



Version 4.14.1 — 15 March 2024

Published by Just Great Software Co. Ltd.

Copyright © 2004–2024 Jan Goyvaerts. All rights reserved.

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

Table of Contents

RegexBuddy Manual.....	1
1. Introducing RegexBuddy.....	3
2. Getting Started with RegexBuddy.....	6
3. Select Your Application or Programming Language	11
4. Define a Match, Replace, or Split Action	15
5. Set Regular Expression Options	18
6. Insert a Token into The Regular Expression	23
7. Insert a Regex Token to Match Specific Characters	25
8. Insert a Regex Token to Match Unicode Characters.....	30
9. Insert a Shorthand Character Class	36
10. Insert a Regex Token to Match One Character out of Many Possible Characters.....	38
11. Insert a Regex Token to Match One Character from Predefined POSIX Classes.....	41
12. Insert a Regex Token to Match at a Certain Position	43
13. Insert a Regex Token to Repeat Another Token.....	45
14. Insert a Regex Token to Match Different Alternatives.....	47
15. Insert a Capturing Group.....	48
16. Insert a Backreference into the Regular Expression.....	50
17. Insert a Token to Recurse into The Regex or a Capturing Group.....	51
18. Insert a Conditional into the Regular Expression.....	54
19. Insert a Grouping Regex Token	55
20. Insert Lookaround	57
21. Insert a Regex Token to Change a Matching Mode	58
22. Insert a Comment.....	60
23. Using RegexMagic with RegexBuddy.....	61
24. Insert a Replacement Text Token.....	62
25. Insert Specific Characters into The Replacement Text.....	63
26. Insert a Backreference into the Replacement Text.....	68
27. Insert a Conditional into the Replacement Text	71
28. Insert The Subject String into The Replacement Text.....	73
29. Analyze and Edit a Regular Expression.....	75
30. Compare a Regex Between Multiple (Versions of) Applications	78
31. Export The Analysis of a Regular Expression.....	81
32. Convert a Regex to Work with Another Application.....	82
33. Testing Regular Expression Actions	85
34. Debugging Regular Expressions	92
35. Comparing the Efficiency of Regular Expressions.....	95
36. Generate Source Code to Use a Regular Expression	101
37. Available Functions in Source Code Snippets	103
38. Copying and Pasting Regular Expressions	107
39. Storing Regular Expressions in Libraries.....	111
40. Parameterizing a Regular Expression Stored in a Library.....	113
41. Using a Regular Expression from a Library.....	114
42. GREP: Search and Replace through Files and Folders.....	116
43. History: Experiment with Multiple Regular Expressions	120
44. Share Experiences and Get Help on The User Forums	122

45. Forum RSS Feeds.....	126
46. Keyboard Shortcuts	127
47. Editing Text with The Keyboard.....	131
48. Adjust RegexBuddy to Your Preferences.....	133
49. Text Layout Configuration	139
50. Text Cursor Configuration.....	144
51. Integrate RegexBuddy with Searching, Editing and Coding Tools	147
52. Basic Integration with RegexBuddy	149
53. Integrating RegexBuddy Using COM Automation.....	154
54. Application Identifiers.....	165
55. Contact RegexBuddy's Developer and Publisher.....	187

Regular Expressions Tutorial..... 189

1. Regular Expressions Tutorial.....	191
2. Literal Characters.....	194
3. Non-Printable Characters.....	196
4. First Look at How a Regex Engine Works Internally	199
5. Character Classes or Character Sets.....	201
6. Character Class Subtraction	204
7. Character Class Intersection	206
8. Shorthand Character Classes	208
9. The Dot Matches (Almost) Any Character	210
10. Start of String and End of String Anchors.....	213
11. Word Boundaries.....	216
12. Alternation with The Vertical Bar or Pipe Symbol.....	219
13. Optional Items.....	221
14. Repetition with Star and Plus	222
15. Use Parentheses for Grouping and Capturing.....	225
16. Using Backreferences To Match The Same Text Again	226
17. Backreferences to Failed Groups.....	229
18. Named Capturing Groups and Backreferences.....	231
19. Relative Backreferences.....	234
20. Branch Reset Groups.....	235
21. Free-Spacing Regular Expressions.....	237
22. Unicode Regular Expressions.....	240
23. Specifying Modes Inside The Regular Expression.....	249
24. Atomic Grouping	251
25. Possessive Quantifiers	253
26. Lookahead and Lookbehind Zero-Length Assertions	256
27. Testing The Same Part of a String for More Than One Requirement	260
28. Keep The Text Matched So Far out of The Overall Regex Match.....	262
29. If-Then-Else Conditionals in Regular Expressions	264
30. Matching Nested Constructs with Balancing Groups.....	267
31. Regular Expression Recursion	271
32. Regular Expression Subroutines	273
33. Infinite Recursion.....	277
34. Quantifiers On Recursion.....	279
35. Subroutine Calls May or May Not Capture.....	281
36. Backreferences That Specify a Recursion Level.....	285

37. Recursion and Subroutine Calls May or May Not Be Atomic	288
38. POSIX Bracket Expressions	292
39. Zero-Length Regex Matches	295
40. Continuing at The End of The Previous Match.....	298
41. Replacement Strings Tutorial	300
42. Special Characters.....	301
43. Non-Printable Characters	303
44. Matched Text	304
45. Numbered and Named Backreferences.....	305
46. Match Context	307
47. Replacement Text Case Conversion.....	308
48. Replacement String Conditionals.....	310

Regular Expressions Examples 313

1. Sample Regular Expressions.....	315
2. Matching Numeric Ranges with a Regular Expression	317
3. Matching Floating Point Numbers with a Regular Expression	319
4. How to Find or Validate an Email Address.....	320
5. How to Find or Validate an IP Address	325
6. Matching a Valid Date	327
7. Replacing Numerical Dates with Textual Dates.....	329
8. Finding or Verifying Credit Card Numbers	331
9. Matching Whole Lines of Text.....	333
10. Deleting Duplicate Lines From a File.....	335
11. Example Regexes to Match Common Programming Language Constructs	336
12. Find Two Words Near Each Other	339
13. Runaway Regular Expressions: Catastrophic Backtracking.....	340
14. Runaway Regular Expressions: Too Many Repetitions	347
15. Preventing Regular Expression Denial of Service (ReDoS).....	350
16. Repeating a Capturing Group vs. Capturing a Repeated Group.....	354
17. Mixing Unicode and 8-bit Character Codes.....	356

Regular Expressions Tools & Programming Languages 359

1. Specialized Tools and Utilities for Working with Regular Expressions	361
2. C++ Regular Expressions with Boost	364
3. Delphi Regular Expressions Classes.....	366
4. EditPad Lite: Basic Text Editor with Full Regular Expression Support	370
5. EditPad Pro: Convenient Text Editor with Full Regular Expression Support	372
6. What Is grep?.....	375
7. GNU Regular Expression Extensions	377
8. Using Regular Expressions in Groovy	379
9. Using Regular Expressions in Java	381
10. Using Regular Expressions with JavaScript	384
11. MySQL Regular Expressions with The REGEXP Operator.....	387
12. Using Regular Expressions with Microsoft .NET	389
13. Oracle Database Regular Expressions	393

14. The PCRE Open Source Regex Library	396
15. The PCRE2 Open Source Regex Library.....	398
16. Perl’s Rich Support for Regular Expressions.....	402
17. PHP Provides Three Sets of Regular Expression Functions	404
18. POSIX Basic Regular Expressions	408
19. PostgreSQL Has Three Regular Expression Flavors	410
20. PowerGREP: Taking grep Beyond The Command Line	412
21. Regular Expressions with PowerShell.....	415
22. Python’s re Module	417
23. Regular Expressions with The R Language.....	421
24. RegexMagic: Regular Expression Generator	424
25. Using Regular Expressions with Ruby.....	426
26. C++ Regular Expressions with std::regex.....	428
27. Tcl Has Three Regular Expression Flavors	431
28. VBScript’s Regular Expression Support.....	435
29. How to Use Regular Expressions in Visual Basic.....	438
30. wxWidgets Supports Three Regular Expression Flavors.....	439
31. XML Schema Regular Expressions	443
32. XQuery and XPath Regular Expressions	445
33. XRegExp Regular Expression Library for JavaScript	447

Regular Expressions Reference 449

1. Regular Expressions Reference	451
2. Special and Non-Printable Characters	454
3. Basic Features.....	459
4. Character Classes	461
5. Shorthand Character Classes	466
6. Anchors	469
7. Word Boundaries.....	472
8. Quantifiers	474
9. Unicode Syntax Reference	479
10. Capturing Groups and Backreferences	482
11. Named Groups and Backreferences.....	486
12. Special Groups.....	490
13. Balancing Groups, Recursion, and Subroutines	494
14. Replacement String Characters	499
15. Matched Text and Backreferences in Replacement Strings.....	503
16. Context and Case Conversion in Replacement Strings.....	508
17. Conditionals in Replacement Strings.....	511

Part 1

RegexBuddy Manual

1. Introducing RegexBuddy

RegexBuddy is your perfect companion for working with regular expressions. Let me give you a short overview of some of the most important things you can achieve with RegexBuddy.

You can learn all there is to know about regular expressions today with RegexBuddy's detailed, step by step regular expressions tutorial. You can find this tutorial in the second part of this manual.

Learn each of the different elements that compose a regular expression, step by step in logical order. If you already have some experience with regular expressions, this logical separation enables you to brush up your knowledge on specific areas. When trying to understand a regex, you only need to click the Explain Token button, and RegexBuddy will present you the appropriate topic in the tutorial.

Regex Tree and Regex Building Blocks

RegexBuddy's regex building blocks make it much easier to define regular expressions. Instead of typing in regex tokens directly, you can just pick what you want from a descriptive menu. Use RegexBuddy's neatly organized tree of regex tokens to keep track of the pattern you have built so far.

When you need to edit a regular expression written by somebody else, or if you are just curious to understand or study a regex you encountered, copy and paste it into RegexBuddy. RegexBuddy's regex tree will give you a clear analysis of the regular expression. Click on the regular expression, or on the regex tree, to highlight corresponding parts. Collapse part of the tree to get a good overview of complex regular expressions.

You can create and edit regular expressions quickly and easily with RegexBuddy. You can mix manipulating RegexBuddy's building blocks and directly editing the regex pattern to suit your own skill and style. Rely on RegexBuddy as you rely on a buddy or coach to assist you.

Compare and Convert Regular Expressions Between Applications and Languages

There are many different implementations of regular expressions. A regular expression that works in one application or programming language may not work or work differently in another application or language, or even in another version of the same application or language.

If your regex needs to work in multiple applications or multiple versions of an application, tell RegexBuddy to compare your regex between those applications. You'll be alerted to any potential differences while you're creating your regular expression, so you don't waste any time testing a regular expression that won't work consistently, or run into unpleasant surprises later if your tests didn't expose the differences.

If you have a regex that works correctly with one application but now you need it to work with another application, tell RegexBuddy to convert your regex from one application to another. RegexBuddy will adjust the regex to the syntax expected by the target application and alert you if it may find different matches in the target application than the original regex would in the original application. This will allow you to quickly decide, with minimal testing, whether the converted regex will still work the way you want. It's also a great

way of using regular expressions that you have found on the Internet but that were intended for applications you're not familiar with.

Regex Tester and Debugger

You should not risk actual data with untested regexes. Copy and paste sample data into RegexBuddy, or open test files. You can step through the search matches in the sample data, and get a detailed report about each match. Or highlight all matches to debug the regex in real time as you edit it.

When you plan to use a regex in a search-and-replace operation, preview the search and replace in RegexBuddy. If you want to split a string using a regex, check the result in RegexBuddy. Avoid nasty surprises when using a regular expression to modify real data or files.

If a regex isn't working exactly the way you'd expect it to, invoke RegexBuddy's debugger to see exactly how the regular expression is applied. You will see which text is matched by each token, at every step during the matching process. You will know exactly why the regex works the way it does, and fix it without any guesswork.

Develop Efficient Software Quickly with Instant Code Snippets

You can save time and code efficiently by using regular expressions when developing applications and scripts. With the proper regex, you can often do in a single line of code, or a few lines of code, what would otherwise require dozens or hundreds.

Rely on RegexBuddy to handle the details, such as which classes and function calls to use, and how to escape special characters. Just select the language you are working with, and the action you want to perform. Test whether a string matches a regex, extract matches from a string, search and replace, split a string, etc. RegexBuddy knows all the common regex actions and how to perform them with a variety of programming languages: C#, VB.NET, Java, Perl, PHP, JavaScript and C/C++.

Your Own RegexBuddy Library

Build your own collection of handy regex patterns, and use them whenever you want to. You can easily browse through and instantly search through the regexes you collected. When you found the regex you want, click the Use button.

For common tasks, use one of the many regular expressions you can find in RegexBuddy's library of pre-created regular expressions. You will find readily useful regexes for a wide variety of tasks. For many tasks, there will be several choices of regex patterns, with the differences clearly described.

Share Experiences with Other Users

Share your experiences with other RegexBuddy users or get help on RegexBuddy's built-in user forums. Simply click the Login button for instant access. You can discuss any topic involving RegexBuddy or regular expressions in general.

2. Getting Started with RegexBuddy

This section provides a brief overview of what you can do with RegexBuddy. It won't really try to explain anything. RegexBuddy is quite straightforward to use, so you can jump right in. For more details, read the rest of Part 1 in the manual.

By default, RegexBuddy shows the regular expression and regex history at the top. The bottom area shows eight tabs: Create, Convert, Test, Debug, Use, Library, GREP and Forum. If you have a large monitor, you can arrange the tabs side by side in two groups as shown below. To do so, click on the View button in the toolbar. It's the third button from the right in the topmost toolbar. Select Side by Side Layout in the View menu. If you have two monitors, the Dual Monitor Side by Side layout gives you a maximum view. You can also rearrange the tabs manually by dragging and dropping them with the mouse. Panels can be tabbed, docked or floating. Toolbars can also be rearranged and made to float.

The screen shot below shows RegexBuddy in side by side layout in its full glory. Other screen shots in this manual will be smaller with most of the panels and toolbars hidden, to keep the file size and download time reasonable. Read on below the screen shot to learn how to create your first regular expression with RegexBuddy.

The screenshot displays the RegexBuddy application in a side-by-side layout. The main window is titled "RegexBuddy" and shows a regular expression in the top-left pane: `\A(?:[1-9]|1[0-2])\/(?:[0-9]|1[0-9]|2[0-9])\/(?:[0-9]|1[0-9]|2[0-9])\z(?:#Shiny·embossed·Logo)`. The top-right pane shows the "History" tab with "Regex 1". The middle-left pane shows the "Test" tab with a list of test cases, including "Valid" and "Invalid" dates. The middle-right pane shows the "Line by line" view of the test results, highlighting the first match: "1/1/01". The bottom panel shows a table of match results:

Match	Start	Length
Match 1 of 3:	1/1/01	8
Group 1:	1	1
Group 2:	1/1/01	10

If at First You Don't Succeed: Cheat

Everything is easier if you cheat, so we'll start with that. While RegexBuddy is designed to help you create and test regular expressions, and learn everything about them, it also comes with a handy library of regular expressions that you'll find useful in many situations. To access it, simply click on the Library panel. Click on a regex that interests you, push the Use button, and pick Use Regex and Test Subject.

RegexBuddy explains how this regular expression works on the Create panel. Each node in the tree corresponds with one elementary piece of the regular expression, called a token. If you click on a node, the corresponding token will be selected in the regular expression. Click the Explain Token button to open RegexBuddy's regular expressions tutorial at the page that explains the node you selected in the tree.

Reading through the whole tutorial from the first page to the last page can be quite overwhelming. Learning as you go by selecting regular expressions from the library and reading relevant parts through Explain Token is more pleasant.

Building Your First Regular Expression

The Create panel and the Test panel in RegexBuddy are two powerful tools to help you create regular expressions that match exactly what you want. If you have the screen space, it's a good idea to keep both visible in the side by side view.

The Create panel explains your regular expression in plain English. Yet, it does maintain a one-on-one relationship with the actual regular expression syntax. This way it helps you learn the actual syntax, rather than being a crutch you'll forever depend upon. As you become more comfortable with regular expressions, you'll start typing in more and more of your regular expressions directly rather than going via the Create panel and its Insert Token menu. But even as an expert, you'll still use the Create panel to help you analyze long regular expressions. Its tree structure is often easier to grasp than a long-winded linear regular expression.

The Test panel shows you what your regular expression actually does. It's a sandbox where you can test your regular expression, before mauling actual data.

Enough talk! Let's create our first regular expression to match a date in American mm/dd/yy format, with years from 00 to 99, and optional leading zeros for the day and month. Now read that sentence again. You may not realize it yet, but you'll soon learn through bitter experience that that rather long sentence is the most important step in crafting a regular expression that does exactly what you want. That is: knowing exactly what you want. If you don't know whether leading zeros should be optional or not, or if the year should have 2 or 4 digits, there's no hope for you. Don't launch RegexBuddy until you know what the job is.

Once you know the job, codify it by preparing test data. You can open a file, download a web page, or just type your samples directly into the Test panel. For this example, we'll enter multiple test subjects line by line. So start with choosing the "Line by line" option in the drop-down list on the Test toolbar. Then copy and paste the following lines:

```
Valid:  
1/1/01  
01/01/78  
12/31/99  
Invalid:
```

```

0/0/0
0/0/00
1/1/1
12/32/52
19/19/19
1/9/1999
1212/12/12
On 6/2/07 I wrote this
On 6/24/13 I edited this

```

Particularly the invalid examples are important. It's often much harder to "see" which undesired matches a regular expression will produce than it is to see that a regular expression will match everything you want.

Now, let's start crafting our regular expression. Begin with clicking the Clear History button in the History. It looks like a File|New icon. This makes sure we start with a clean slate. To get the same results as explained below, select "C# (.NET 2.0–4.5)" in the list of applications. If you select another application, you may get slightly different results.

