive, place a plus after it. «`x++`» is the same as «`(?>x+)`». Similarly, you can use «`x*+`», «`x?+`» and «`x{m,n}+`». Note that you cannot make a lazy quantifier possessive. It would match the minimum number of matches and never expand the match because backtracking is not allowed.

Tool and Language Support for Atomic Grouping and Possessive Quantifiers

Atomic grouping is a recent addition to the regex scene, and only supported by the latest versions of most regex flavors. Perl supports it starting with version 5.6. The Java supports it starting with JDK version 1.4.2, though the JDK documentation uses the term “independent group” rather than “atomic group”. All versions of .NET support atomic grouping, as do recent versions of PCRE, PHP's pgreg functions and Ruby. Python does not support atomic grouping.

At this time, possessive quantifiers are only supported by the Java JDK 1.4.0 and later, and PCRE version 4 and later.

The latest versions of EditPad Pro and PowerGREP support both atomic grouping and possessive quantifiers, as do all versions of RegexBuddy.

Atomic Grouping Inside The Regex Engine

Let's see how «`^(?>(.*?,){11})P`» is applied to "`1,2,3,4,5,6,7,8,9,10,11,12,13`". The caret matches at the start of the string and the engine enters the atomic group. The star is lazy, so the dot is initially skipped. But the comma does not match "`1`", so the engine backtracks to the dot. That's right: backtracking is allowed here. The star is not possessive, and is not immediately enclosed by an atomic group. That is, the regex engine did not cross the closing round bracket of the atomic group. The dot matches „`1`", and the comma matches too. «`{11}`» causes further repetition until the atomic group has matched „`1,2,3,4,5,6,7,8,9,10,11,`".

Now, the engine leaves the atomic group. Because the group is atomic, all backtracking information is discarded and the group is now considered a single token. The engine now tries to match «`P`» to the "`1`" in the 12th field. This fails.

So far, everything happened just like in the original, troublesome regular expression. Now comes the difference. «`P`» failed to match, so the engine backtracks. The previous token is an atomic group, so the group's entire match is discarded and the engine backtracks further to the caret. The engine now tries to match the caret at the next position in the string, which fails. The engine walks through the string until the end, and declares failure. Failure is declared after 30 attempts to match the caret, and just one attempt to match the atomic group, rather than after 30 attempts to match the caret and a huge number of attempts to try all combinations of both quantifiers in the regex.

That is what atomic grouping and possessive quantifiers are for: efficiency by disallowing backtracking. The most efficient regex for our problem at hand would be «`^(?>((?>[^,\r\n]*),){11})P`» , since possessive, greedy repetition of the star is faster than a backtracking lazy dot. If possessive quantifiers are available, you can reduce clutter by writing «`^(?>([^,\r\n]*+,){11})P`» .

When To Use Atomic Grouping or Possessive Quantifiers

Atomic grouping and possessive quantifiers speed up failure by eliminating backtracking. They do not speed up success, only failure. When nesting quantifiers like in the above example, you really should use atomic grouping and/or possessive quantifiers whenever possible. While «`x[^x]*+x`» and «`x(?>[^x]*)x`» fail faster than «`x[^x]*x`», the increase in speed is minimal. If the final x in the regex cannot be matched, the regex engine backtracks once for each character matched by the star. With simple repetition, the amount of time wasted with pointless backtracking increases in a linear fashion to the length of the string. With combined repetition, the amount of time wasted increases exponentially and will very quickly exhaust the capabilities of your computer. Still, if you are smart about combined repetition, you often can avoid the problem without atomic grouping as in the example above.

If you are simply doing a search in a text editor, using simple repetition, you will not earn back the extra time to type in the characters for the atomic grouping. If the regex will be used in a tight loop in an application, or process huge amounts of data, then atomic grouping may make a difference.

Note that atomic grouping and possessive quantifiers can alter the outcome of the regular expression match. «`\d+6`» will match „`123456`" in "`123456789`". «`\d++6`» will not match at all. «`\d+`» will match the entire string. With the former regex, the engine backtracks until the 6 can be matched. In the latter case, no backtracking is allowed, and the match fails. Again, the cause of this is that the token «`\d`» that is repeated can also match the delimiter «`6`». Sometimes this is desirable, often it is not.

This shows again that understanding how the regex engine works on the inside will enable you to avoid many pitfalls and craft efficient regular expressions that match exactly what you want.

16. Lookahead and Lookbehind Zero-Width Assertions

Perl 5 introduced two very powerful constructs: “lookahead” and “lookbehind”. Collectively, these are called “lookaround”. They are also called “zero-width assertions”. They are zero-width just like the start and end of line, and start and end of word anchors that I already explained. The difference is that lookarounds will actually match characters, but then give up the match and only return the result: match or no match. That is why they are called “assertions”. They do not consume characters in the string, but only assert whether a match is possible or not.

Lookarounds allow you to create regular expressions that are impossible to create without them, or that would get very longwinded without them. All regex flavors discussed in this book support lookaround. The exception is JavaScript, which supports lookahead but not lookbehind.

Positive and Negative Lookahead

Negative lookahead is indispensable if you want to match something not followed by something else. When explaining character classes, I already explained why you cannot use a negated character class to match a “q” not followed by a “u”. Negative lookahead provides the solution: «`q(?!u)`». The negative lookahead construct is the pair of round brackets, with the opening bracket followed by a question mark and an exclamation point. Inside the lookahead, we have the trivial regex «`u`».

Positive lookahead works just the same. «`q(?=u)`» matches a q that is followed by a u, without making the u part of the match. The positive lookahead construct is a pair of round brackets, with the opening bracket followed by a question mark and an equals sign.

You can use any regular expression inside the lookahead. (Note that this is not the case with lookbehind. I will explain why below.) Any valid regular expression can be used inside the lookahead. If it contains capturing parentheses, the backreferences will be saved. Note that the lookahead itself does not create a backreference. So it is not included in the count towards numbering the backreferences. If you want to store the match of the regex inside a backreference, you have to put capturing parentheses around the regex inside the lookahead, like this: «`(?=(regex))`». The other way around will not work, because the lookahead will already have discarded the regex match by the time the backreference is to be saved.

Regex Engine Internals

First, let's see how the engine applies «`q(?!u)`» to the string “`Iraq`”. The first token in the regex is the literal «`q`». As we already know, this will cause the engine to traverse the string until the „`q`” in the string is matched. The position in the string is now the void behind the string. The next token is the lookahead. The engine takes note that it is inside a lookahead construct now, and begins matching the regex inside the lookahead. So the next token is «`u`». This does not match the void behind the string. The engine notes that the regex inside the lookahead failed. Because the lookahead is negative, this means that the lookahead has successfully matched at the current position. At this point, the entire regex has matched, and „`q`” is returned as the match.

Let's try applying the same regex to “`quit`”. «`q`» matches „`q`”. The next token is the «`u`» inside the lookahead. The next character is the “`u`”. These match. The engine advances to the next character: “`i`”. However, it is done with the regex inside the lookahead. The engine notes success, and discards the regex match. This causes the engine to step back in the string to “`u`”.

Because the lookahead is negative, the successful match inside it causes the lookahead to fail. Since there are no other permutations of this regex, the engine has to start again at the beginning. Since «q» cannot match anywhere else, the engine reports failure.

Let's take one more look inside, to make sure you understand the implications of the lookahead. Let's apply «q(?=u)i» to "quit". I have made the lookahead positive, and put a token after it. Again, «q» matches „q" and «u» matches „u". Again, the match from the lookahead must be discarded, so the engine steps back from "i" in the string to "u". To lookahead was successful, so the engine continues with «i». But «i» cannot match "u". So this match attempt fails. All remaining attempts will fail as well, because there are no more q's in the string.

Positive and Negative Lookbehind

Lookbehind has the same effect, but works backwards. It tells the regex engine to temporarily step backwards in the string, to check if the text inside the lookbehind can be matched there. «(?<!a)b» matches a "b" that is not preceded by an "a", using negative lookbehind. It will not match "cab", but will match the „b" (and only the „b") in "bed" or "debt". «(?<=a)b» (positive lookbehind) matches the „b" (and only the „b") in „cab", but does not match "bed" or "debt".

The construct for positive lookbehind is «(?<=text)»: a pair of round brackets, with the opening bracket followed by a question mark, "less than" symbol and an equals sign. Negative lookbehind is written as «(?<!text)», using an exclamation point instead of an equals sign.

More Regex Engine Internals

Let's apply «(?<=a)b» to "thingamabob". The engine starts with the lookbehind and the first character in the string. In this case, the lookbehind tells the engine to step back one character, and see if an "a" can be matched there. The engine cannot step back one character because there are no characters before the "t". So the lookbehind fails, and the engine starts again at the next character, the "h". (Note that a negative lookbehind would have succeeded here.) Again, the engine temporarily steps back one character to check if an "a" can be found there. It finds a "t", so the positive lookbehind fails again.

The lookbehind continues to fail until the regex reaches the "m" in the string. The engine again steps back one character, and notices that the „a" can be matched there. The positive lookbehind matches. Because it is zero-width, the current position in the string remains at the "m". The next token is «b», which cannot match here. The next character is the second "a" in the string. The engine steps back, and finds out that the "m" does not match «a».

The next character is the first "b" in the string. The engine steps back and finds out that „a" satisfies the lookbehind. «b» matches „b", and the entire regex has been matched successfully. It matches one character: the first „b" in the string.

Important Notes About Lookbehind

The good news is that you can use lookbehind anywhere in the regex, not only at the start. If you want to find a word not ending with an "s", you could use «\b\w+(?<!s)\b». This is definitely not the same as

«\b\w+[^s]\b». When applied to "John's", the former will match „John" and the latter „John'" (including the apostrophe). I will leave it up to you to figure out why. (Hint: «\b» matches between the apostrophe and the "s"). The latter will also not match single-letter words like "a" or "I". The correct regex without using lookbehind is «\b\w*[^s\W]\b» (star instead of plus, and \W in the character class). Personally, I find the lookbehind easier to understand. The last regex, which works correctly, has a double negation (the \W in the negated character class). Double negations tend to be confusing to humans. Not to regex engines, though.

The bad news is that most regex flavors do not allow you to use just any regex inside a lookbehind, because they cannot apply a regular expression backwards. Therefore, the regular expression engine needs to be able to figure out how many steps to step back before checking the lookbehind.

Therefore, many regex flavors, including those used by Perl 5 and Python, only allow fixed-length strings. You can use any regex of which the length of the match can be predetermined. This means you can use literal text and character classes. You cannot use repetition or optional items. You can use alternation, but only if all options in the alternation have the same length.

Some regex flavors support the above, plus alternation with strings of different lengths. But each string in the alternation must still be of fixed length, so only literals and character classes can be used. This includes PCRE, PHP and EditPad Pro.

More advanced flavors support the above, plus finite repetition. This means you can still not use the star or plus, but you can use the question mark and the curly braces with the max parameter specified. These regex flavors recognize the fact that finite repetition can be rewritten as an alternation of strings with different, but fixed lengths. The only regex flavor that I know of that currently supports this is Sun's regex package in the JDK 1.4.

The only regex flavors that allow you to use a full regular expression inside lookbehind are those used by RegexBuddy (2.0.3 and later), PowerGREP (3.0.0 and later), and the .NET framework (all versions).

Finally, JavaScript and Ruby do not support lookbehind at all.

Lookaround Is Atomic

The fact that lookaround is zero-width automatically makes it atomic. As soon as the lookaround condition is satisfied, the regex engine forgets about everything inside the lookaround. It will not backtrack inside the lookaround to try different permutations.

The only situation in which this makes any difference is when you use capturing groups inside the lookaround. Since the regex engine does not backtrack into the lookaround, it will not try different permutations of the capturing groups.

For this reason, the regex «(?=(\d+))\w+\1» will never match "123x12". First the lookaround captures „123" into «\1». «\w+» then matches the whole string and backtracks until it matches only „1". Finally, «\w+» fails since «\1» cannot be matched at any position. Now, the regex engine has nothing to backtrack to, and the overall regex fails. The backtracking steps created by «\d+» have been discarded. It never gets to the point where the lookahead captures only "12".

Obviously, the regex engine does try further positions in the string. If we change the subject string, the regex «`(?=(\d+))\w+\1`» will match „`56x56`” in “`456x56`”.