The easiest way to create a regular expression, is to have your sample matches ready on the Test panel, and simply proceed from left to right. Regular expressions work with text, character by character. So we'll have to translate what we want, our date format, into a pattern of characters.

First up is the month, which consist of a digit 0 or 1, followed by a digit 0 through 9. The first digit is optional if the number is less than 10. Let's try this. Click the Insert Token button on the Create panel, and select Character Class. In the box "literal characters", type "01" (zero one, without the quotes), and click OK. We just created our very first regular expression: `[01]`. The Test panel immediately highlights all digits 0 and 1. Now, this first token has to be optional. So we click Insert Token again, and select Quantifier (repetition). Set the minimum to zero and the maximum to one. This essentially makes the character class token optional. Choose the "greedy" option, and click OK. RegexBuddy puts a question mark after the character class: `[01]?`. A question mark in a regular expression indeed makes the preceding token optional. To finish our month number, we insert another character class. Select Insert Token, Character Class and click the Clear button. Under "range of characters", type 0 in the left box, and 9 in the right. Click OK. The regex so far is `[01]?[0-9]`. The test panel highlights a whole bunch of numbers. (Can you spot our first mistake? More about that later.)

The date separator is easy: a literal slash. Click Insert Token, Literal Text, type a forward slash, and click OK. RegexBuddy appends a forward slash to your regex. Very clever. The highlighting on the Test panel changes dramatically. We've already progressed to the point where years are no longer matched as lonely digits.

Next up is the day. It consists of two numbers. An optional digit between 0 and 3, and a required digit between 0 and 9. We already know how to match a range of digits and how to make one optional. So just type `[0-3]?[0-9]` at the end of your regex. Adding the date separator while we're at it, we get: `[01]?[0-9]/[0-3]?[0-9]/`. Things are starting to shape up.

The year consists of two digits ranging from 0 to 9 each. You could just type in `[0-9][0-9]`. Or, you could type in `[0-9]` and get some practice inserting a quantifier that repeats the token twice. The result is then: `[01]?[0-9]/[0-3]?[0-9]/[0-9]{2}`.

All done! Our regex matches what we want. Copy it into the source code, compile, ship to customer, and wait for the bug reports to roll in.

The regular expression indeed matches the dates we want. But it also matches a bunch of stuff we don't want! How important is this? This is yet another thing that must be specified in the requirements. If you're parsing a computer-generated database export that you know will only contain valid dates, you could just use `[0-9]{2}/[0-9]{2}/[0-9]{2}` to grab all dates. No need to make your regex complicated to filter out 99/99/99, because the database can't store that no-date anyway. But if you're going to process user-provided data, you'd better case your regex in molded stainless steel with a shiny embossed logo.

The first problem is that our month and day parts allow too many numbers, like 0 and 19 for the month, and 0 and 32 for the day. Let's begin with the month. While the first digit is indeed an optional 0 or 1, and the second digit is always between 0 and 9, there's another restriction we didn't put into the regex: if the first digit is 1, then the second digit must be between 0 and 2. And if the first digit is 0 or missing, then the second digit can't be zero. So we essentially have two alternatives for the second digit, depending on what the first digit is. Let's do this.

First, delete the tokens for matching the month from the regex, leaving `/[0-3]?[0-9]/[0-9]{2}`. Put the cursor at the start of the regex. For the first alternative, we have an optional zero and a digit between 1 and 9. We already know how to do this, so just type: `0?[1-9]`. Now we need to tell RegexBuddy we want to add an alternative to what we just typed. This we do by selecting the Alternation item in the Insert Token menu. RegexBuddy will insert a vertical bar, also known as the pipe symbol. Now we type in the second alternative: `1[0-2]` matches 10, 11 and 12. Our regex is now `0?[1-9]|1[0-2]/[0-3]?[0-9]/[0-9]{2}`.

Unfortunately, that didn't quite go as planned. The Test panel now highlights individual digits all over the place. The Create panel tells us why: the vertical bar alternates the what's to the left of it with everything that's to the right of it. You can see that by clicking on "match this alternative" in the regex tree, and then on "or match this alternative" below. The first alternative is correct, but the second one should stop at the `/`.

To do this, we need to group the two alternatives for the month together. In the regular expression, select `0?[1-9]|1[0-2]`. Do this like you would select text in any text editor. Then click Insert Token, and select Numbered Capturing Group. We could have used a non-capturing group since we're not interested in capturing anything. However, non-capturing groups use a more complicated syntax than numbered capturing groups. You can try them if you want though. It won't make any difference in this example.

Now let's look at the Create panel again: the capturing group's node now sits on the same level in the tree as the two tokens that match the `/` literally. The two nodes for the alternatives for the date sit nice and cozy below the group node. If you click on them again, you'll see each alternative selects exactly the two alternatives we typed in three paragraphs ago.

We can use the exact same technique for the day of the month. If the first digit is zero or missing, we match `0?[1-9]`. If the first digit is a 1 or 2, we match `[12][0-9]`. If the first digit is a 3, we match `3[01]`. Put together in a group with alternation, we match the month with: `(0?[1-9]|1[0-2]|2[0-9]|3[01])`.

Our overall regex is now `(0?[1-9]|1[0-2]|2[0-9]|3[01])/[0-3]?[0-9]/[0-9]{2}`. Looking at it like this, you can see why even regex gurus find RegexBuddy's Create panel helpful. Even though you already know all the syntax used, the regex tree helps to analyze what's going on.

We're almost there. Everything highlighted on the Test panel is now a valid mm/dd/yy date. However, the regex is being sneaky and matching text that looks like a date from the middle of longer strings. There are two ways to go about this. When validating user input, you'll want to check that the input is nothing but a date. For that, we can use start-of-string and end-of-string anchors. Place the cursor at the start of the regex and click Insert Token, Anchors, Beginning of The String. Move the cursor to the end of the regex, and click

Insert Token, Anchors, End of The String. Our final regex is `\A(0?[1-9]|1[0-2])/(0?[1-9]|12)[0-9]|3[01])/[0-9]{2}\z`.

If you wanted to extract the date from “On 6/2/07 I wrote this” (I did!), you can’t use the `\A` and `\z` anchors. In that case, use Insert Token, Anchors, Word Boundary instead of Beginning or End of The String. A word boundary checks if the match isn’t in the middle of a word or number.

So how about that shiny embossed logo? Easy! Click Insert Token, Comment and type “shiny embossed logo”. Done!

How to Figure This out on Your Own

Of course you’re asking me how you’re supposed to know what to pick from the Insert Token menu. Well, RegxBuddy is designed to teach you about regular expressions while making it much easier to work with them regardless of your experience. So it doesn’t try to hide how regular expressions work. To the contrary: just like the regex tree on the Create panel has a one-on-one relationship with the actual regular expression syntax, so does the Insert Token menu.

At first, you’ll find this confusing. But you’ll soon get the hang of it. The next step after working through this “getting started” tutorial is to read the regular expressions quick start and the help topics for the Insert Token menu. It’s only a few pages. They’ll give you a great overview of exactly what you can do with regular expressions. Then it’ll be much easier to experiment with the Insert Token menu.

Remember:

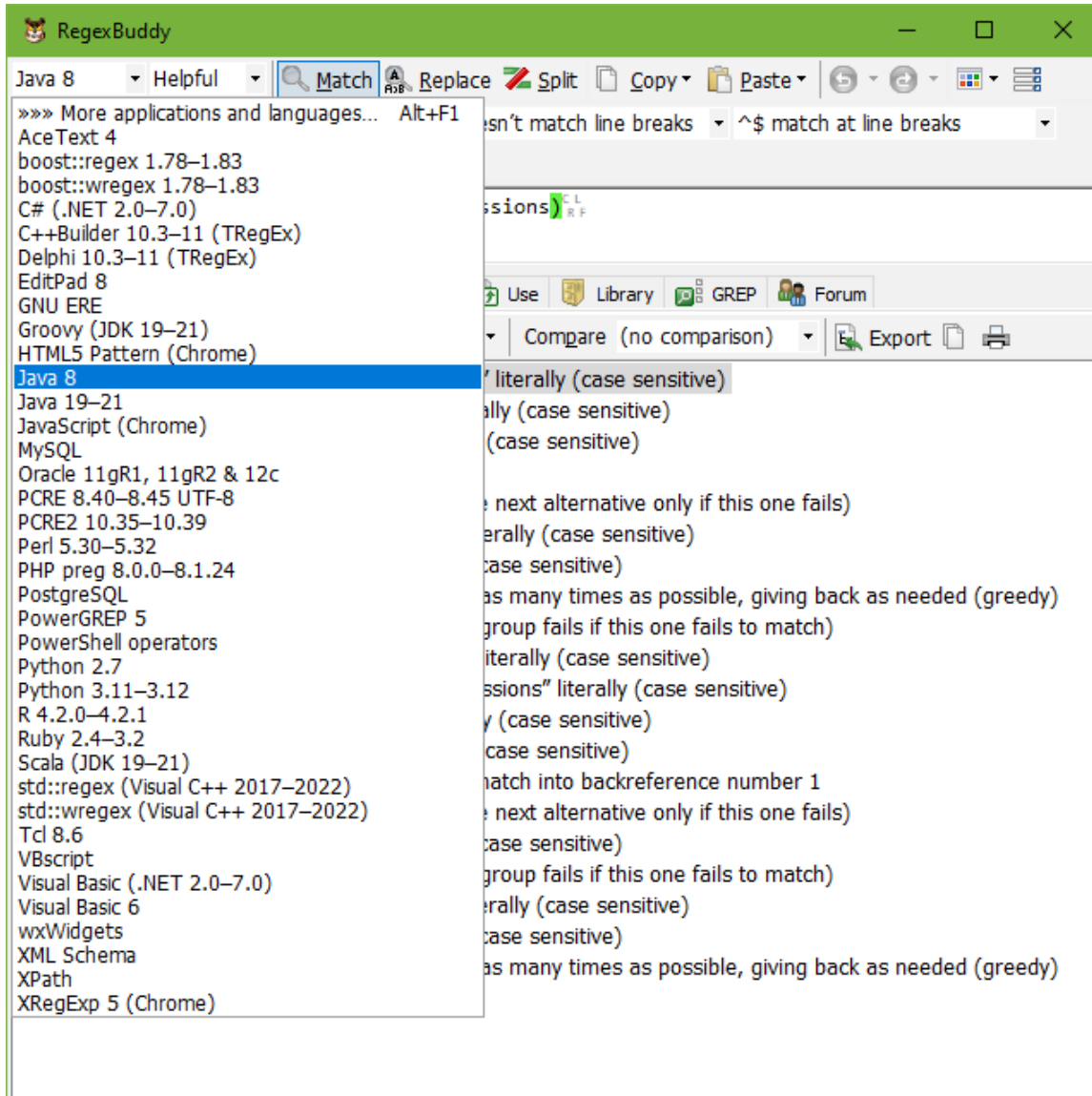
1. Figure out exactly what you want to match, and what you don’t want to match. Write it down.
2. On the Test panel, put in examples of everything you want to match, and everything you don’t want to match.
3. Create more stupid variations of everything you don’t want to match on the Test panel.
4. Craft your regular expression, from left to right, bit by bit.
5. Fix up your regex until the Test panel says you’re good to go.

3. Select Your Application or Programming Language

About the only thing that all regular expression engines have in common is that its designers always try to come up with “new and improved” regex features. The result is that there is a lot of inconsistency in the regex syntax supported by various applications and programming languages. Even different versions of the same application or language can interpret a regex differently. Fortunately, RegexBuddy takes care of all that for you.

When creating a new regular expression, you should select the regex flavor appropriate for the tool or language in which you plan to use the regular expression before creating the regular expression. You can do so via the drop-down list in the top left corner of RegexBuddy’s window, or by pressing Alt+F on the keyboard (F for “flavor”). Your favorite applications are shown directly in the list. By default, these are the latest versions of the most popular applications that RegexBuddy supports.

Selecting a different application in the list changes the way RegexBuddy interprets the active regular expression. It does not modify the regular expression. When pasting a pre-created regular expression into RegexBuddy, select the application the regular expression is intended for, even if it is different from the one you intend to use the regular expression with. If you want to convert a regular expression to a different application, use the Convert panel.



More Applications and Languages

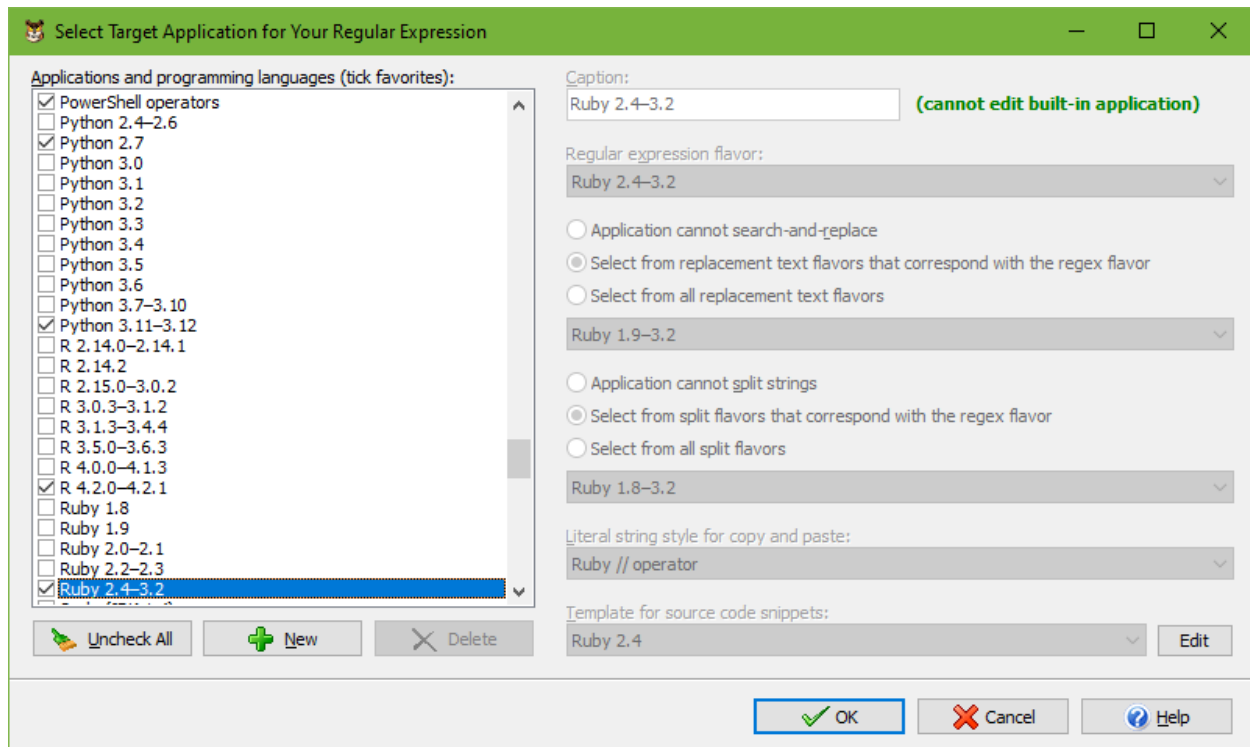
Select “More applications and languages” at the top of the list or press Alt+F1 on the keyboard to show a dialog box with the complete list of predefined applications and languages. The ones you tick in the dialog box are the favorites shown directly in the drop-down list. The built-in applications cannot be edited or deleted. But you can edit their associated source code templates.

An “application” in RegxBuddy is a group of five settings:

1. Regular expression flavor: The syntax supported by and the behavior of the regular expression engine used by this application.
2. Replacement text flavor: The replacement text syntax, if any, supported by this application.
3. Split flavor: The behavior of this application when splitting a string along regex matches, if supported.

4. String style: Rules for formatting a regex or replacement text as a literal string when copying and pasting.
5. Template for source code snippets: Template for generating code snippets on the Use panel. Click the Edit button to edit a template or to create a new one.

When adding custom applications, you are limited to selecting predefined regex, replacement, and split flavors. Because the flavor definitions are very complex, you cannot create your own. You can select them in combinations that were not previously used in any application. If you're working with an application or programming language that uses a regex, replacement, or split flavor that is different from any of the flavors supported by RegxBuddy, you can let us know as a feature request for future versions of RegxBuddy.



You can select a custom source code template for custom applications. First, click the Edit button to open the template editor and save the new template. Close the template editor and then select the newly saved template for the custom application.

Helpful or Strict Emulation Mode

Accurate emulation of an application's regex flavor can sometimes get in the way of helping you create regular expressions, particularly if you're working with a regex flavor that is different than the one you normally use. To alleviate this, you can toggle RegxBuddy between Helpful and Strict mode via the drop-down list that sits right next to the applications drop-down list. In Helpful mode, RegxBuddy tries to be helpful and point out potential mistakes in your regex. In Strict mode, RegxBuddy emulates the selected application exactly, even in situations where the application's behavior isn't very sensible.

In Helpful mode, RegxBuddy helps you deal with unsupported syntax. For example, you may be used to using `\A` to match the start of the string. But in JavaScript, `\A` matches a literal `A`. In Helpful mode,

RegexBuddy assumes that if you wanted to match `A` literally, you'd just enter `A` as your regex. So if you enter `\A`, RegexBuddy takes the liberty of telling you that JavaScript does not support `\A` as a start-of-string anchor. If you double-click the error on the Create panel, RegexBuddy will replace `\A` with `^` which JavaScript does support. In Strict mode, however, RegexBuddy will tell you that `\A` matches a literal `A` in JavaScript, and behave that way on the Test panel.

In Helpful mode, RegexBuddy also helps you deal with deficiencies in the emulated application's regex engine. For example, Java has various issues when dealing with character class intersection. Java's documentation specifies `[a-z&&[def]]` as the proper syntax. In reality, `[a-z&&def]` works just as well. Things get complicated when you mess up the syntax. Java does not treat `[a-z&&[def]ghi]` as an error. It interprets it as `[a-z&&ghi]`, completely ignoring the `[def]` part. With Java selected in Strict mode, RegexBuddy does the same. In Helpful mode, RegexBuddy treats `[a-z&&[def]ghi]` as an error, pointing out that `ghi` overwrites the `[def]` part. But if you select Ruby as your application, there is no difference in Helpful or Strict mode. In both modes, RegexBuddy will treat `[a-z&&[def]ghi]` as `[a-z&&defghi]` which is what Ruby does and what makes sense.