If you don't use capturing groups inside 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.

17. Testing The Same Part of The String for More Than One Requirement

Lookaround, which I introduced in detail in the previous topic, is a very powerful concept. Unfortunately, it is often underused by people new to regular expressions, because lookaround is a bit confusing. The confusing part is that the lookaround is zero-width. So if you have a regex in which a lookahead is followed by another piece of regex, or a lookbehind is preceded by another piece of regex, then the regex will traverse part of the string twice.

To make this clear, I would like to give you another, a bit more practical example. Let's say we want to find a word that is six letters long and contains the three subsequent letters "`cat`". Actually, we can match this without lookaround. We just specify all the options and hump them together using alternation: «`cat\w{3}|\wcat\w{2}|\w{2}cat\w|\w{3}cat`». Easy enough. But this method gets unwieldy if you want to find any word between 6 and 12 letters long containing either "cat", "dog" or "mouse".

Lookaround to The Rescue

In this example, we basically have two requirements for a successful match. First, we want a word that is 6 letters long. Second, the word we found must contain the word "cat".

Matching a 6-letter word is easy with «`\b\w{6}\b`». Matching a word containing "cat" is equally easy: «`\b\w*cat\w*\b`».

Combining the two, we get: «`(?=\b\w{6}\b)\b\w*cat\w*\b`» . Easy! Here's how this works. At each character position in the string where the regex is attempted, the engine will first attempt the regex inside the positive lookahead. This sub-regex, and therefore the lookahead, matches only when the current character position in the string is at the start of a 6-letter word in the string. If not, the lookahead will fail, and the engine will continue trying the regex from the start at the next character position in the string.

The lookahead is zero-width. So when the regex inside the lookahead has found the 6-letter word, the current position in the string is still at the beginning of the 6-letter word. At this position will the regex engine attempt the remainder of the regex. Because we already know that a 6-letter word can be matched at the current position, we know that «`\b`» matches and that the first «`\w*`» will match 6 times. The engine will then backtrack, reducing the number of characters matched by «`\w*`», until «`cat`» can be matched. If «`cat`» cannot be matched, the engine has no other choice but to restart at the beginning of the regex, at the next character position in the string. This is at the second letter in the 6-letter word we just found, where the lookahead will fail, causing the engine to advance character by character until the next 6-letter word.

If «`cat`» can be successfully matched, the second «`\w*`» will consume the remaining letters, if any, in the 6-letter word. After that, the last «`\b`» in the regex is guaranteed to match where the second «`\b`» inside the lookahead matched. Our double-requirement-regex has matched successfully.

Optimizing Our Solution

While the above regex works just fine, it is not the most optimal solution. This is not a problem if you are just doing a search in a text editor. But optimizing things is a good idea if this regex will be used repeatedly and/or on large chunks of data in an application you are developing.

You can discover these optimizations by yourself if you carefully examine the regex and follow how the regex engine applies it, as I did above. I said the third and last «`\b`» are guaranteed to match. Since it is zero-width, and therefore does not change the result returned by the regex engine, we can remove them, leaving: «`(?=\b\w{6}\b)\w*cat\w*`». Though the last «`\w*`» is also guaranteed to match, we cannot remove it because it adds characters to the regex match. Remember that the lookahead discards its match, so it does not contribute to the match returned by the regex engine. If we omitted the «`\w*`», the resulting match would be the start of a 6-letter word containing "cat", up to and including "cat", instead of the entire word.

But we can optimize the first «`\w*`». As it stands, it will match 6 letters and then backtrack. But we know that in a successful match, there can never be more than 3 letters before "cat". So we can optimize this to «`\w{0,3}`». Note that making the asterisk lazy would not have optimized this sufficiently. The lazy asterisk would find a successful match sooner, but if a 6-letter word does not contain "cat", it would still cause the regex engine to try matching "cat" at the last two letters, at the last single letter, and even at one character beyond the 6-letter word.

So we have «`(?=\b\w{6}\b)\w{0,3}cat\w*`». One last, minor, optimization involves the first «`\b`». Since it is zero-width itself, there's no need to put it inside the lookahead. So the final regex is: «`\b(?=\w{6}\b)\w{0,3}cat\w*`».

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.

18. Continuing at The End of The Previous Match

The anchor «\G» matches at the position where the previous match ended. During the first match attempt, «\G» matches at the start of the string in the way «\A» does.

Applying «\G\w» to the string "test string" matches „t". Applying it again matches „e". The 3rd attempt yields „s" and the 4th attempt matches the second „t" in the string. The fifth attempt fails. During the fifth attempt, the only place in the string where «\G» matches is after the second t. But that position is not followed by a word character, so the match fails.

End of The Previous Match vs Start of The Match Attempt

With some regex flavors or tools, «\G» matches at the start of the match attempt, rather than at the end of the previous match result. This is the case with EditPad Pro, where «\G» matches at the position of the text cursor, rather than the end of the previous match. When a match is found, EditPad Pro will select the match, and move the text cursor to the end of the match. The result is that «\G» matches at the end of the previous match result only when you do not move the text cursor between two searches. All in all, this makes a lot of sense in the context of a text editor.

\G Magic with Perl

In Perl, the position where the last match ended is a "magical" value that is remembered separately for each string variable. The position is not associated with any regular expression. This means that you can use «\G» to make a regex continue in a subject string where another regex left off.

If a match attempt fails, the stored position for «\G» is reset to the start of the string. To avoid this, specify the continuation modifier /c.

All this is very useful to make several regular expressions work together. E.g. you could parse an HTML file in the following fashion:

```
while ($string =~ m/</g) {
  if ($string =~ m/\GB>/c) {
    # Bold
  } elsif ($string =~ m/\GI>/c) {
    # Italics
  } else {
    # ...etc...
  }
}
```

The regex in the while loop searches for the tag's opening bracket, and the regexes inside the loop check which tag we found. This way you can parse the tags in the file in the order they appear in the file, without having to write a single big regex that matches all tags you are interested in.

\G in Other Programming Langauges

This flexibility is not available with most other programming languages. E.g. in Java, the position for «`\G`» is remembered by the Matcher object. The Matcher is strictly associated with a single regular expression and a single subject string. What you can do though is to add a line of code to make the match attempt of the second Matcher start where the match of the first Matcher ended. «`\G`» will then match at this position.

19. If-Then-Else Conditionals in Regular Expressions

A special construct «`(?ifthen|else)`» allows you to create conditional regular expressions. If the *if* part evaluates to true, then the regex engine will attempt to match the *then* part. Otherwise, the *else* part is attempted instead. The syntax consists of a pair of round brackets. The opening bracket must be followed by a question mark, immediately followed by the *if* part, immediately followed by the *then* part. This part can be followed by a vertical bar and the *else* part. You may omit the *else* part, and the vertical bar with it.

For the *if* part, you can use the lookahead and lookbehind constructs. Using positive lookahead, the syntax becomes «`(?(?=regex)then|else)`». Because the lookahead has its own parentheses, the *if* and *then* parts are clearly separated.

Remember that the lookaround constructs do not consume any characters. If you use a lookahead as the *if* part, then the regex engine will attempt to match the *then* or *else part* (depending on the outcome of the lookahead) at the same position where the *if* was attempted.

Alternatively, you can check in the *if* part whether a capturing group has taken part in the match thus far. Place the number of the capturing group inside round brackets, and use that as the if part. Note that although the syntax for a conditional check on a backreference is the same as a number inside a capturing groups, no capturing groups is created. The number and the brackets are part of the if-then-else syntax started with «`(?`».

The regex «`(a)?b(?(1)c|d)`» matches „`bd`" and „`abc`". If the a can be matched, the first capturing group takes part in the match. The conditional then uses the *then* part «`c`». Otherwise, the *else* part «`d`» is used.

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.

Regex Flavors

Conditionals are supported by PCRE, the PHP preg functions and the .NET framework. The latest versions of EditPad Pro, PowerGREP and RegexBuddy also support them.

The .NET framework allows you to use the name of a named capturing group for the *if* test, e.g.: «`(?<test>a)?b(?(test)c|d)`». None of the other flavors, though they all support named capture, allow the name of a group as a conditional test. You have to use the number.

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 if the second capturing group took part in the match thus far. It will have if the header is the From or To header. In that case, we 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.

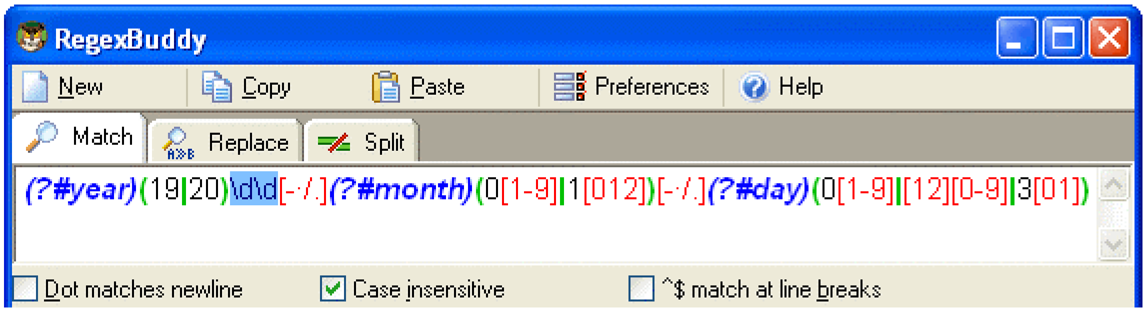
If the second capturing group did not participate in the match this far, the *else* part «`.+`» is attempted instead.

Finally, we place an extra pair of round brackets around the conditional. This captures the contents of the email header matched by the conditional into the third backreference. The conditional itself does not capture anything.

20. Adding Comments to Regular Expressions

If you have worked through the entire tutorial, I guess you will agree that regular expressions can quickly become rather cryptic. Therefore, many modern regex flavors allow you to insert comments into regexes. The syntax is «`(?#comment)`» where “comment” can be whatever you want, as long as it does not contain a closing round bracket. The regex engine ignores everything after the «`(?#`» until the first closing round bracket.

E.g. I could clarify the regex to match a valid date by writing it as «`(?#year)(19|20)\d\d[- /.](?#month)(0[1-9]|1[012])[- /.](?#day)(0[1-9]|[12][0-9]|3[01])`» . Now it is instantly obvious that this regex matches a date in yyyy-mm-dd format. Some software, such as RegexBuddy, EditPad Pro and PowerGREP can apply syntax coloring to regular expressions while you write them. That makes the comments really stand out, enabling the right comment in the right spot to make a complex regular expression much easier to understand.



```
(?#year)(19|20)\d\d[- /.](?#month)(0[1-9]|1[012])[- /.](?#day)(0[1-9]|[12][0-9]|3[01])
```

☐ Dot matches newline ☑ Case insensitive ☐ ^$ match at line breaks

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 web site), 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[^>]*>(.*?)</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]*)[^>]*>(.*?)</\1>`» will match the opening and closing pair of any HTML tag. Be sure to turn off case sensitivity. The key in this solution is the use of the backreference «`\1`» in the regex. Anything between the tags is captured into the second backreference. This solution will also not match tags nested in themselves.

Trimming Whitespace

You can easily trim unnecessary whitespace from the start and the end of a string or the lines in a text file by doing a regex search-and-replace. Search for «`^[ \t]+`» and replace with nothing to delete leading whitespace (spaces and tabs). Search for «`[ \t]+$`» to trim trailing whitespace. Do both by combining the regular expressions into «`^[ \t]+|[ \t]+$`» . Instead of `[ \t]` which matches a space or a tab, you can expand the character class into «`[ \t\r\n]`» if you also want to strip line breaks. Or you can use the shorthand «`\s`» instead.

IP Addresses

Matching an IP address is another good example of a trade-off between regex complexity and exactness. «`\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b`» will match any IP address just fine, but will also match

„`999.999.999.999`" as if it were a valid IP address. Whether this is a problem depends on the files or data you intend to apply the regex to. To restrict all 4 numbers in the IP address to 0..255, you can use this complex beast: «`\b(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.`»«`(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.`»«`(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.`»«`(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b`» (everything on a single line). The long regex stores each of the 4 numbers of the IP address into a capturing group. You can use these groups to further process the IP number.