4. Define a Match, Replace, or Split Action

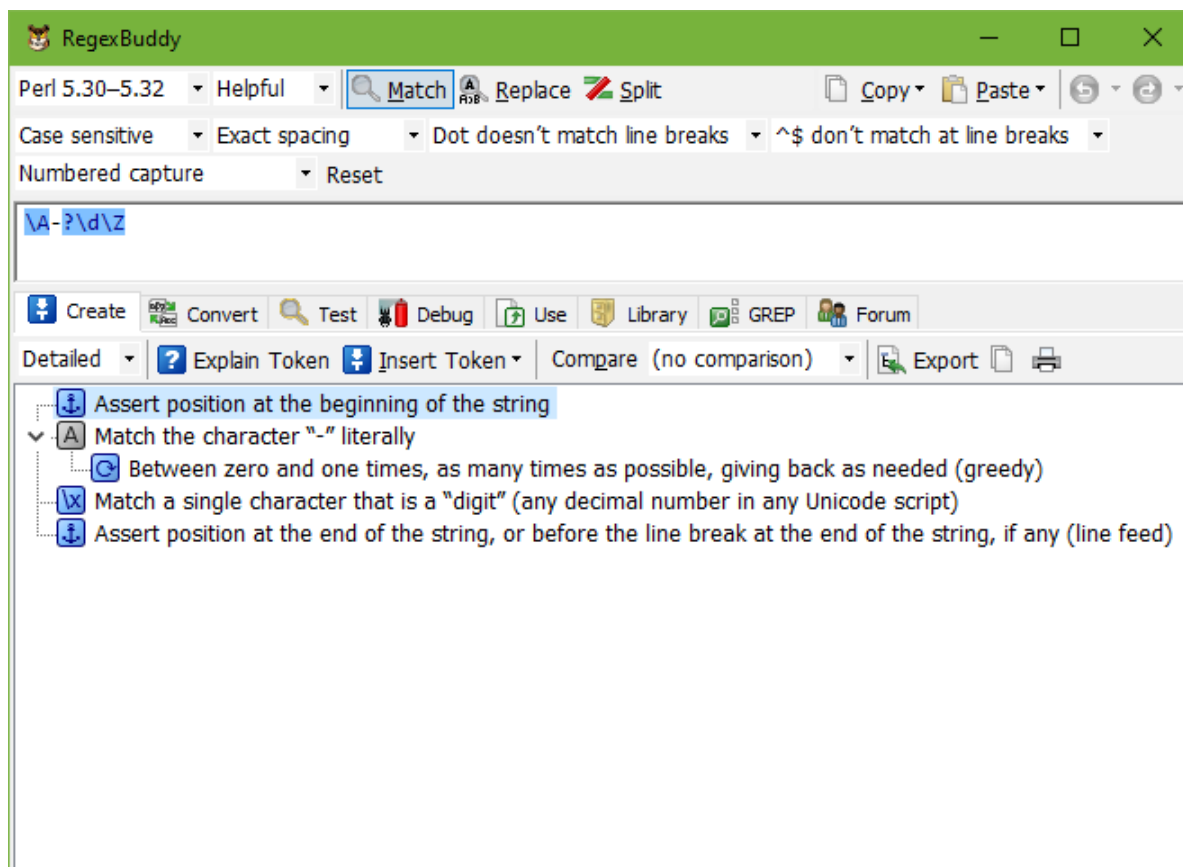
With most tools and languages, you can perform three actions using a regular expression: match, replace or split. You can define, test, debug and implement all three with RegexBuddy.

Match

A match action applies the regular expression pattern to a subject string, trying to find a match. If successful, the match can be applied again to the remainder of the subject string, to find subsequent matches. When you perform a search using a regular expression in a text editor, you are technically executing a match action. The file you are editing is the subject string.

When programming, you can use match actions to validate user input and external data. The regex `\A-?\d\Z`, for example, checks whether an integer number was entered. Match actions make it easy to parse and process data. Use capturing groups to extract just the data you want.

To define a match action in RegexBuddy, click on the Match button near the top of the RegexBuddy window. Enter the regular expression into the text box. You can right-click the text box to insert regex tokens without having to remember their exact syntax.

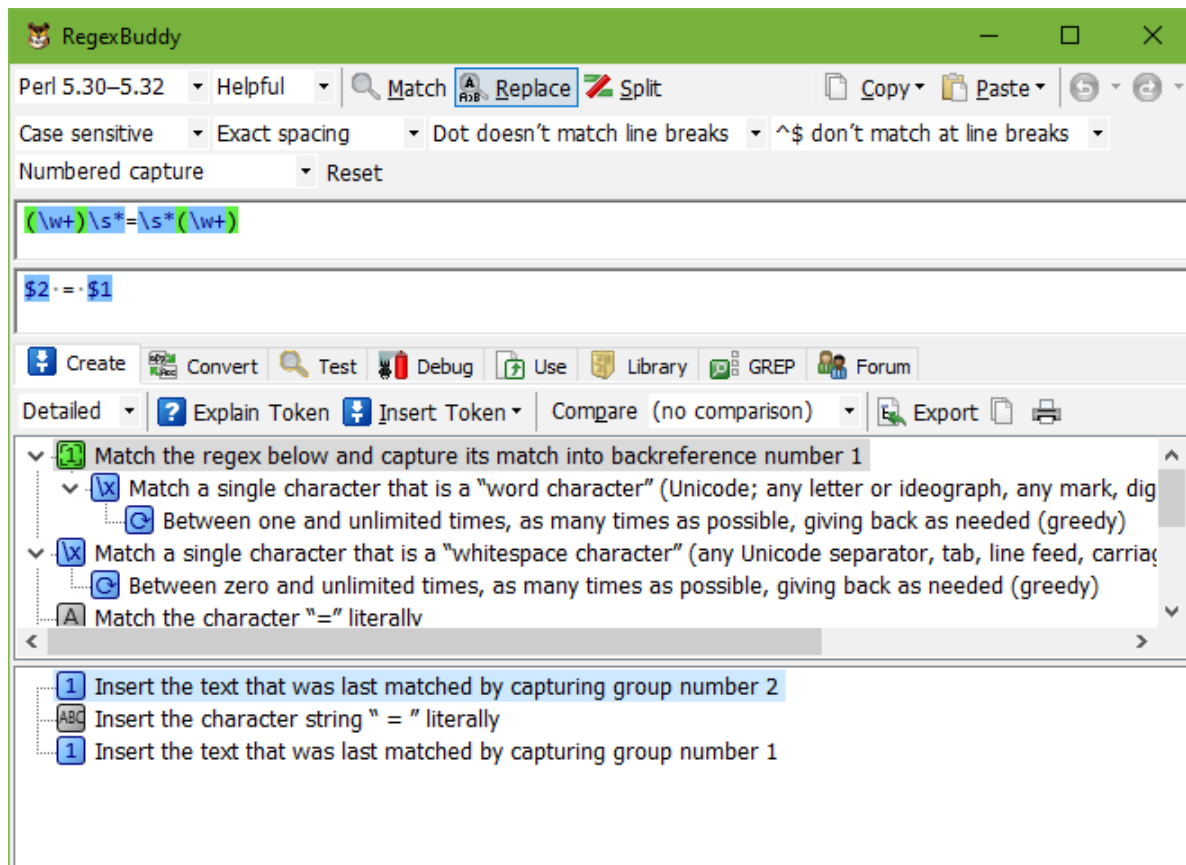


Replace

A replace action applies the regular expression pattern to a subject string, and replaces the match or each of the matches with a replacement string. Depending on the tool or language used, this will either modify the original subject string, or create a new string with the replacements applied. By using backreferences in the replacement text, you can easily perform some pretty complex substitutions. To invert a list of assignments, for example, turning `one = another` into `another = one`, use the regex `(\w+)\s*=\s*(\w+)` and replace with `$2 = $1`.

To define a replace action in RegexBuddy, click on the Replace button at the top. Enter the regular expression in the topmost text box, and the replacement text into the text box just below. To easily insert a backreference into the replacement text, right-click in the text box with the replacement text at the spot where you want to insert the backreference. This will move the text cursor to that position and show the context menu. Move the mouse to the Insert Token item to expand it. Finally, select Use Backreference.

Some applications do not have the ability to search-and-replace. The Replace button is grayed out when you select such an application.



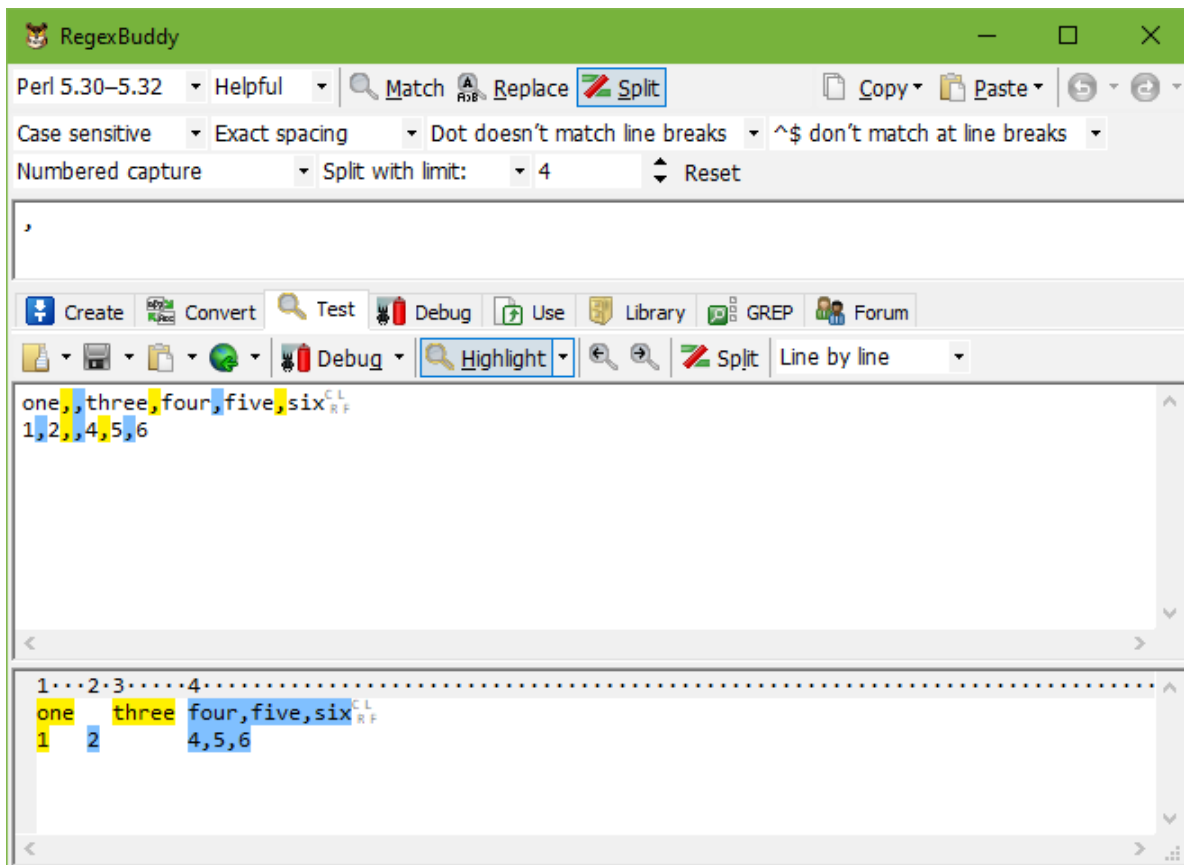
Split

A split action creates an array or list of strings from a given subject string. The first string in the resulting array is the part of the subject before the first regex match. The second string is the part between the first and second matches, the third string the part between the second and third matches, etc. The final string is the remainder of the subject after the last regex match. The text actually matched by the regex is discarded.

In RegexBuddy, simply click on the Split button at the top and enter the regular expression into the text box. You can right-click the text box to insert regex tokens without having to remember their exact syntax.

Many applications allow you to specify a limit for the number of times the string should be split. Some allow you to choose whether text matched by capturing groups should be added to the array and whether empty strings may be added to the array. Those options will appear after the regular expression options when RegexBuddy is in Split mode.

Some applications do not have the ability to split strings. The Split button is grayed out when you select such an application.



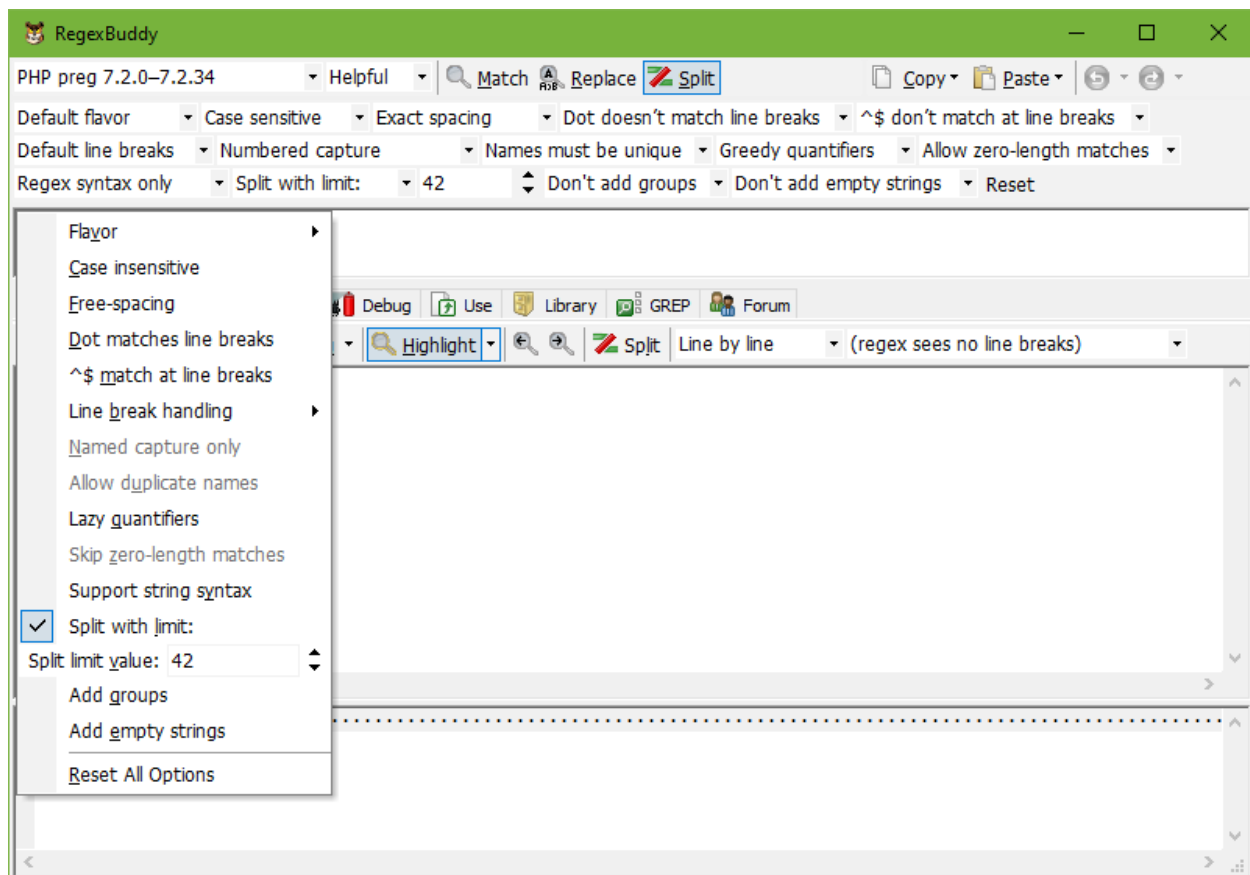
5. Set Regular Expression Options

A mistake many people new to regular expressions make is to take any regex at face value. A regular expression is meaningless in isolation. It will be interpreted by a particular application or programming library. Different tools interpret regular expressions differently. Sometimes the differences are subtle. Sometimes various tools use completely different syntax.

On top of that, most regex engines support various matching modes or options, such as case sensitivity. A regular expression written with the “case insensitive” option in mind will miss most of the intended matches if you forget to turn on that option when you implement the regular expression.

Fortunately, RegexBuddy makes things easy for you. When using RegexBuddy’s library of regular expressions, click the Use button rather than copying and pasting the regex, and RegexBuddy will move over the application and mode selections along with the regex. When converting a regex from one application to another, RegexBuddy takes any differences in supported matching modes into account, and adjusts the regex or warns as needed.

When you implement your regular expression on the Use panel, RegexBuddy will automatically set the same matching options in the source code snippets. If you copy and paste the regular expression manually into your source code, the matching options will *not* be copied over automatically, since they’re not part of the regular expression itself. Make sure to set the options in your own source code.



You can set the regular expression options by selecting different values the drop-down lists on the toolbar above the regular expression in RegxBuddy. The value shown in each drop-down list is the present state of that option. You can also toggle the options by pressing Alt+O on the keyboard and then pressing the underlined letter of the option you want to toggle as it appears in the popup menu. The popup menu uses check marks to indicate which options are on.

By default, RegxBuddy only shows the options that the selected application actually allows you to change. On the Operation tab in the Preferences you can choose to always show all the options that RegxBuddy supports. The above screen shot illustrates this. The options that the active application doesn't support have only one item in the drop-down list, so you can see the state of the option, but not change it. In the Alt+O popup menu, options that can't be changed are grayed out. The options for splitting strings are never shown in Match or Replace mode, even when you turned on the preference to show all options.