If you don't need access to the individual numbers, you can shorten the regex with a quantifier to: «`\b(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}`»«`(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b`» . Similarly, you can shorten the quick regex to «`\b(?:\d{1,3}\.){3}\d{1,3}\b`»

More Detailed Examples

Matching a Floating Point Number. Also illustrates the common mistake of making everything in a regular expression optional.

Matching Valid Dates. A regular expression that matches 31-12-1999 but not 31-13-1999.

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.

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.

2. Matching Floating Point Numbers with a Regular Expression

In this example, I will show you how you can avoid a common mistake often made by people inexperienced with regular expressions. As an example, we will try to build a regular expression that can match any floating point number. Our regex should also match integers, and floating point numbers where the integer part is not given (i.e. zero). We will not try to match numbers with an exponent, such as 1.5e8 (150 million in scientific notation).

At first thought, the following regex seems to do the trick: «`[-+]?[0-9]*\.?[0-9]*`». This defines a floating point number as an optional sign, followed by an optional series of digits (integer part), followed by an optional dot, followed by another optional series of digits (fraction part).

Spelling out the regex in words makes it obvious: everything in this regular expression is optional. This regular expression will consider a sign by itself or a dot by itself as a valid floating point number. In fact, it will even consider an empty string as a valid floating point number. This regular expression can cause serious trouble if it is used in a scripting language like Perl or PHP to verify user input.

Not escaping the dot is also a common mistake. A dot that is not escaped will match any character, including a dot. If we had not escaped the dot, "`4.4`" would be considered a floating point number, and "`4X4`" too.

When creating a regular expression, it is more important to consider what it should *not* match, than what it should. The above regex will indeed match a proper floating point number, because the regex engine is greedy. But it will also match many things we do not want, which we have to exclude.

Here is a better attempt: «`[-+]?([0-9]*\.[0-9]+|[0-9]+)`». This regular expression will match an optional sign, that is either followed by zero or more digits followed by a dot and one or more digits (a floating point number with optional integer part), or followed by one or more digits (an integer).

This is a far better definition. Any match will include at least one digit, because there is no way around the «`[0-9]+`» part. We have successfully excluded the matches we do not want: those without digits.

We can optimize this regular expression as: «`[-+]?([0-9]*\.)?[0-9]+`» .

If you also want to match numbers with exponents, you can use: «`[-+]?([0-9]*\.)?[0-9]+([eE][-+]?[0-9]+)?`» . Notice how I made the entire exponent part optional by grouping it together, rather than making each element in the exponent optional.

3. Matching a Valid Date

«(19|20)\d\d[- /.](0[1-9]|1[012])[- /.](0[1-9]|[12][0-9]|3[01])» matches a date in yyyy-mm-dd format from between 1900-01-01 and 2099-12-31, with a choice of four separators. The year is matched by «(19|20)\d\d». I used alternation to allow the first two digits to be 19 or 20. The round brackets are mandatory. Had I omitted them, the regex engine would go looking for 19 or the remainder of the regular expression, which matches a date between 2000-01-01 and 2099-12-31. Round brackets are the only way to stop the vertical bar from splitting up the entire regular expression into two options.

The month is matched by «0[1-9]|1[012]», again enclosed by round brackets to keep the two options together. By using character classes, the first option matches a number between 01 and 09, and the second matches 10, 11 or 12.

The last part of the regex consists of three options. The first matches the numbers 01 through 09, the second 10 through 29, and the third matches 30 or 31.

Smart use of alternation allows us to exclude invalid dates such as 2000-00-00 that could not have been excluded without using alternation. To be really perfectionist, you would have to split up the month into various options to take into account the length of the month. The above regex still matches 2003-02-31, which is not a valid date. Making leading zeros optional could be another enhancement.

If you want to require the delimiters to be consistent, you could use a backreference. «(19|20)\d\d([- /.])(0[1-9]|1[012])\2(0[1-9]|[12][0-9]|3[01])» will match „1999-01-01" but not "1999/01-01".

Again, how complex you want to make your regular expression depends on the data you are using it on, and how big a problem it is if an unwanted match slips through. If you are validating the user's input of a date in a script, it is probably easier to do certain checks outside of the regex. For example, excluding February 29th when the year is not a leap year is far easier to do in a scripting language. It is far easier to check if a year is divisible by 4 (and not divisible by 100 unless divisible by 400) using simple arithmetic than using regular expressions.

Here is how you could check a valid date in Perl. Note that I added anchors to make sure the entire variable is a date, and not a piece of text containing a date. I also added round brackets to capture the year into a backreference.

```perl
sub isvaliddate {
  my $input = shift;
  if ($input =~ m!^((?:19|20)\d\d)[- /.](0[1-9]|1[012])[- /.](0[1-9]|[12][0-9]|3[01])$!) {
    # At this point, $1 holds the year, $2 the month and $3 the day of the date entered
    if ($3 == 31 and ($2 == 4 or $2 == 6 or $2 == 9 or $2 == 11)) {
      return 0; # 31st of a month with 30 days
    } elsif ($3 >= 30 and $2 == 2) {
      return 0; # February 30th or 31st
    } elsif ($2 == 2 and $3 == 29 and not ($1 % 4 == 0 and ($1 % 100 <> 0 or $1 % 400 == 0))) {
      return 0; # February 29th outside a leap year
    } else {
      return 1; # Valid date
    }
  } else {
    return 0; # Not a date
  }
}
```

To match a date in mm/dd/yyyy format, rearrange the regular expression to «(0[1-9]|1[012])[- /.](0[1-9]|[12][0-9]|3[01])[- /.](19|20)\d\d» . For dd-mm-yyyy format, use «(0[1-9]|[12][0-9]|3[01])[- /.](0[1-9]|1[012])[- /.](19|20)\d\d» .

4. Matching Whole Lines of Text

Often, you want to match complete lines in a text file rather than just the part of the line that satisfies a certain requirement. This is useful if you want to delete entire lines in a search-and-replace in a text editor, or collect entire lines in an information retrieval tool. To keep this example simple, let's say we want to match lines containing the word “John”. The regex «`John`» makes it easy enough to locate those lines. But the software will only indicate „`John`” as the match, not the entire line containing the word.