- **Flavor**
 - **Default flavor:** Use the application's default regex syntax and matching behavior. For most applications this is the only choice.
 - **ECMAScript:** For std::regex this selects the ECMAScript grammar, which is the default and the most feature-rich. It is somewhat similar to JavaScript's regex syntax. For .NET, this turns on RegexOptions.ECMAScript, which changes the behavior of a few regex tokens to more closely match that of JavaScript. Do not confuse selecting ECMAScript as an option for std::regex or .NET with selecting actual ECMAScript as your application. To work with the actual ECMAScript regex flavor, select JavaScript as your application.
 - **Basic:** Select the basic grammar for std::regex.
 - **Extended:** Select the extended grammar for std::regex.
 - **Grep:** Select the grep grammar for std::regex.
 - **EGrep:** Select the egrep grammar for std::regex.
 - **Awk:** Select the awk grammar for std::regex.
- **Case sensitivity**
 - **Case insensitive:** Differences between uppercase and lowercase characters are ignored. `cat` matches `CAT`, `Cat`, or `CaT` or any other capitalization in addition to `cat`.
 - **Case sensitive:** Differences between uppercase and lowercase characters are significant. `cat` matches only `cat`.
- **Free-spacing mode**
 - **Free-spacing:** Unescaped literal spaces and line breaks in the regex that are not inside a character class are ignored so you can use them to format your regex to make it more readable. In most applications this mode also makes `#` the start of a comment that runs until the end of the line. Unescaped literal whitespace inside character classes does still add that whitespace to the character class.
 - **Free-spacing [...]:** Free-spacing mode that also ignores whitespace inside character classes. Only a few regex flavors support this.
 - **Exact spacing:** Unescaped spaces, line breaks, and `#` characters in the regex are treated as literal characters that the regex must match.
- **Single-line mode**
 - **Dot matches line breaks:** The dot matches absolutely any character, whether it is a line break character or not. Sometimes this option is called "single line mode".
 - **Dot doesn't match line breaks:** The dot matches any character that is not a line break character. Which characters are line break characters depends on the application and the line break mode.
- **Multi-line mode**
 - **^\$ match at line breaks:** The `^` and `$` anchors match after and before line breaks, or at the start and the end of each line in the subject string. Which characters are line break characters

depends on the application and the line break mode. Sometimes this option is called “multi-line mode”.

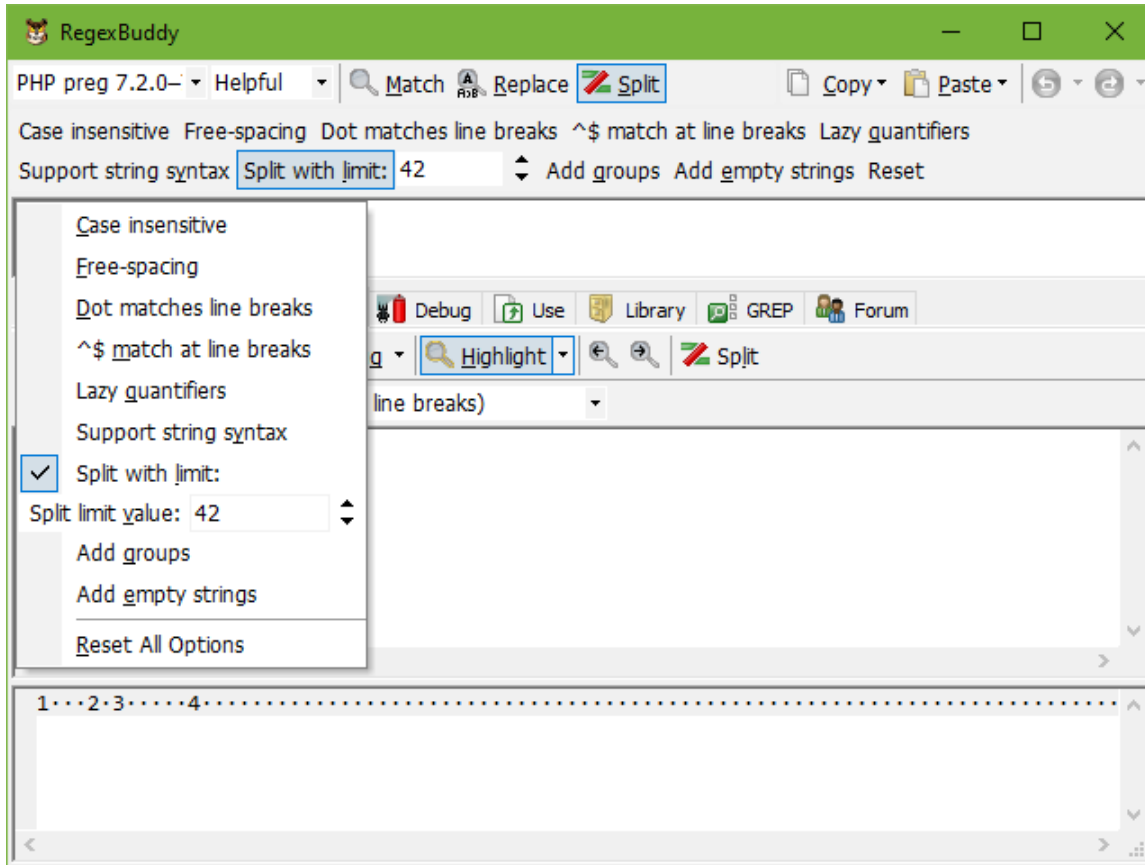
- **^\$ don't match at line breaks:** The `^` and `$` anchors only match at the start and the end of the whole subject string. Depending on the application, `$` may still match before a line break at the very end of the string.
- **^\$ match at line breaks; dot doesn't match line breaks:** Some applications allow these two options only to be set in combination.
- **^\$ don't match at line breaks; dot matches line breaks:** Some applications allow these two options only to be set in combination.
- **Line break handling**
 - **Default line breaks:** The dot and anchors use the application's default interpretation for line breaks.
 - **LF only:** The dot and anchors recognize only the line feed character `\n` as a line break.
 - **CR only:** The dot and anchors recognize only the carriage return character `\r` as a line break.
 - **CRLF pairs only:** The dot and anchors recognize only the line feed and carriage return characters when they appear as a pair as a line break.
 - **CR, LF, or CRLF:** The dot and anchors recognize the line feed and carriage return characters as line breaks, whether they appear alone or as a pair.
 - **Unicode line breaks:** The dot and anchors recognize all Unicode line breaks. This includes the line feed and carriage return characters as line breaks, whether they appear alone or as a pair, as well as the vertical tab `\v`, the form feed `\f`, next line `\u0085`, line separator `\u2028`, and paragraph separator `\u2029` characters.
- **Explicit capture**
 - **Named capture only:** A plain pair of parentheses is a non-capturing group. Named capturing groups still capture. Sometimes this option is called “explicit capture”.
 - **Numbered capture:** A plain pair of parentheses is a numbered capturing group.
- **Duplicate names**
 - **Allow duplicate names:** Multiple named capturing groups in the regular expression are allowed to have the same name. How backreferences to shared names are treated depends on the application.
 - **Names must be unique:** All named capturing groups in the regular expression must have a unique name. Groups with the same name are treated as an error.
- **Lazy quantifiers**
 - **Lazy quantifiers:** Quantifiers are lazy by default. Adding a question mark makes them greedy. So `a*` is lazy and `a*?` is greedy.
 - **Greedy quantifiers:** Quantifiers are greedy by default. If the application supports lazy quantifiers, then adding a question mark makes quantifiers lazy. So `a*` is greedy and `a*?`, if supported, is lazy.
- **Zero-length matches**
 - **Skip zero-length matches:** Skip any zero-length matches found by the regular expression. `\d*` will only match sequences of digits. Some applications skip matches by backtracking, so `\d*|a` matches all sequences of digits and all letters `a`. Other applications skip matches by advancing through the string, so `\d*|a` matches all sequences of digits but never matches `a` because it is skipped over when the first alternative finds a zero-length match.
 - **Allow zero-length matches:** Treat zero-length matches as normal matches. `\d*` will match sequences of digits and will also find a zero-length at each position in the string that is not followed by a digit.
- **String syntax**
 - **Support string syntax:** Make RegexpBuddy recognize syntax that is supported by string literals in a programming language as if it were part of the regular expression or replacement

syntax. Select this option if you plan to use your regex and/or replacement text as a literal string in source code. For example, with Python 3.2 and earlier, RegxBuddy will recognize `\uFFFF` in this mode, because Python recognizes such escapes in literal strings.

- **Regex syntax only:** Tell RegxBuddy not to recognize any syntax other than what the regular expression engine itself supports. Select this option if you plan to provide the regex and/or replacement text as user input to the application. For example, with Python 3.2 and earlier, RegxBuddy will treat `\uFFFF` as an error in this mode, because the regex engine in Python 3.2 and earlier does not recognize this escape.
- **Split limit**
 - **Split without limit:** Split the string as many times as possible.
 - **Split with limit:** Limit the number of times the string is split. Some applications take this number as the maximum number of regex matches to split on, while other applications take this number as the maximum number of strings in the returned array. Some applications add the unsplit remainder of the string to the array, while others don't. A zero or negative limit makes some applications split the string as many times as possible, while others will not split the string at all. A zero or negative limit also makes some applications discard empty strings from the end of the returned array.
- **Split capture**
 - **Add groups:** When splitting a string, add text matched by capturing groups to the array. Some applications add all groups, while others add only the first or the last group.
 - **Don't add groups:** When splitting a string, do nothing with text matched by capturing groups.
- **Split empty**
 - **Add empty strings:** When splitting a string, regex matches that are adjacent to each other or the start or end of the string and capturing groups that find zero-length matches (when adding groups) cause empty strings to be added to the array. Some applications may not add empty strings to the array in certain situations even when this option is set, sometimes depending on whether and which limit you specified
 - **Don't add empty strings:** When splitting a string, never add any empty strings to the returned array.

Default Options

The drop-down lists use positive labels to indicate all states. The case sensitivity or insensitivity option, for example, is always indicated as “case sensitive” and “case insensitive”. This avoids confusing double negations like “case insensitive off”. It also avoids confusion when switching between applications where one is case sensitive by default with an option to make it case insensitive, and the other is case insensitive by default and has an option to make it case sensitive.



But if the potential confusion of toggle buttons does not bother you, you choose to “show options that can be changed with toggle buttons that indicate the “on“ state” on the Operation tab in the Preferences. The above screen shot shows this. The benefit is that the buttons need only one click to toggle an option while drop-down lists require two. The actual application you will use the regex with likely also has toggle buttons or checkboxes rather than drop-down lists. But if you switch between applications in RegxBuddy, you will need to take into account that the meaning of the buttons may invert. If an application is case sensitive by default, the button will say “case insensitive”. If you then switch to an application that is case insensitive by default, the button will change its label to “case sensitive” and will toggle its state (depressed or not). By changing their labels the buttons ensure that the application’s default options are being used when all buttons are off (not depressed).

Regardless of whether options are shown with drop-down lists or toggle buttons, you can click the Reset button or select Reset All Options in the popup menu to change all options to the defaults for the current application.

6. Insert a Token into The Regular Expression

RegexBuddy makes it easy to build regular expressions without having to remember every detail of the complex regular expression syntax.

To insert a token into your regular expression, right-click in the editor box for the regular expression at the spot where you want to insert the token. This will move the cursor to that position, and show the context menu. In the context menu, select Insert Token. Alternatively, move the cursor by left-clicking or using the arrow keys. Then open the Insert Token menu by clicking the Insert Token button on the Create panel or pressing Alt+I on the keyboard.















Some tokens, including lookaround, are grouping tokens. You can insert a group on its own. RegexBuddy will then put the text cursor inside the newly added group, so you can fill it right away. To place an existing part of the regular expression inside the new group, first select that part, and then add the grouping token.















Quantifiers are a special case. If you select part of the regular expression before inserting a quantifier, RegexBuddy will turn the selection into a non-capturing group and apply the quantifier to that. If not, the quantifier is inserted as a normal token, and it will repeat whatever precedes it.

When analyzing a regular expression on the Create panel, you can easily designate the spot where you want to insert the token. Click on a token in the regex tree, and the new token will be inserted right after it. Click on the Insert Token button in the toolbar to access the Insert Token menu.

List of Regex Tokens

The Insert Token menu offers the following items. Note that depending on the regular expression flavor that you're working with, certain items may not be available, or may insert different tokens into the regular expression. Some regex flavors don't offer certain features, or use a different syntax. The Unicode grapheme item, for example, is disabled for regex flavors that don't support Unicode. It inserts `\X` for flavors like Perl that have a specific token for matching Unicode graphemes. For flavors that support Unicode but not `\X`, it inserts `(?>\P{M}\p{M}*)` which uses the Unicode property syntax to match a single Unicode grapheme.

-  Literal text
-  Non-printable character
-  Any character including line breaks
-  Any character excluding line breaks
-  8-bit character
-  Unicode category
-  Unicode script
-  Unicode block
-  Unicode character
-  Unicode grapheme
-  Shorthand character class
-  Character class
-  POSIX class
-  Anchors

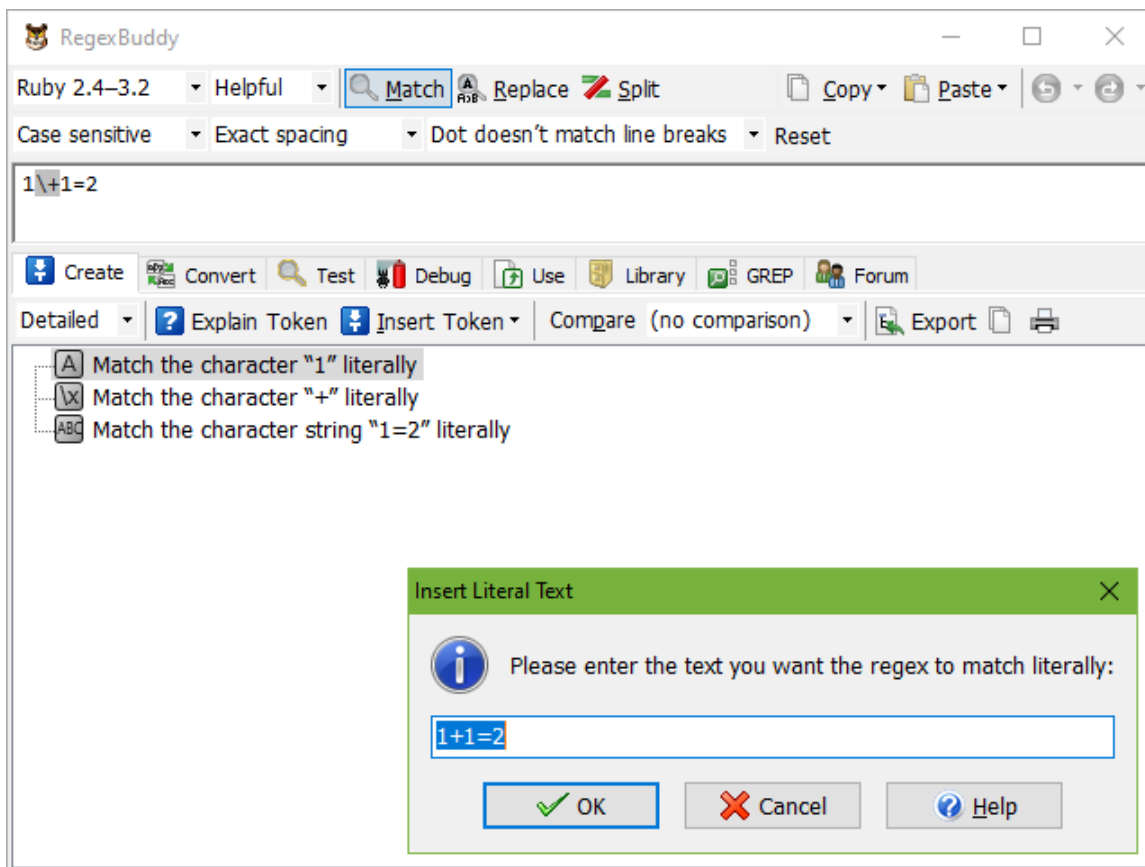
-  Quantifier
-  Alternation
-  Numbered Capturing group
-  Named Capturing Group
-  Backreference
-  Recursion
-  Subroutine Call
-  Conditional
-  Non-capturing Group
-  Atomic Group
-  Lookaround
-  Mode modifiers
-  Comment
-  RegexMagic

7. Insert a Regex Token to Match Specific Characters

The Insert Token button on the Create panel makes it easy to insert the following regular expression tokens to match specific characters. See the Insert Token help topic for more details on how to build up a regular expression via this menu.