The solution is fairly simple. To specify that we need an entire line, we will use the caret and dollar sign and turn on the option to make them match at embedded newlines. In software aimed at working with text files like EditPad Pro and PowerGREP, the anchors always match at embedded newlines. To match the parts of the line before and after the match of our original regular expression «`John`», we simply use the dot and the star. Be sure to turn *off* the option for the dot to match newlines.

The resulting regex is: «`^.*John.*$`». You can use the same method to expand the match of any regular expression to an entire line, or a block of complete lines. In some cases, such as when using alternation, you will need to group the original regex together using round brackets.

Finding Lines Containing or Not Containing Certain Words

If a line can meet any out of series of requirements, simply use alternation in the regular expression. «`^.*\b(one|two|three)\b.*$`» matches a complete line of text that contains any of the words “one”, “two” or “three”. The first backreference will contain the word the line actually contains. If it contains more than one of the words, then the last (rightmost) word will be captured into the first backreference. This is because the star is greedy. If we make the first star lazy, like in «`^.*?\b(one|two|three)\b.*$`», then the backreference will contain the first (leftmost) word.

If a line must satisfy all of multiple requirements, we need to use lookahead. «`^(?=.*?\bone\b)(?=.*?\btwo\b)(?=.*?\bthree\b).*$`» matches a complete line of text that contains *all* of the words “one”, “two” and “three”. Again, the anchors must match at the start and end of a line and the dot must not match line breaks. Because of the caret, and the fact that lookahead is zero-width, all of the three lookaheads are attempted at the start of the each line. Each lookahead will match any piece of text on a single line («`.*?`») followed by one of the words. All three must match successfully for the entire regex to match. Note that instead of words like «`\bword\b`», you can put any regular expression, no matter how complex, inside the lookahead. Finally, «`.*$`» causes the regex to actually match the line, after the lookaheads have determined it meets the requirements.

If your condition is that a line should *not* contain something, use negative lookahead. «`^((?!regexp).)*$`» matches a complete line that does *not* match «`regexp`». Notice that unlike before, when using positive lookahead, I repeated both the negative lookahead and the dot together. For the positive lookahead, we only need to find one location where it can match. But the negative lookahead must be tested at each and every character position in the line. We must test that «`regexp`» fails everywhere, not just somewhere.

Finally, you can combine multiple positive and negative requirements as follows: «`^(?=.*?\bmust-have\b)(?=.*?\bmandatory\b)((?!avoid|illegal).)*$`» . When checking multiple positive requirements, the «`.*`» at the end of the regular expression full of zero-width assertions made sure that we actually matched something. Since the negative requirement must match the entire line, it is easy to replace the «`.*`» with the negative test.

5. Deleting Duplicate Lines From a File

If you have a file in which all lines are sorted (alphabetically or otherwise), you can easily delete (subsequent) duplicate lines. Simply open the file in your favorite text editor, and do a search-and-replace searching for «`^(.*)(\r?\n\1)+$`» » matches a single-line string that does not allow the quote character to appear inside the string. Using the negated character class is more efficient than using a lazy dot. «`"[^"]*"`» allows the string to span across multiple lines.

«`"[^"\\\r\n]*(\\.[^"\\\r\n]*)*"`» matches a single-line string in which the quote character can appear if it is escaped by a backslash. Though this regular expression may seem more complicated than it needs to be, it is much faster than simpler solutions which can cause a whole lot of backtracking in case a double quote appears somewhere all by itself rather than part of a string. «`"[^"\\]*(\\.[^"\\]*)*"`» 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]*)*"`».

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.

Part 6

Regular Expression Reference

1. Basic Syntax Reference

Characters

Character: Any character except `[\^$.|?*+()`
Description: All characters except the listed special characters match a single instance of themselves.
Example: «`a`» matches „a”

Character: `\` (backslash) followed by any of `[\^$.|?*+()`
Description: A backslash escapes special characters to suppress their special meaning.
Example: «`\+`» matches „+”

Character: `\xFF` where FF are 2 hexadecimal digits
Description: Matches the character with the specified ASCII/ANSI value, which depends on the code page used. Can be used in character classes.
Example: «`\xA9`» matches „©” when using the Latin-1 code page.

Character: `\n`, `\r` and `\t`
Description: Match an LF character, CR character and a tab character respectively. Can be used in character classes.
Example: «`\r\n`» matches a DOS/Windows CRLF line break.

Character Classes or Character Sets [abc]

Character: `[` (opening square bracket)
Description: Starts a character class. A character class matches a single character out of all the possibilities offered by the character class. Inside a character class, different rules apply. The rules in this section are only valid inside character classes. The rules outside this section are not valid in character classes, except `\n`, `\r`, `\t` and `\xFF`
Character: Any character except `^-]\` add that character to the possible matches for the character class.
Description: All characters except the listed special characters.
Example: «`[abc]`» matches „a”, „b” or „c”

Character: `\` (backslash) followed by any of `^-]\`
Description: A backslash escapes special characters to suppress their special meaning.
Example: «`[\^\]]`» matches „^” or „]”

Character: `-` (hyphen) except immediately after the opening `[`
Description: Specifies a range of characters. (Specifies a hyphen if placed immediately after the opening `[`)
Example: «`[a-zA-Z0-9]`» matches any letter or digit

Character: `^` (caret) immediately after the opening `[`
Description: Negates the character class, causing it to match a single character *not* listed in the character class. (Specifies a caret if placed anywhere except after the opening `[`)
Example: «`[^a-d]`» matches „x” (any character except a, b, c or d)

Character:	`\d`, `\w` and `\s`
Description:	Shorthand character classes matching digits 0-9, word characters (letters and digits) and whitespace respectively. Can be used inside and outside character classes
Example:	«`[\d\s]`» matches a character that is a digit or whitespace

Character:	`\D`, `\W` and `\S`
Description:	Negated versions of the above. Should be used only outside character classes. (Can be used inside, but that is confusing).)
Example:	«`\D`» matches a character that is not a digit

Dot

Character:	`.` (dot)
Description:	Matches any single character except line break characters `\r` and `\n`. Most regex flavors have an option to make the dot match line break characters too.
Example:	«`.`» matches „`x`” or (almost) any other character

Anchors

Character:	`^` (caret)
Description:	Matches at the start of the string the regex pattern is applied to. Matches a position rather than a character. Most regex flavors have an option to make the caret match after line breaks (i.e. at the start of a line in a file) as well.
Example:	«`^.`» matches „`a`” in “`abc\ndef`”. Also matches „`d`” in "multi-line" mode.

Character:	`$` (dollar)
Description:	Matches at the end of the string the regex pattern is applied to. Matches a position rather than a character. Most regex flavors have an option to make the dollar match before line breaks (i.e. at the end of a line in a file) as well. Also matches before the very last line break if the string ends with a line break.
Example:	«`.$`» matches „`f`” in “`abc\ndef`”. Also matches „`c`” in "multi-line" mode.

Character:	`\A`
Description:	Matches at the start of the string the regex pattern is applied to. Matches a position rather than a character. Never matches after line breaks.
Example:	«`\A.`» matches „`a`” in “`abc`”

Character:	`\Z`
Description:	Matches at the end of the string the regex pattern is applied to. Matches a position rather than a character. Never matches before line breaks, except for the very last line break if the string ends with a line break.
Example:	«`.\Z`» matches „`f`” in “`abc\ndef`”

Character:	`\z`
Description:	Matches at the end of the string the regex pattern is applied to. Matches a position rather than a character. Never matches before line breaks.
Example:	«`.\z`» matches „`f`” in “`abc\ndef`”

Word Boundaries

Character:	`\b`
Description:	Matches at the position between a word character (anything matched by «`\w`») and a non-word character (anything matched by «`[^\w]`» or «`\W`») as well as at the start and/or end of the string if the first and/or last characters in the string are word characters.
Example:	«`.\b`» matches „`c`” in “`abc`”

Character:	`\B`
Description:	Matches at the position between two word characters (i.e the position between «`\w\w`») as well as at the position between two non-word characters (i.e. «`\W\W`»).
Example:	«`\B.\B`» matches „`b`” in “`abc`”

Alternation

Character:	`|` (pipe)
Description:	Causes the regex engine to match either the part on the left side, or the part on the right side. Can be strung together into a series of options.
Example:	«`abc|def|xyz`» matches „`abc`”, „`def`” or „`xyz`”

Character:	`|` (pipe)
Description:	The pipe has the lowest precedence of all operators. Use grouping to alternate only part of the regular expression.
Example:	«`abc(def|xyz)`» matches „`abcdef`” or „`abcxyz`”

Quantifiers

Character:	`?` (question mark)
Description:	Makes the preceding item optional. Greedy, so the optional item is included in the match if possible.
Example:	«`abc?`» matches „`ab`” or „`abc`”

Character:	`??`
Description:	Makes the preceding item optional. Lazy, so the optional item is excluded in the match if possible. This construct is often excluded from documentation because of its limited use.
Example:	«`abc??`» matches „`ab`” or „`abc`”

Character:	`*` (star)
Description:	Repeats the previous item zero or more times. Greedy, so as many items as possible will be matched before trying permutations with less matches of the preceding item, up to the point where the preceding item is not matched at all.
Example:	«`".*"`» matches „`"def" "ghi"`” in “`abc "def" "ghi" jkl`”

Character:	`*?` (lazy star)
Description:	Repeats the previous item zero or more times. Lazy, so the engine first attempts to skip the previous item, before trying permutations with ever increasing matches of the preceding item.
Example:	«`".*?"`» matches „`"def"`" in "`abc "def" "ghi" jkl`"
Character:	`+` (plus)
Description:	Repeats the previous item once or more. Greedy, so as many items as possible will be matched before trying permutations with less matches of the preceding item, up to the point where the preceding item is matched only once.
Example:	«`".+"`» matches „`"def" "ghi"`" in "`abc "def" "ghi" jkl`"
Character:	`+?` (lazy plus)
Description:	Repeats the previous item once or more. Lazy, so the engine first matches the previous item only once, before trying permutations with ever increasing matches of the preceding item.
Example:	«`".+?"`» matches „`"def"`" in "`abc "def" "ghi" jkl`"
Character:	`{n}` where n is an integer >= 1
Description:	Repeats the previous item exactly n times.
Example:	«`a{3}`» matches „`aaa`"
Character:	`{n,m}` where n >= 1 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 „`aa`", „`aaa`" or „`aaaa`"
Character:	`{n,m}?` where n >= 1 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 „`aaaa`", „`aaa`" or „`aa`"
Character:	`{n,}` where n >= 1
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`"
Character:	`{n,}?` where n >= 1
Description:	Repeats the previous item between n and m 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`"

2. Advanced Syntax Reference

Grouping and Backreferences

Character:	`(regex)`
Description:	Round brackets group the regex between them. They capture the text matched by the regex inside them that can be reused in a backreference, and they allow you to apply regex operators to the entire grouped regex.
Example:	«`(abc){3}`» matches „`abcabcabc`”. First group matches „`abc`”.
Character:	`(?:regex)`
Description:	Non-capturing parentheses group the regex so you can apply regex operators, but do not capture anything and do not create backreferences.
Example:	«`(?:abc){3}`» matches „`abcabcabc`”. No groups.
Character:	`\1` through `\9`
Description:	Substituted with the text matched between the 1st through 9th pair of capturing parentheses. Some regex flavors allow more than 9 backreferences.
Example:	«`(abc|def)=\1`» matches „`abc=abc`” or „`def=def`”, but not “`abc=def`” or “`def=abc`”.

Modifiers

Character:	`(?i)`
Description:	Turn on case insensitivity for the remainder of the regular expression. (Older regex flavors may turn it on for the entire regex.)
Example:	«`te(?i)st`» matches „`teST`” but not “`TEST`”.
Character:	`(?-i)`
Description:	Turn off case insensitivity for the remainder of the regular expression.
Example:	«`(?i)te(?-i)st`» matches „`TEst`” but not “`TEST`”.