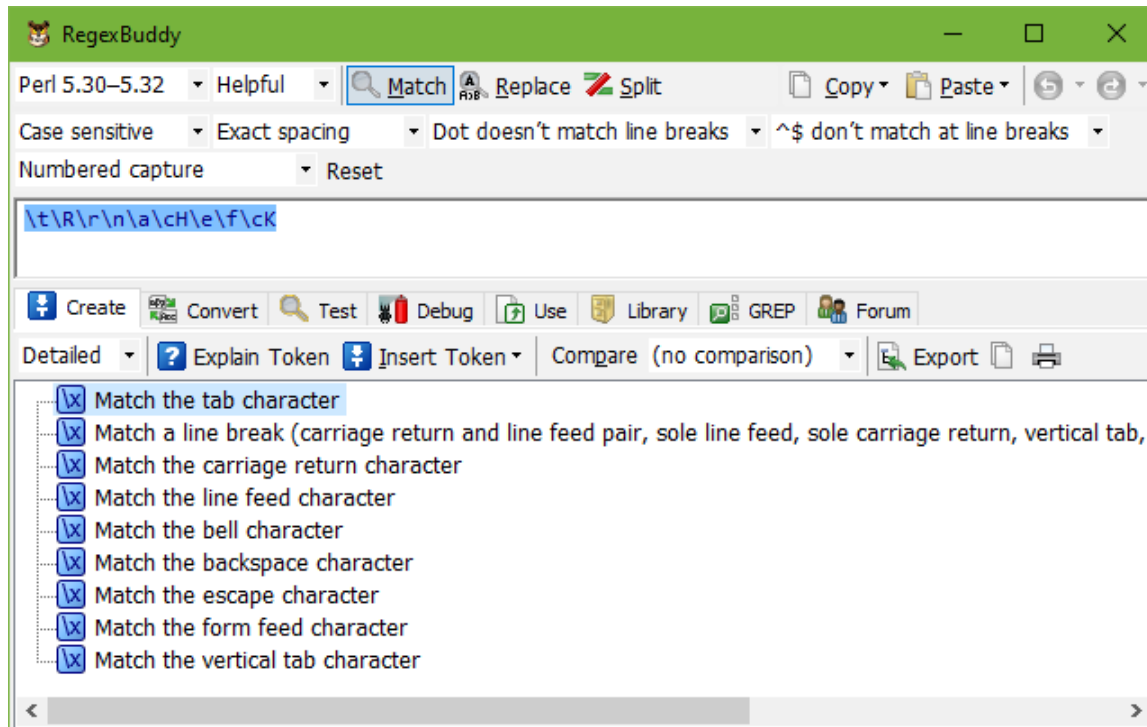
Literal Text

Enter one or more characters that will be matched literally. RegxBuddy will escape any metacharacters you enter with a backslash. Metacharacters are characters that have a special meaning in regular expressions.



Non-Printable Character

Match a specific non-printable character, such as a tab, line feed, carriage return, alert (bell), backspace, escape, form feed, or vertical tab. You can insert non-printable characters directly into the regular expression, or inside a character class. If the selected application supports any kind of escape sequence that represents the character you want to insert, then RegxBuddy inserts that escape sequence. Otherwise, RegxBuddy inserts the non-printable character directly.



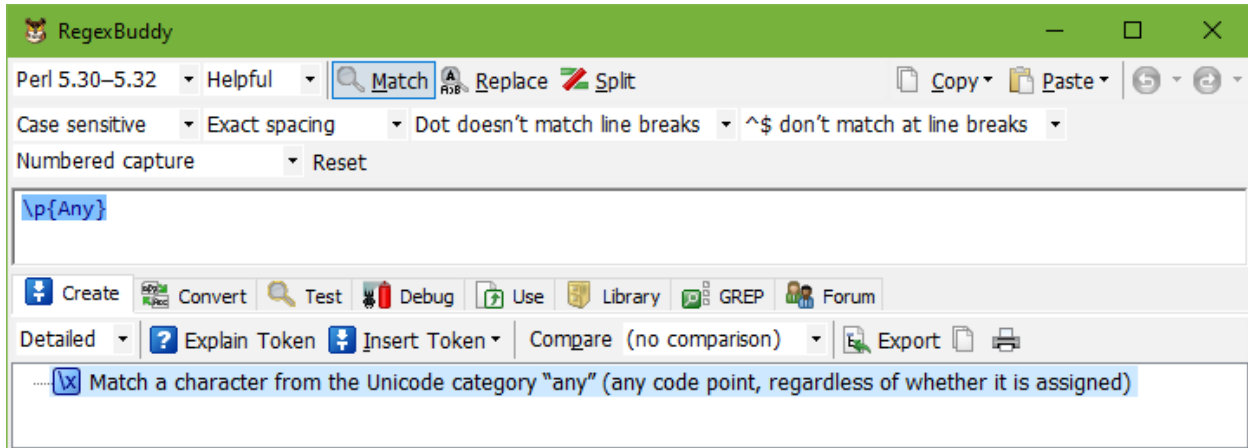
Any Line Break

Some applications such as Perl and PCRE support `\R` which matches any single line break regardless of its style, and regardless of the active line break mode. This includes the line feed and carriage return characters as line breaks, whether they appear alone or as a pair, as well as the vertical tab `\v`, the form feed `\f`, next line `\u0085`, line separator `\u2028`, and paragraph separator `\u2029` characters. So this token matches two characters in case of a CRLF line break, and a single character in case of any other line break.

A few applications such as EditPad 7 and PowerGREP 4 transparently handle differences between LF, CR, and CRLF line breaks. When the regex contains one of these line breaks literally (an actual line break—not the `\r` or `\n` tokens) it will match any of these line breaks. For these applications, the Insert Token|Non-Printable Characters|Any Line Break command inserts a literal line break into the regex.

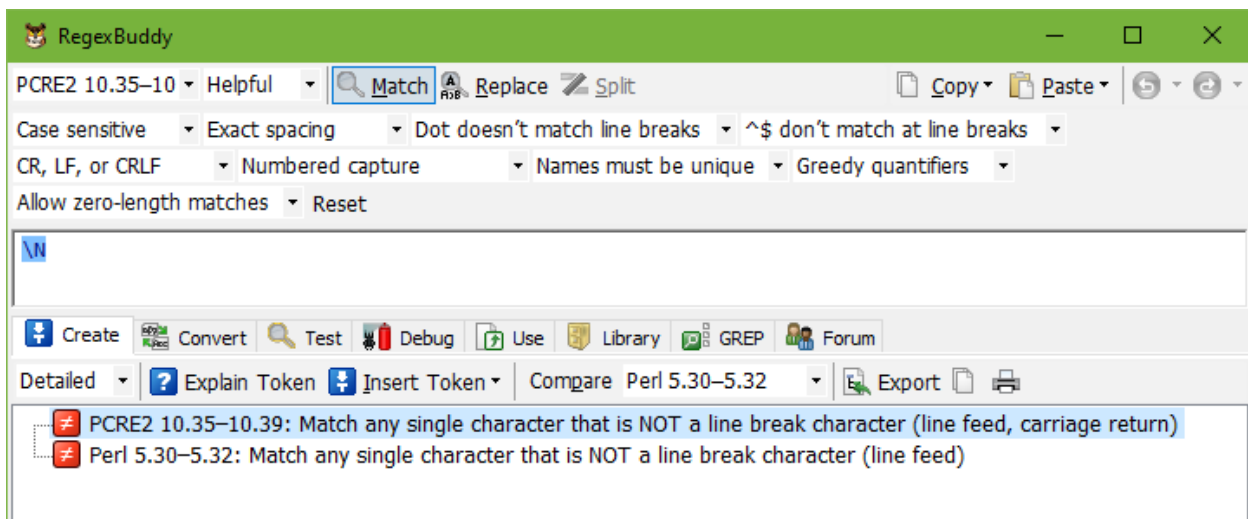
Any Character Including Line Breaks

If the application has a specific token like `\p{Any}` that always matches any single character, regardless of whether the character is a line break, then this item inserts that. If not, the item inserts a dot if the dot matches line breaks option can be turned on. It will be turned on when needed. If the option is not available or the regex already relies on the option being off, then the item inserts a character class that contains the full range of characters.



Any Character Excluding Line Breaks

If the application has a specific token like `\N` that always matches any single character that is not a line break, then this item inserts that. If not, the item inserts a dot if the dot matches line breaks option can be turned off. It will be turned off when needed. If the option is not available or the regex already relies on the option being on, then the item inserts a negated character class that matches any character except line breaks.



8-bit Character

Matches a specific character from an 8-bit code page. Use this to insert characters that you cannot type on your keyboard when working with an application or programming language that does not support Unicode.

In the screen that appears, first select the code page or encoding that you will be working with in the application where you'll implement your regular expression. The code pages labeled "Windows" are the Windows "ANSI" code pages. The default code page will be the code page you're using on the Test panel, if that is an 8-bit code page. To properly test your regular expression, you'll need to select the same code page on the Test panel as you used when inserting 8-bit characters into your regex.

RegexBuddy shows you a grid of all available characters in that code page. Above the grid, choose whether you want to match only one particular character, or if you want to match one character from a number of possible characters. If you select to match one character, click on the character in the grid and then click OK. Otherwise, clicking on a character in the grid will toggle its selection state. Select the characters you want, and click OK.

RegexBuddy inserts a single hexadecimal character escape in the form of `\xFF` into your regular expression to match the character you selected. If you select multiple characters, RegexBuddy puts the hexadecimal escapes for them in a character class. If your regex flavor does not support hexadecimal escapes, RegexBuddy inserts the characters literally.

RegexBuddy

PCRE 4.0-4.4 | Helpful | Match | Replace | Split | Copy | Paste |

`[\x43\x52\xA9\xAE]`

Create | Convert | Test | Debug | Use | Library | GREP | Forum

Detailed | Explain Token | Insert Token | Compare (no comparison) | Export |

Match a single character present in the list below

- The character "C" which occupies position 0x43 (67 decimal) in the character set (case sensitive)
- The character "R" which occupies position 0x52 (82 decimal) in the character set (case sensitive)
- The character with position 0xA9 (169 decimal) in the character set
- The character with position 0xAE (174 decimal) in the character set

8-bit Character Map

8-bit code page:
Windows 1252: Western European

Match a single character

Match one character out of a list of characters

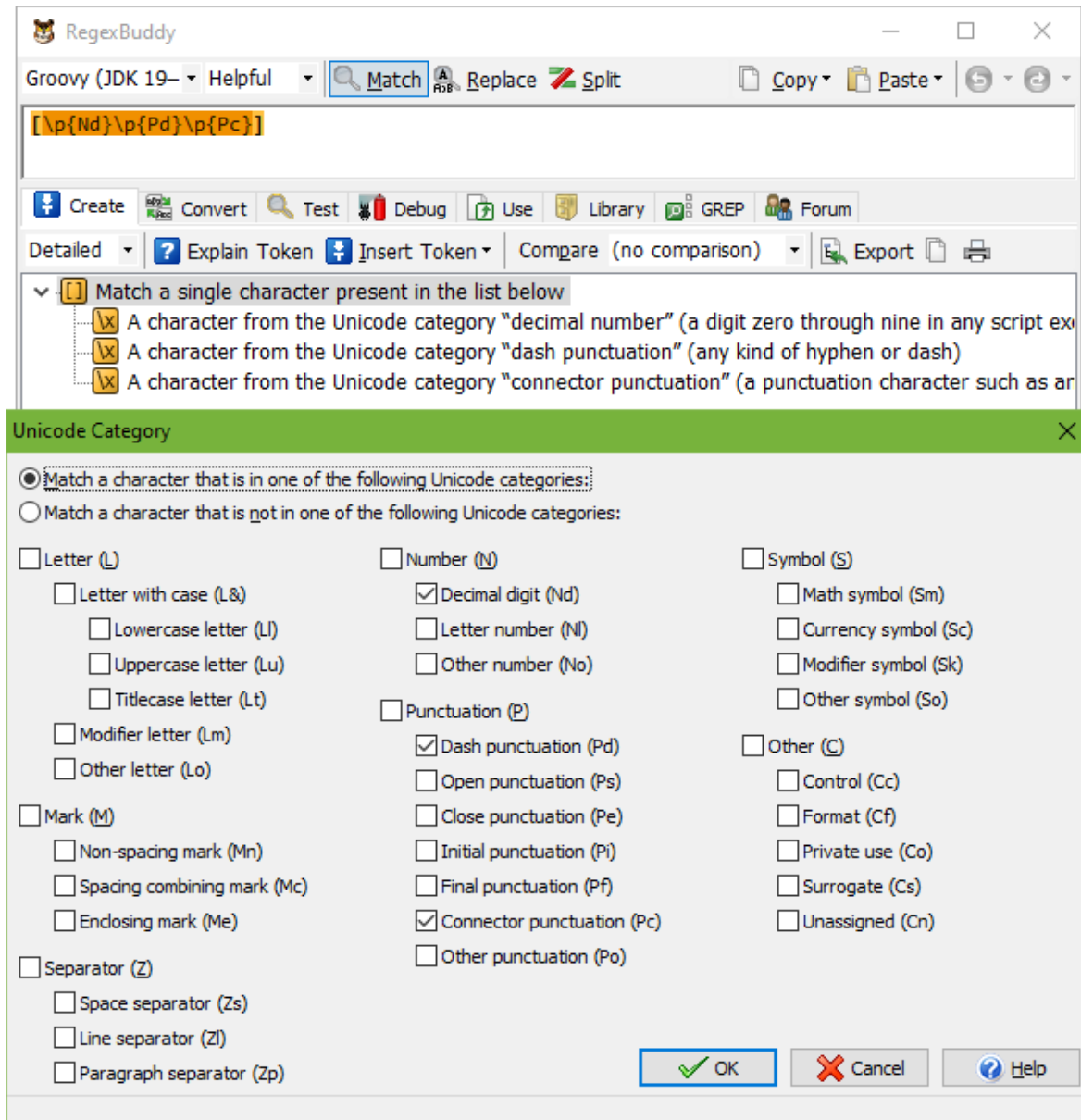
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		⓪	Ⓛ	Ⓜ	Ⓝ	Ⓞ	Ⓟ	Ⓠ	Ⓡ	Ⓢ	Ⓣ	Ⓤ	Ⓥ	Ⓦ	Ⓧ	Ⓨ
1		▶	◀	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
2		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	␣
8	€	×	,	f	"	...	†	‡	^	%	Š	<	Œ	×	Ž	×
9	×	'	'	"	"	•	—	—	~	™	š	>	œ	×	ž	ÿ
A		i	ç	£	¤	¥		§	"	©	a	«	¬	-	®	¯
B	°	±	²	³	´	µ	¶	·	,	¹	º	»	¼	½	¾	¿
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

OK | Cancel | Help

8. Insert a Regex Token to Match Unicode Characters

The Insert Token button on the Create panel makes it easy to insert the following regular expression tokens to match Unicode characters. See the Insert Token help topic for more details on how to build up a regular expression via this menu.

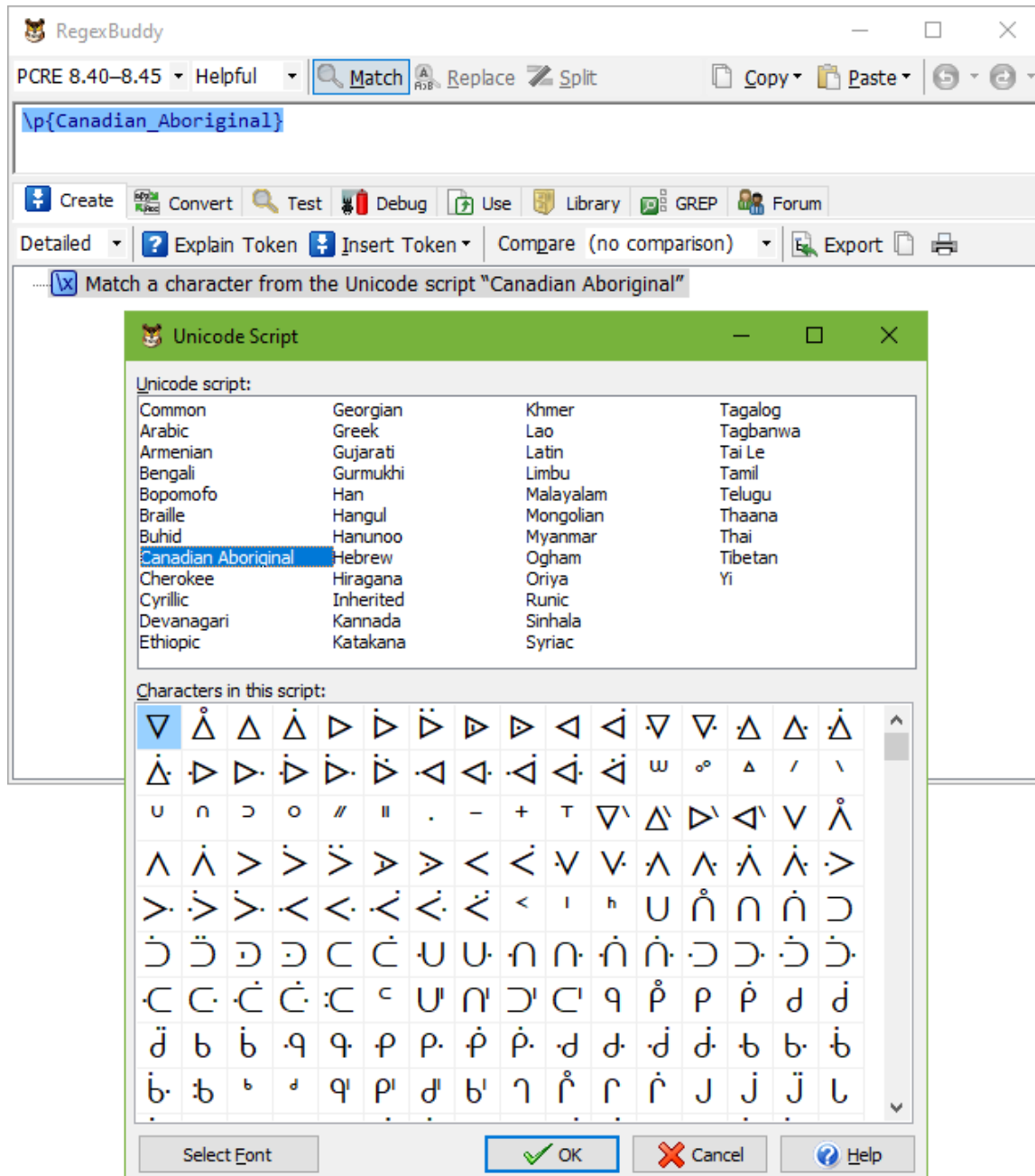
Unicode Category



The Unicode standard places each character into exactly one category. Insert a regular expression token to match a Unicode category if you want to match any character from a particular Unicode category. This makes it easy to match any letter, any digit, etc. regardless of language, script or text encoding.

In the window that appears, select one or more categories that the character you want to match should belong to. If you select more than one category, RegexBuddy will combine the Unicode category regex tokens into a character class to match any character belonging to any of the categories you selected.

Unicode Script



The Unicode standard places each assigned code point (character) into one script. A script is a group of code points used by a particular human writing system. Insert a regular expression token to match a Unicode script if you want to match any character from a particular Unicode script. This makes it easy to match any

character from a certain writing system. Note that a writing system is not the same as a language. Some writing systems like Latin span multiple languages, while some languages like Japanese have multiple scripts.

In the window that appears, select the script that you're interested in. RegxBuddy will insert a regex token that matches any single character from the script.

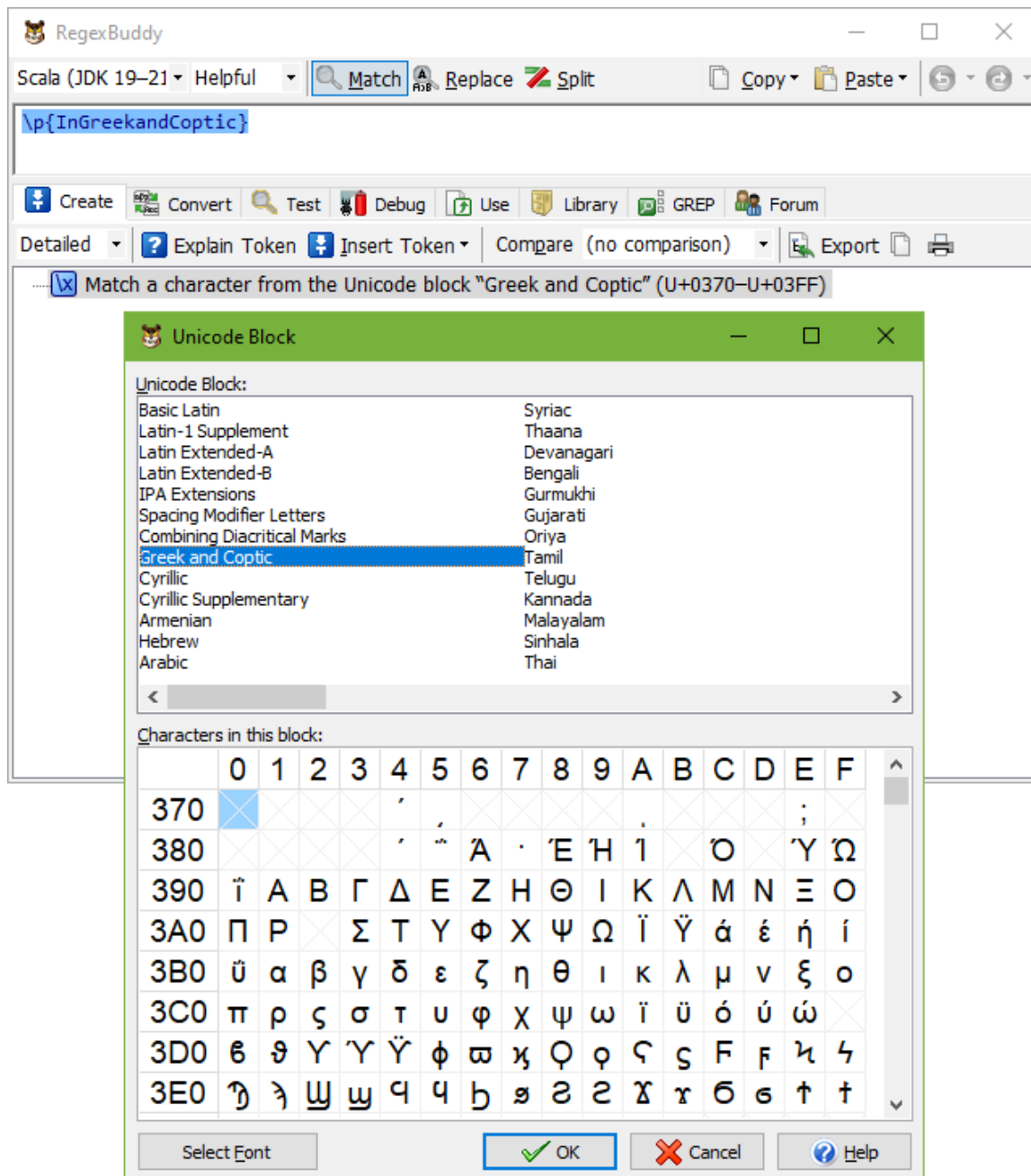
The window will show a preview of the characters in the script. If you move the mouse over the grid, you can see the hexadecimal and decimal representations of each character's code point occupies in the Unicode standard. If you see a great number of squares instead of characters in the grid, click the Select Font button to change the grid's font. The squares indicate the font cannot display the character. The last row of the grid may have squares that are crossed out with thin gray lines. This simply indicates the script doesn't have any more characters to fill up the last row.

Unicode Block

The Unicode standard divides the Unicode character map into different blocks or ranges of code points. Characters with similar purposes are grouped together in Unicode blocks. The arrangement is not 100% strict. Some characters are placed in what seems the wrong block, mostly for historic reasons (i.e. compatibility with legacy character encodings). Though some blocks have the same names as scripts, they don't necessarily include the same characters. If you want to match characters based on their meaning to human readers, use Unicode scripts. If you want to match characters based on their Unicode code points, use Unicode blocks.

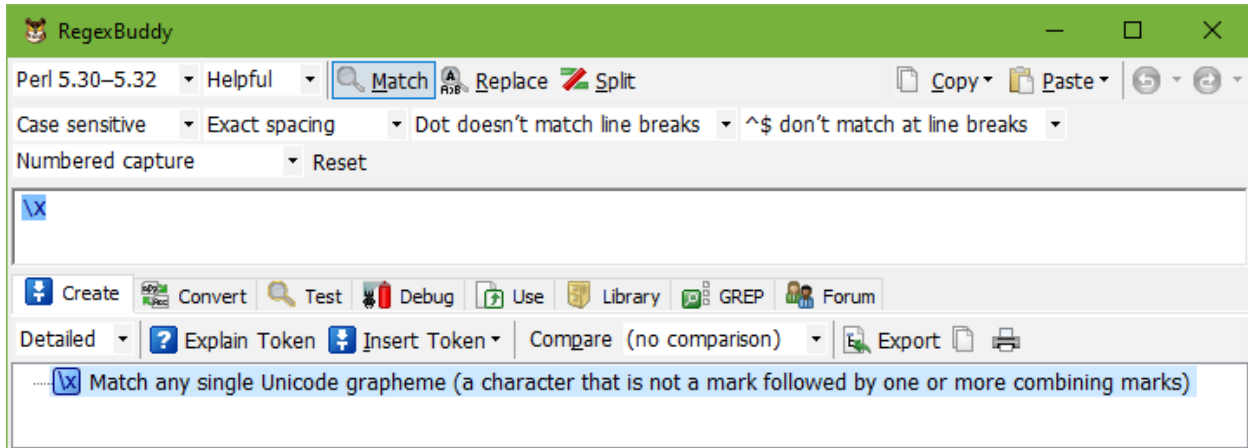
In the window that appears, select the block that you're interested in. RegxBuddy will insert a regex token that matches any single character from the block.

The window will show a preview of the characters in the block. If you move the mouse over the grid, you can see the hexadecimal and decimal representations of each character's code point occupies in the Unicode standard. If you see a great number of squares instead of characters in the grid, click the Select Font button to change the grid's font. The squares indicate the font cannot display the character. The grid may have squares that are crossed out with thin gray lines. That means that the Unicode standard does not assign any characters to those code points. The regex token to match a Unicode block will match any code point in the block, whether a character is assigned to it or not.



Unicode Grapheme

Insert `\X` or equivalent syntax to match any Unicode grapheme.



Unicode Character

Matches a specific Unicode character or Unicode code point. Use this to insert characters that you cannot type on your keyboard when working with an application or programming language that supports Unicode.

In the screen that appears, RegxBuddy shows a grid with all available Unicode characters. Since the Unicode character set is very large, this can be a bit unwieldy. If you know what Unicode category the character you want belongs to, select it from the drop-down list at the top to see only characters of that category. If you move the mouse over the grid, you can see the hexadecimal and decimal representations of each character's code point in the Unicode standard.

If you see a great number of squares instead of characters in the grid, click the Select Font button to change the grid's font. The squares indicate the font cannot display the character. With the "all code points" character map option selected, certain squares will be crossed out with thin gray lines. These squares indicate unassigned Unicode code points. These are reserved by the Unicode standard for future expansion. With any other character map option selected, the last row of the grid may have squares that are crossed out with thin gray lines. This simply indicates the selected category doesn't have any more characters to fill up the last row.

Above the grid, choose whether you want to match only one particular character, or if you want to match one character from a number of possible characters. If you select to match one character, click on the character in the grid and then click OK. Otherwise, clicking on a character in the grid will toggle its selection state. Select the characters you want, and click OK.

RegxBuddy inserts a single Unicode character escape in the form of `\uFFFF` or `\x{FFFF}` into your regular expression to match the character you selected. If you select multiple characters, RegxBuddy puts the Unicode escapes for them in a character class. If your regex flavor does not support Unicode escapes, RegxBuddy inserts the characters literally.

9. Insert a Shorthand Character Class

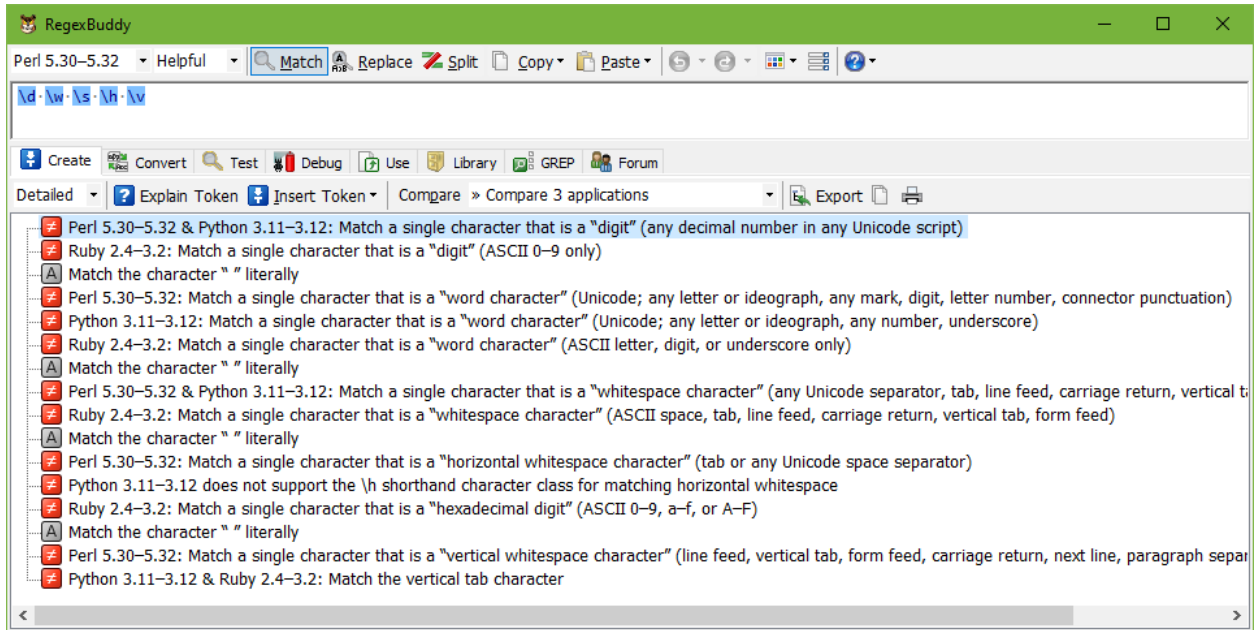
The Insert Token button on the Create panel makes it easy to insert shorthand character classes without having to remember the exact letter for the shorthand. See the Insert Token help topic for more details on how to build up a regular expression via this menu.

A shorthand character class matches a single character from a specific set of characters. Most regex flavors offer shorthands for digits, word characters, and whitespace. You can insert shorthand character classes directly into the regular expression, or inside a character class.

RegexBuddy supports the following shorthands. The actual characters that each shorthand matches depend on the application. No application supports all of these, as `\h` indeed appears twice in the list with two different meanings.

- **Digit:** `\d` matches a single digit 0–9. It may also match digits in other scripts, depending on the regex flavor.
- **Word Character:** `\w` matches a single “word character” which includes letters, digits, underscores. It may also match other characters, depending on the regex flavor.
- **Whitespace Character:** `\s` matches a single “whitespace character”. This includes spaces and line breaks, depending on the regex flavor.
- **Uppercase Letter:** `\u` matches a single uppercase letter A–Z. It may also match uppercase letters in other scripts, depending on the regex flavor.
- **Lowercase Letter:** `\l` matches a single lowercase letter a–z. It may also match lowercase letters in other scripts, depending on the regex flavor.
- **Hexadecimal Digit:** `\h` matches a single hexadecimal digit (0–9, A–F, and a–f).
- **Horizontal Space Character:** `\h` matches a single “horizontal whitespace character”, which includes the tab and any Unicode space separator.
- **Vertical Space Character:** `\v` matches a single “vertical whitespace character”, which includes the line feed `\n`, carriage return `\r`, vertical tab `\v`, form feed `\f`, next line `\u0085`, line separator `\u2028`, and paragraph separator `\u2029` characters. `\v` is traditionally used to match the vertical tab, which it still is in languages such as JavaScript, Python, and Ruby. But recent versions of Perl and PCRE treat `\v` as a shorthand that includes the vertical tab along with all other vertical whitespace.
- **Initial Character in XML Name:** `\i` matches any character that can appear as the first character in an XML name.
- **Consecutive Character in XML Name:** `\c` matches any character that can appear as the second or following character in an XML name.

If an application does not support a particular shorthand, then the menu item for this shorthand may generate alternative syntax such as a POSIX class or a Unicode category. Or, the item may be disabled.



10. Insert a Regex Token to Match One Character out of Many Possible Characters

The Insert Token button on the Create panel makes it easy to insert a tokens to match one character out of many possible characters. See the Insert Token help topic for more details on how to build up a regular expression via this menu.

A character class matches a single character out of a set of characters. Inside the character class, you can use a wide range of regular expression tokens. Many of these also work outside character classes. Some of them are unique to character classes.

First, you can specify if you want to match a character from the list of characters that you specify, or if you want to match a character that's not in your list. If you've already used the Insert Character Class dialog during the current RegexBuddy session, it will default to the last class you edited. Click the Clear button if you want to start with a clean slate.

To match a character out of a bunch that you can type on your keyboard, simply type them into the Literal characters box. RegexBuddy takes care of escaping those characters that have a special meaning inside character classes. Incidentally, these are *not* the same characters as those that have a special meaning in regular expressions outside character classes.

If you want to match a character out of a series that you cannot type on your keyboard, click the ellipsis (...) buttons next to the 8-bit characters or Unicode characters fields to pick the characters you want to insert. You'll get the same selection dialogs as when you try to insert a token for matching 8-bit or Unicode characters directly into your regular expression. RegexBuddy will generate the same `\xFF` and `\uFFFF` syntax, though the `\x` and `\u` will not be shown in the Insert Character Class dialog for brevity.

The ellipsis buttons next to the Unicode categories, Unicode scripts, Unicode blocks, and POSIX classes fields also show the same dialogs as their corresponding items in the Insert Token menu. The same regex tokens will be inserted into the character class, and shown in the Insert Character Class dialog.

The second column in the dialog box starts with six checkboxes. The three at the left allow you to include three common shorthand character classes in your character class. Their negated counterparts don't have checkboxes. Using negated shorthands in character classes is not recommended. The three other checkboxes allow you to easily insert three commonly used non-printable characters. If you want to use others, you can enter them in the "other character class tokens" field further down.

The Insert Character Class dialog provides four sets of edit boxes where you can specify character ranges. Enter the first character in the range at the left, and the last character at the right of the long dash. Though you can use any sort of character as the start or end point of a range, you should use only letters and digits to define ranges. It's easy for anyone to understand which characters are included in a range of letters or digits. If you want to include a range of 8-bit or Unicode characters, you can use the `\xFF` or `\uFFFF` syntax to specify the range's endpoints.

In the "other character class tokens" field you can enter additional tokens that RegexBuddy should include into the character class. Whatever you type in here should be valid regular expression syntax inside a character class. For example, you could type `\e` to include the escape character. Essentially, this is a catch-all field if you like to use certain rarely used regex constructs that RegexBuddy doesn't provide special support for in its

Insert Character Class dialog. Typically, you'll only use this if you double-clicked on a character class in the regex tree of a regular expression created by somebody else.

The “intersected character class” and “subtracted character class” field allows you to specify a character class that should be intersected with or subtracted from the one you've just specified. Click the ellipsis (...) button to open a second Insert Character Class dialog to define the character class to be intersected or subtracted.

Intersection is written as `[A&&[B]]` and results in a character class that only matches characters that are in both sets **A** and **B**. Subtraction is written as `[A-[B]]` and results in a character class that only matches characters that are in set **A** but not in set **B**. Both options will be available if the current application supports character class intersection or character class subtraction. `[A-[^B]]` is the same as `[A&&[B]]`, and `[A&&[^B]]` is the same as `[A-[B]]`, so RegexBuddy can easily adapt your choice to the available syntax. In these examples, **A** and **B** are placeholders for larger sets of characters. For example, if you specify the Unicode script `\p{Thai}` for your base character class, and you specify the Unicode category `\p{Number}` for your subtracted character class, then the resulting class `[\p{Thai}-[\p{Number}]]` or `[\p{Thai}&&[^{\p{Number}}]]` will match any Thai character that is not a number.