Character:	`(?s)`
Description:	Turn on "dot matches newline" for the remainder of the regular expression. (Older regex flavors may turn it on for the entire regex.)
Character:	`(?-s)`
Description:	Turn off "dot matches newline" for the remainder of the regular expression.
Character:	`(?m)`
Description:	Caret and dollar match after and before newlines for the remainder of the regular expression. (Older regex flavors may apply this to the entire regex.)
Character:	`(?-m)`
Description:	Caret and dollar only match at the start and end of the string for the remainder of the regular expression.
Character:	`(?i-sm)`
Description:	Turns on the options "i" and "m", and turns off "s" for the remainder of the regular expression. (Older regex flavors may apply this to the entire regex.)
Character:	`(?i-sm:regex)`
Description:	Matches the regex inside the span with the options "i" and "m" turned on, and "s" turned off.
Example:	«`(?i:te)st`» matches „`TEst`" but not "`TEST`".

Atomic Grouping and Possessive Quantifiers

Character:	`(?>regex)`
Description:	Atomic groups prevent the regex engine from backtracking back into the group (forcing the group to discard part of its match) after a match has been found for the group. Backtracking can occur inside the group before it has matched completely, and the engine can backtrack past the entire group, discarding its match entirely. Eliminating needless backtracking provides a speed increase. Atomic grouping is often indispensable when nesting quantifiers to prevent a catastrophic amount of backtracking as the engine needlessly tries pointless permutations of the nested quantifiers.
Example:	«`x(?>\w+)x`» is more efficient than «`x\w+x`» if the second x cannot be matched.

Character:	`?+`, `*+`, `++` and `{m,n}+`
Description:	Possessive quantifiers are a limited yet syntactically cleaner alternative to atomic grouping. Only available in a few regex flavors. They behave as normal greedy quantifiers, except that they will not give up part of their match for backtracking.
Example:	«`x++`» is identical to «`(?>x+)`»

Lookaround

Character:	`(?=regex)`
Description:	Zero-width positive lookahead. Matches at a position where the pattern inside the lookahead can be matched. Matches only the position. It does not consume any characters or expand the match. In a pattern like «`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`".

Character:	`(?!regex)`
Description:	Zero-width negative lookahead. Identical to positive lookahead, except that the overall match will only succeed if the regex inside the lookahead fails to match.
Example:	«`t(?!s)`» matches the first „`t`” in „`streets`”.

Character:	`(?<=text)`
Description:	Zero-width positive lookbehind. Matches at a position to the left of which text appears. Since regular expressions cannot be applied backwards, the test inside the lookbehind can only be plain text. Some regex flavors allow alternation of plain text options in the lookbehind.
Example:	«`(?<=s)t`» matches the first „`t`” in „`streets`”.

Character:	`(?<!text)`
Description:	Zero-width negative lookbehind. Matches at a position if the text does not appear to the left of that position.
Example:	«`(?<!s)t`» matches the second „`t`” in „`streets`”.

Continuing from The Previous Match

Character:	`\G`
Description:	Matches at the position where the previous match ended, or the position where the current match attempt started (depending on the tool or regex flavor). Matches at the start of the string during the first match attempt.
Example:	«`\G[a-z]`» first matches „`a`”, then matches „`b`” and then fails to match in “`ab_cd`”.

Conditionals

Character:	`(?(?=regex)then|else)`
Description:	If the lookahead succeeds, the "then" part must match for the overall regex to match. If the lookahead fails, the "else" part must match for the overall regex to match. Not just positive lookahead, but all four lookarounds can be used. Note that the lookahead is zero-width, so the "then" and "else" parts need to match and consume the part of the text matched by the lookahead as well.
Example:	«`(?(?<=a)b|c)`» matches the second „`b`” and the first „`c`” in “`babxcac`”

Comments

Character:	`(?#comment)`
Description:	Everything between `(?#` and `)` is ignored by the regex engine.
Example:	«`a(?#foobar)b`» matches „`ab`”

3. Syntax Reference for Specific Regex Flavors

.NET Syntax for Named Capture and Backreferences

Character:	`(?<name>regex)`
Description:	Round brackets group the regex between them. They capture the text matched by the regex inside them that can be referenced by the name between the sharp brackets. The name may consist of letters and digits.
Character:	`(?'name'regex)`
Description:	Round brackets group the regex between them. They capture the text matched by the regex inside them that can be referenced by the name between the single quotes. The name may consist of letters and digits.
Character:	`\k<name>`
Description:	Substituted with the text matched by the capturing group with the given name.
Example:	«`(?<group>abc|def)=\k<group>`» matches „`abc=abc`” or „`def=def`”, but not “`abc=def`” or “`def=abc`”.

Character:	`\k'name'`
Description:	Substituted with the text matched by the capturing group with the given name.
Example:	«`(?'group'abc|def)=\k'group'`» matches „`abc=abc`” or „`def=def`”, but not “`abc=def`” or “`def=abc`”.

Python Syntax for Named Capture and Backreferences

Character:	`(?P<name>regex)`
Description:	Round brackets group the regex between them. They capture the text matched by the regex inside them that can be referenced by the name between the sharp brackets. The name may consist of letters and digits.
Character:	`(?P=name)`
Description:	Substituted with the text matched by the capturing group with the given name. Not a group, despite the syntax using round brackets.
Example:	«`(?P<group>abc|def)=(?P=group)`» matches „`abc=abc`” or „`def=def`”, but not “`abc=def`” or “`def=abc`”.

4. Unicode Syntax Reference

Unicode Characters

Character:	`\X`
Description:	Matches a single Unicode grapheme, whether encoded as a single code point or multiple code points using combining marks. A grapheme most closely resembles the everyday concept of a "character".
Example:	«`\X`» matches „à" encoded as U+0061 U+0300, „à" encoded as U+00E0, „©", etc.

Character:	`\uFFFF` where FFFF are 4 hexadecimal digits
Description:	Matches a specific Unicode code point. Can be used inside character classes.
Example:	«`\u00E0`» matches „à" encoded as U+00E0 only. «`\u00A9`» matches „©"

Unicode Properties

Character:	`\p{L}` or `\p{Letter}`
Description:	Matches a single Unicode code point that has the property "letter". See Unicode Character Properties in the tutorial for a complete list of properties. Each Unicode code point has exactly one property. Can be used inside character classes.
Example:	«`\p{L}`» matches „à" encoded as U+00E0; «`\p{S}`» matches „©"

Character:	`\P{L}` or `\P{Letter}`
Description:	Matches a single Unicode code point that does *not* have the property "letter". Can be used inside character classes.
Example:	«`\P{L}`» matches „©"

Index