RegexBuddy

Java 19-21 Helpful Match Replace Split Copy Paste

`[^\x09\x0A\x0E\x0F\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1A\x1B\x1C\x1D\x1E\x1F\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2A\x2B\x2C\x2D\x2E\x2F\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3A\x3B\x3C\x3D\x3E\x3F\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4A\x4B\x4C\x4D\x4E\x4F\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5A\x5B\x5C\x5D\x5E\x5F\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6A\x6B\x6C\x6D\x6E\x6F\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7A\x7B\x7C\x7D\x7E\x7F\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8A\x8B\x8C\x8D\x8E\x8F\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9A\x9B\x9C\x9D\x9E\x9F\xA0\xA1\xA2\xA3\xA4\xA5\xA6\xA7\xA8\xA9\xAA\xAB\xAC\xAD\xAE\xAF\xB0\xB1\xB2\xB3\xB4\xB5\xB6\xB7\xB8\xB9\xBA\xBB\xBC\xBD\xBE\xBF\xC0\xC1\xC2\xC3\xC4\xC5\xC6\xC7\xC8\xC9\xCA\xCB\xCC\xCD\xCE\xCF\xD0\xD1\xD2\xD3\xD4\xD5\xD6\xD7\xD8\xD9\xDA\xDB\xDC\xDD\xDE\xDF\xE0\xE1\xE2\xE3\xE4\xE5\xE6\xE7\xE8\xE9\xEA\xEB\xEC\xED\xEE\xEF\xF0\xF1\xF2\xF3\xF4\xF5\xF6\xF7\xF8\xF9\xFA\xFB\xFC\xFD\xFE\xFF]`

Create Convert Test Debug Use Library GREP Forum

Detailed Explain Token Insert Token Compare (no comparison) Export

Match a single character present in the list below

- The literal character "]"
- The literal character "["
- The literal character "^"
- The character with position 0xA9 (169 decimal) in the character set
- The character with position 0xAE (174 decimal) in the character set
- The character "B" which occupies Unicode code point U+0E3F
- The character "€" which occupies Unicode code point U+20AC
- A "whitespace character" (ASCII space, tab, line feed, carriage return, vertical tab, form feed)
- A character in the range between "A" and "F" (case sensitive)
- A character in the range between "a" and "f" (case sensitive)
- A character from the Unicode category "connector punctuation" (a punctuation character such as an underscore)
- A character from the Unicode category "math symbol" (any mathematical symbol)
- A character from the Unicode script "Braille"
- A character from the Unicode block "Thai" (U+0E00-U+0E7F)
- A character from the POSIX character class "lower" (ASCII lowercase a-z only)
- The literal character "-"
- Except any character from the Unicode category "number" (any kind of numeric character in any script)

Insert Character Class

Match one of the following characters:

Match a character that is not one of the following characters: Clear

Literal characters:

8-bit characters (hexadecimal values 00-FF):

Unicode characters (hexadecimal values 0000-FFFF):

Unicode categories:

Unicode scripts:

Unicode blocks:

POSIX classes:

Digits Tab

Word characters Carriage return

Whitespace characters Line feed

Range of characters: -

2nd range of characters: -

3rd range of characters: -

4th range of characters: -

Other character class tokens:

Intersected character class:

Subtracted character class:

OK Cancel Help

11. Insert a Regex Token to Match One Character from Predefined POSIX Classes

The Insert Token button on the Create panel makes it easy to insert the following regular expression tokens to match one character out of many possible characters. See the Insert Token help topic for more details on how to build up a regular expression via this menu.

The POSIX standard defines a number of POSIX character classes, such as “alpha” for letters and “digit” for numbers. On a fully POSIX compliant system, these classes will include letters and digits from other languages and scripts, rather than just a to z and 0 to 9.

When you select POSIX Class in the Insert Token menu, a dialog box showing all the POSIX classes appears.

At the top of the dialog box, you can select whether you want to match a character that fits one of the POSIX classes you’ll select, or one that does not fit any of the selected classes.

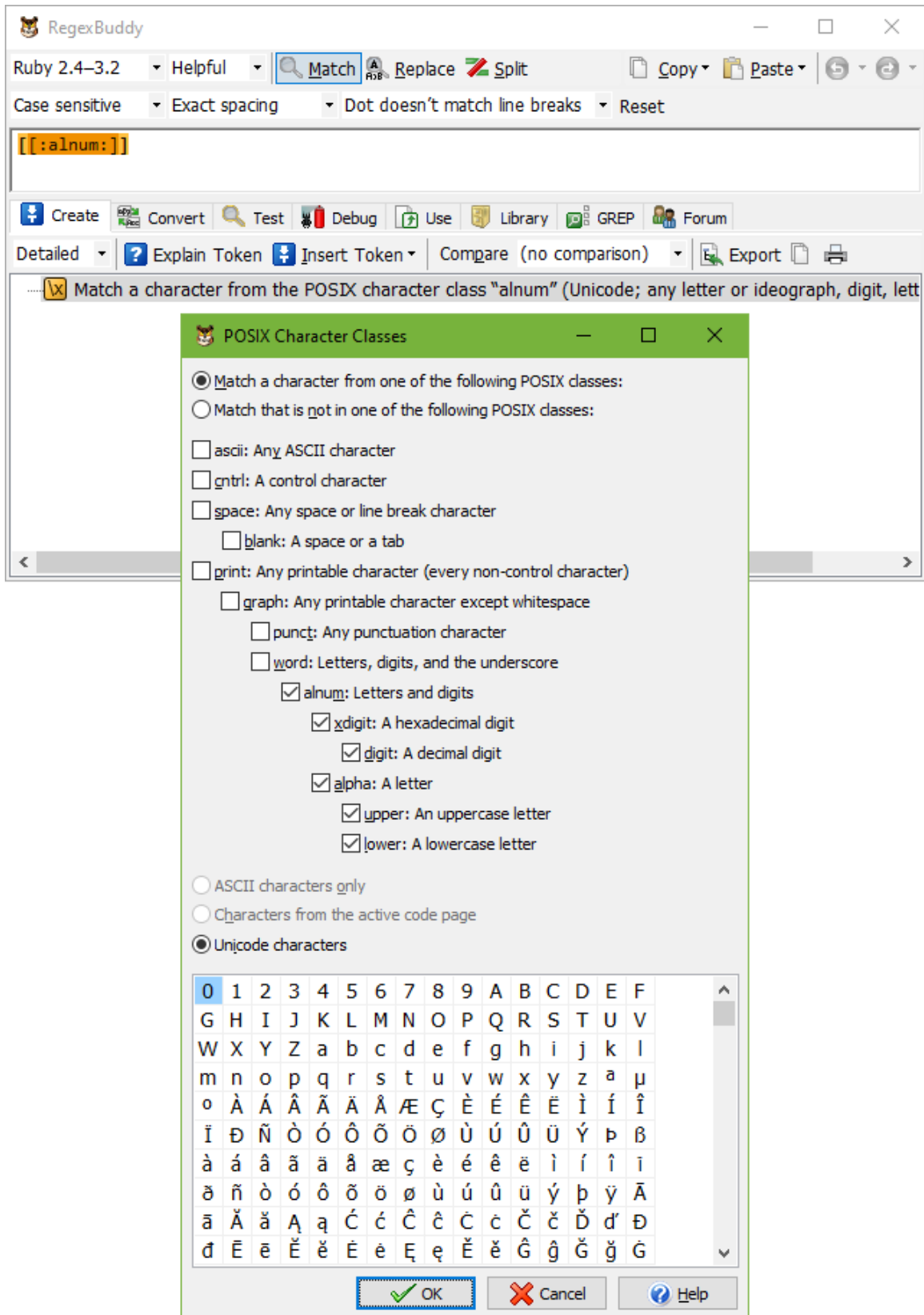
The classes are arranged in a sort of tree. If a checkbox is indented under another one, that means that the indented one is a strict subset of the one above it. E.g. all characters in the “space” class are also part of the “blank” class. When you tick a checkbox, RegexBuddy will automatically tick the wholly contained classes as well. E.g. ticking “space” will also tick “blank”. When you untick a checkbox, RegexBuddy will automatically untick all classes that contain any of the unticked class’s characters. E.g. unticking “xdigit” will also untick “alpha”, because “xdigit” includes a-f which are also part of “alpha”.

An exception to the tree structure is the “ascii” class. When a regex flavor only supports ASCII, it includes every possible character. However, most flavors support characters beyond ASCII. With those flavors, most POSIX classes will include non-ASCII characters. The “ascii” class, however, always matches one of the 128 ASCII characters. So RegexBuddy’s POSIX class dialog treats the “ascii” class separately.

Below the tree you can select whether you want the POSIX class to match only ASCII characters, or all relevant characters from the active code page, or any relevant character supported by Unicode. Most applications give you only one choice.

If your application matches only ASCII characters with POSIX classes, then the grid at the bottom shows the 128 ASCII characters. It highlights the characters included in the selected POSIX classes. Clicking or moving the mouse over the grid has no effect.

If your application matches non-ASCII characters with (some) POSIX classes, then the grid at the bottom shows the characters matched by the POSIX class or classes you have ticked. If you move the mouse over the grid, you can see the hexadecimal and decimal representations of each character’s code point in the Unicode standard. The grid will be empty if you didn’t tick any POSIX classes. The grid won’t highlight anything, and clicking it still won’t do anything.



12. Insert a Regex Token to Match at a Certain Position

The Insert Token button on the Create panel makes it easy to insert the following regular expression tokens to match at a certain position. They're called "anchors" because they essentially anchor the regular expression at the position they match. See the Insert Token help topic for more details on how to build up a regular expression via this menu.

RegexBuddy supports the following anchors. Most applications only support some of these, and there is quite a bit of variation in the syntax.

- **Beginning of the string:** `\A` or `\^` matches at the very start of the string only. If these are unsupported, `^` is inserted if “`^` match at line breaks” can be turned off.
- **End of the string:** `\Z` or `\'` matches at the very end of the string only. If these are unsupported then `^` is inserted if “`^` match at line breaks” can be turned off and the application does not allow it to match before the final line break in that mode.
- **End of the string or before the final line break:** `\Z` matches at the very end of the string, or before the line break, if any, at the very end of the string. If the application does not support `\Z` or matches at the end of the string only, then `^` is inserted if “`^` match at line breaks” can be turned off and the application allows it to match before the final line break in that mode.

You can use `\Awhatever\Z` to verify that a string consists of nothing but “whatever”.

- **Beginning of a line:** Inserts `^` and turns on “`^` match at line breaks” to make `^` match at the start of a line.
- **End of a line:** Inserts `$` and turns on “`^` match at line breaks” to make `$` match at the end of a line.

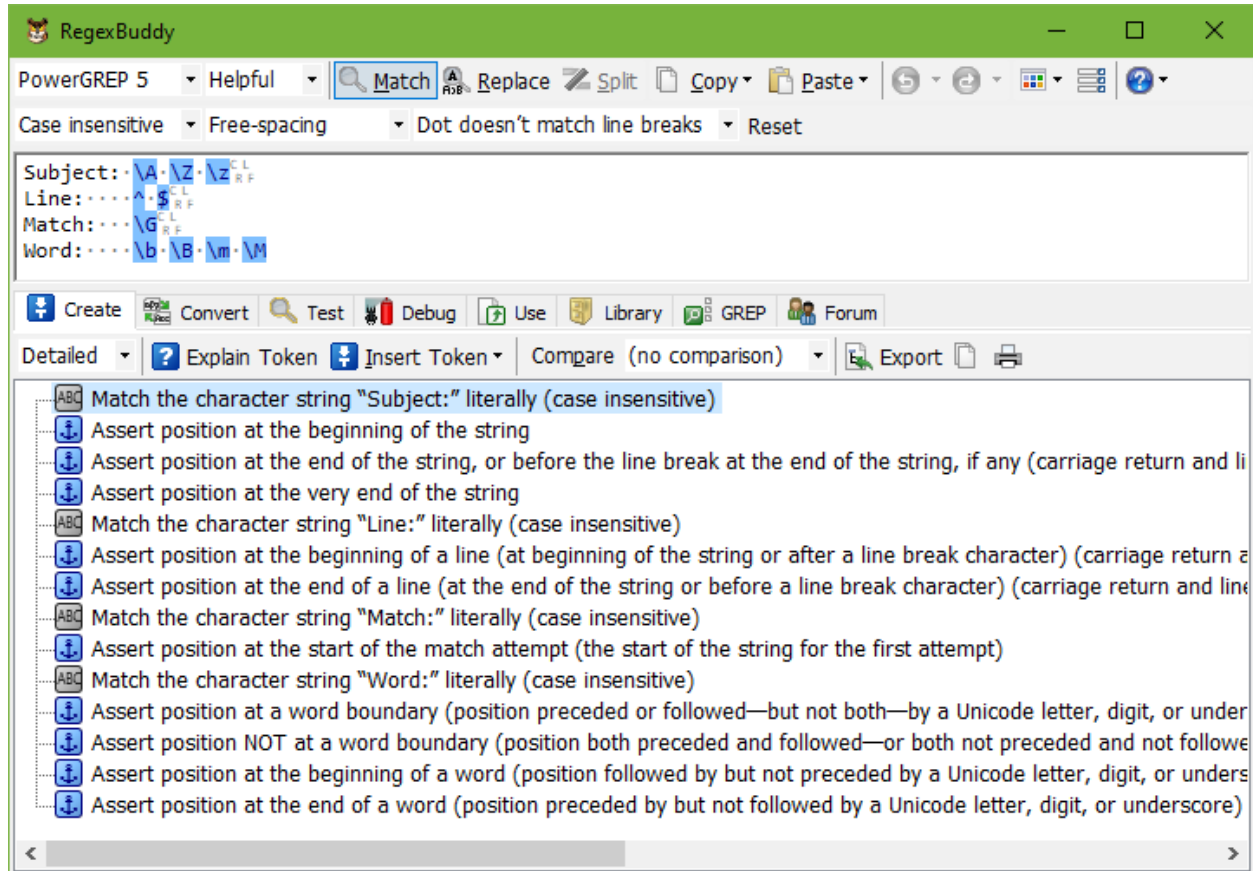
You can use `^whatever$` to match “whatever” only if it can be found all by itself on a line. Use `^.*whatever.*$` to completely match any line that has “whatever” somewhere on it.

- **Start of the match attempt:** `\G` matches at the start of the current match attempt. In some applications, `\A` and `\^` are implemented incorrectly, causing them to match at the start of any match attempt rather than just the beginning of the string. For such applications, “start of match attempt” inserts `\A` or `\^` and “beginning of the string” is grayed out.
- **End of the previous match:** `\G` matches at the end of the previous match, or at the very start of the string during the first match attempt. There is only a difference between the end of the previous match and the start of the match attempt if the previous match was a zero-length match. Then the start of the match attempt is one character beyond the end of the preceding zero-length match.

Put `\G` at the start of your regular expression if you only want to attempt it at one position, without going through the whole subject string.

- **Word boundary:** `\b` or `\y` matches between a word character and a non-word character, as well as between a word character and the start and the end of the string.
- **Non a word boundary:** `\B` or `\Y` matches between two word characters, as well as between two non-word characters. Essentially, it matches everywhere `\b` doesn't.
- **Beginning of a word:** `\m`, `\<`, or `[[[:<:]]` matches at any position followed by but not preceded by a word character.
- **End of a word:** `\M`, `\>`, or `[[[:>:]]` matches at any position preceded by but not followed by a word character.

Placing word boundaries before and after a word as in `\bword\b` or `\<word\>` is the regex equivalent of a “whole words only” search.



13. Insert a Regex Token to Repeat Another Token

The Insert Token button on the Create panel makes it easy to insert regular expression tokens to repeat other tokens in your regular expression. Repeating tokens is done with “quantifiers”. See the Insert Token help topic for more details on how to build up a regular expression via this menu.

A quantifier repeats the preceding regular expression token or group. If you specify a nonzero minimum amount, the regular expression will fail to match if the token can’t be repeated that many times. If you specify a maximum amount rather than ticking the “unlimited” checkbox, the token will be repeated at most that many times. Note that it doesn’t matter if it could have been repeated more times. The remainder of the subject text is simply ignored, or left for the remainder of the regular expression to match.

If the minimum and maximum are equal, then the regex engine will simply try to repeat the token that many times. If the maximum amount is greater than the minimum, or unlimited, then the repetition style that you choose comes into play.

The “greedy” mode is supported by all regular expression flavors. This mode first tries to match the maximum allowed, and then reduces the repetition one step at a time up to the minimum as long as the remainder of the regular expression fails to match.

The “lazy” mode is supported by many applications. It’s essentially the opposite of greedy. It matches the minimum required, and then expands the repetition step by step as long as the remainder of the regex fails to match.

The “possessive” mode is a relatively new feature only supported by a few regular expression flavors. It will try to match as many times as it can, up to the maximum, and then stop. If the remainder of the regex fails to match, it will not reduce the repetition.

If you have the subject string `aaaa` and you want to repeat the regex token `a` between 1 and unlimited times, the greedy quantifier `a+` matches `aaaa`, the lazy quantifier `a+?` matches `a` and the possessive quantifier `a++` also matches `aaaa`.

If we put a dot at the end of the regex indicating that after our 1 to unlimited `a` characters we one to match one more character, regardless of what it is, then the greedy quantifier `a+.` still matches `aaaa` (three times `a` followed by any one character). The lazy quantifier `a?.` matches `aa` (one `a` followed by any one character). The possessive quantifier `a++.`, however, fails to match. First `a++` matches all four `aaaa` characters. Then there’s nothing left for the dot to match. Unlike the greedy quantifier, the possessive one doesn’t give back.

It’s important to understand these mechanics of repeating a token and giving back. This is called backtracking. See the regular expressions tutorial for a more in-depth explanation.

The screenshot shows the RegexBuddy application interface. The main window title is "RegexBuddy". The menu bar includes "Java 4", "Helpful", "Match", "Replace", and "Split". The toolbar contains "Copy", "Paste", and navigation buttons. The main text area contains the regex pattern `a{2,7}b{3,}c++`. Below the text area is a toolbar with "Create", "Convert", "Test", "Debug", "Use", "Library", "GREP", and "Forum". The main content area is titled "Detailed" and shows the explanation of the regex pattern:

- Match the character "a" literally (case sensitive)
 - Between 2 and 7 times, as many times as possible, giving back as needed (greedy)
- Match the character "b" literally (case sensitive)
 - Between 3 and unlimited times, as few times as possible, expanding as needed (lazy)
- Match the character "c" literally (case sensitive)
 - Between one and unlimited times, as many times as possible, without giving back (possessive)

An "Insert Quantifier" dialog box is open, showing the following options:

- Minimum amount: 2
- Maximum amount: 7 Unlimited
- As many times as possible, giving back as needed (greedy)
- As few times as possible, expanding as needed (lazy)
- As many times as possible, without giving back (possessive)

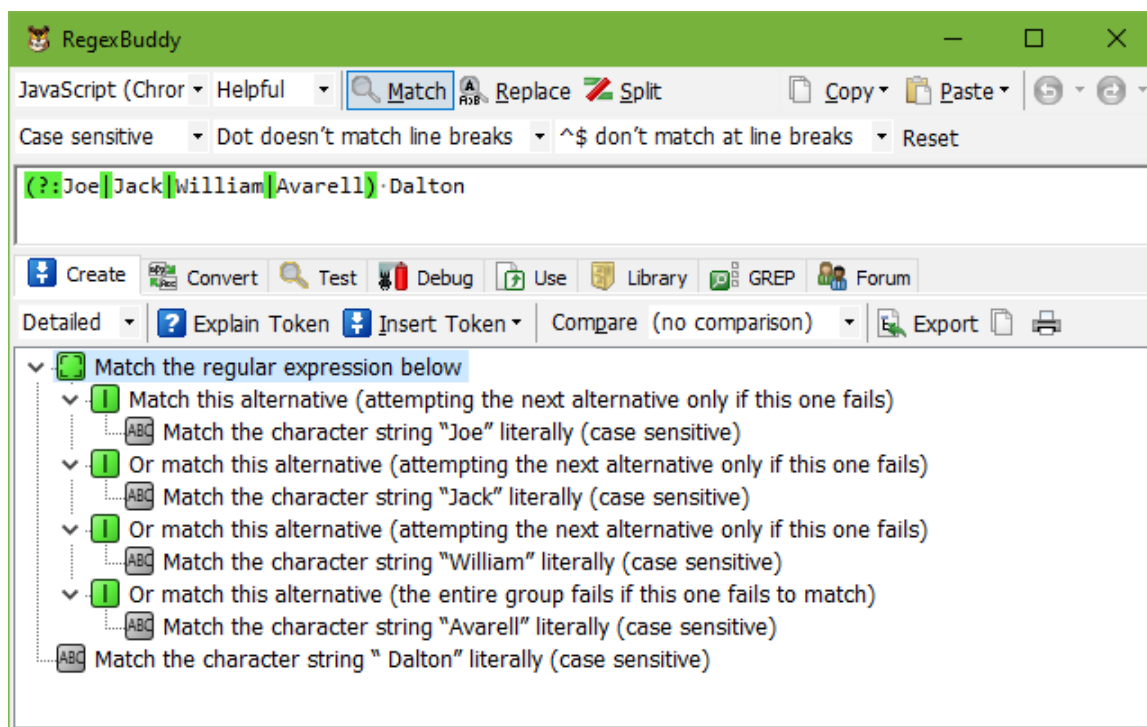
The dialog box has "OK", "Cancel", and "Help" buttons at the bottom.

14. Insert a Regex Token to Match Different Alternatives

The Insert Token button on the Create panel makes it easy to insert a regular expression token to branch between other tokens in your regular expression. Branching is done with the “alternation” operator. See the Insert Token help topic for more details on how to build up a regular expression via this menu.

Alternation causes the overall regular expression or the group (if the alternation is inserted inside a group) to match if either the part to the left of the vertical bar, or the part to the right of the vertical bar can be matched. You can insert multiple vertical bars to create more than two alternatives. `Joe|Jack|Mary` matches Joe, Jack or Mary.

If you want to alternate only part of a regular expression, you’ll need to place a group around the alternation. To match the names of the Dalton brothers, use `(?:Joe|Jack|William|Avairell) Dalton`.



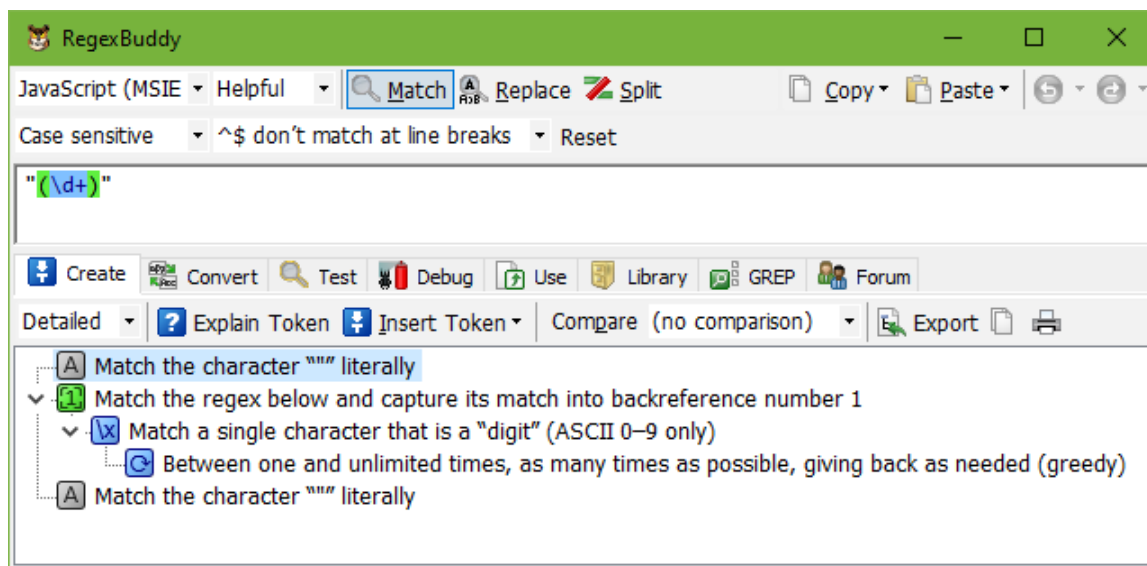
15. Insert a Capturing Group

The Insert Token button on the Create panel makes it easy to group regular expression tokens together and to capture their part of the match. Simply select the part of the regular expression you want to group, and then select the kind of group you want in the Insert Token menu. After adding numbered or named capturing groups, you can use Insert Token|Backreference to insert backreferences to those groups, which attempt to match the same text as most recently captured by the group they reference.

Numbered Capturing Group

A numbered capturing group groups the selected tokens together and stores their part of the match in a numbered backreference. Numbered capturing groups are automatically numbered from left to right, starting with number one. If you insert a quantifier after the capturing group, only the text matched by the last iteration will be captured. E.g. when `(ab)+c` matches `ababc`, the group captures `ab`. If you want to capture the text matched by all iterations, include the quantifier in the group, e.g.: `((ab)+)c`.

Most regex flavors support up to 99 capturing groups. Some support only 9.

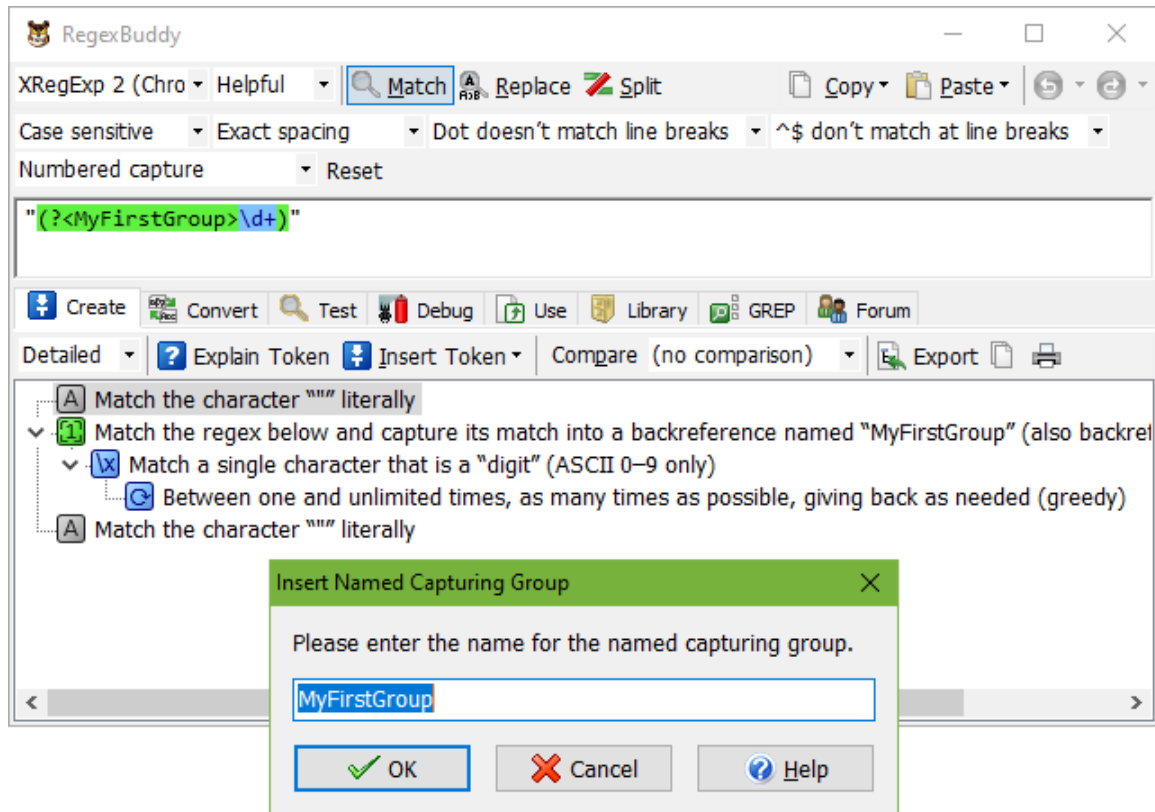


Named Capturing group

A named capturing group works just like a numbered capturing group, except that it creates a named backreference rather than a numbered one. RegexBuddy prompts you for the name of the group that you want to insert.

Mixing named and numbered groups in a single regular expression is not recommended. Some regular expression flavors number both named and numbered groups from left to right, while others don't include the named groups in the numbering. This leads to confusion about which numbered group is referenced by which numbered backreference.

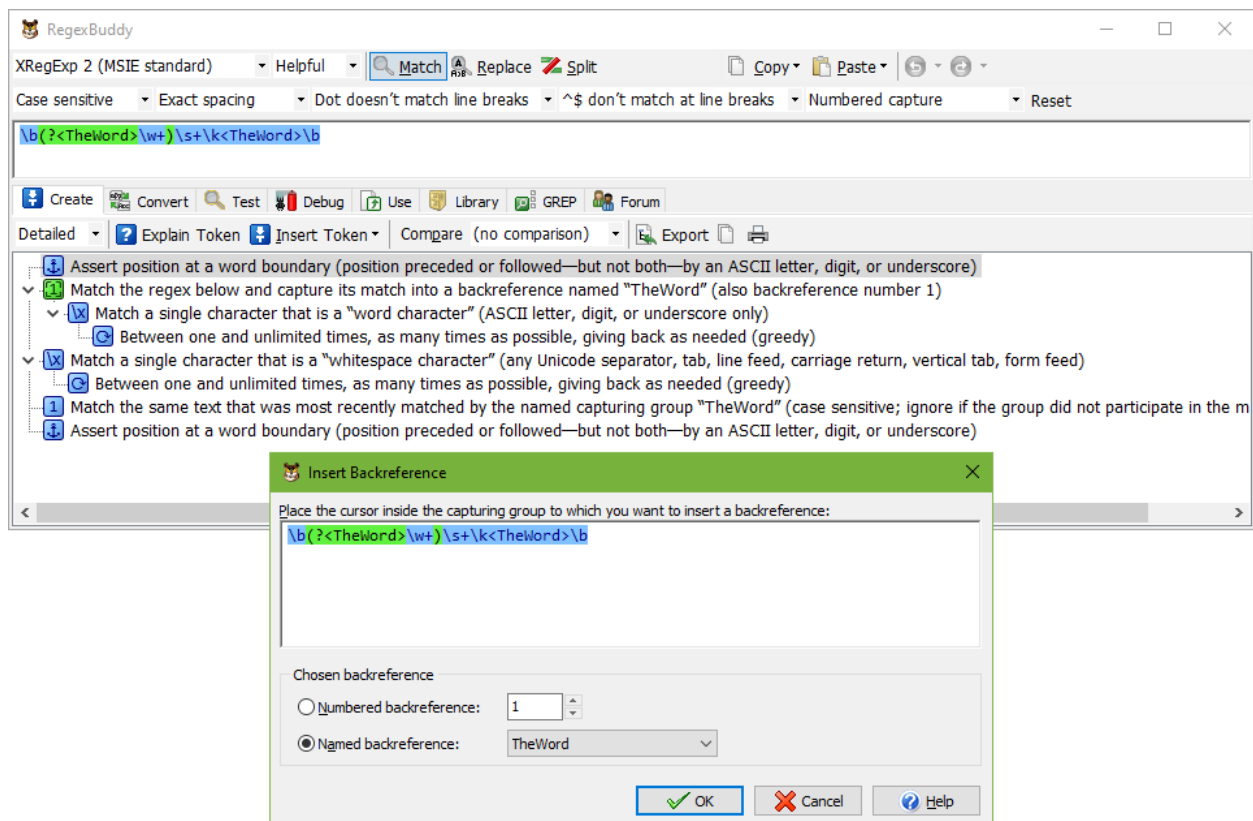
Most applications require all named groups to have unique names. Some allow duplicate names, though sometimes only as an option. When groups with duplicate names are allowed, you should only use the same name for groups that are in separate alternatives, so that at most one group with a given name will actually capture something. Applications behave quite differently when two groups with the same name participate in a match.



16. Insert a Backreference into the Regular Expression

If you've added one or more numbered or named capturing groups to your regular expression then you can insert backreferences to those groups via Insert Token|Backreference. In the window that appears, click inside the capturing group to which you want to insert a backreference. RegexBuddy automatically inserts a named backreference when you select a named group and a numbered backreference when you select a numbered group. The backreference will match the same text as most recently matched by the capturing group.

For example, the regular expression `\b(?<TheWord>w+)\s+\k<TheWord>\b` matches a word that it captures into numbered group #1. It then attempts to match one or more spaces followed by backreference #1. The backreference attempts to match the exact same word as the capturing group just matches. Essentially, this regex matches doubled words.



17. Insert a Token to Recurse into The Regex or a Capturing Group

The Insert Token button on the Create panel makes it easy to insert tokens that recurse into the whole regular expression or into a capturing group. Only a few regex engines such as Perl, PCRE, and Ruby support this.

Recursion into The Whole Regular Expression

With `(?R)` or `\g<0>` you can make your regular expression recurse into itself. The Recursion item in the Insert Token menu automatically selects the correct syntax for your application.

You'll need to make sure that your regular expression does not recurse infinitely. The recursion token must not be the first token in the regex. The regex must match at least one character before the next recursion, so that it will actually advance through the string. The regex also needs at least one alternative that does not recurse, or the recursion itself must be optional, to allow the recursion to stop at some point.

Recursion is mostly used to match balanced constructs. The regex `\([^\(\)]*(?: (?R) [^\(\)]*+)*\)` matches a pair of parentheses with all parentheses between them correctly nested, regardless of how many nested pairs there are or how deeply they are nested. This regex satisfies both requirements for valid recursion. The recursion token is preceded by `\(` which matches sure that at least one character (an opening parenthesis) is matched before the next recursion is attempted. The recursion is also optional because it is inside a group that is made optional with the quantifier `*+`.

The screenshot shows the RegxBuddy application interface. The main text area contains the regular expression: `\([^\(\)]*(?: (?R) [^\(\)]*+)*\)`. Below the text area, the 'Detailed' view is expanded, showing the following token breakdown:

- Match the opening parenthesis character
- Match any single character NOT present in the list "()"
 - Between zero and unlimited times, as many times as possible, without giving back (possessive)
- Match the regular expression below
 - Between zero and unlimited times, as many times as possible, without giving back (possessive)
 - Match the entire regular expression (recursion)
 - Match any single character NOT present in the list "()"
 - Between zero and unlimited times, as many times as possible, without giving back (possessive)
- Match the closing parenthesis character

Subroutine Calls

If you've added one or more numbered or named capturing groups to your regular expression then you can make that group recurse into itself. `(?1)` or `\g<1>` recurses into a numbered group, while `(?&name)` or `\g<name>` recurses into a named group. The regex `\A(\([^()]*+(?:(?1)[^()]*+)*+)\)\z` matches a pair of properly nested parentheses in the same way the example in the previous section does, but adds anchors to make the regex match the whole string (or not at all). The anchors need to be excluded from the recursion, which we do by adding a capturing group and limiting the recursion to the capturing group.

You can use the same syntax to insert a subroutine call to a named or numbered capturing group. `(\d+)\+(?1)=(?1)` is equivalent to `(\d+)\+(\?:\d+)\+(\?:\d+)\+` and matches something like `1+2=3`. This illustrates the key difference between a subroutine call and a backreference. A backreference matches the exact same text that was most recently matched by the group. A subroutine call reuses the part of the regex inside the group. Subroutine calls can significantly increase the readability and reduce the complexity of regular expressions that need to match the same construct (but not the exact same text) in more than one place. If we extend these two regex to match sums of floating point numbers in scientific notation, they become `([0-9]*+\.[0-9]*+|[eE][-+]?+[0-9]*+)(?1)=(?1)` and `([0-9]*+\.[0-9]*+|[eE][-+]?+[0-9]*+)\+(\?:[0-9]*+\.[0-9]*+|[eE][-+]?+[0-9]*+)\+(\?:[0-9]*+\.[0-9]*+|[eE][-+]?+[0-9]*+)\+`.

To get the correct syntax for your application, select Subroutine Call in the Insert Token menu. In the window that appears, click inside the capturing group to which you want to insert a subroutine call. RegxBuddy automatically inserts a named subroutine call when you select a named group, and a numbered subroutine call when you select a numbered group.

Different Behavior of Recursion in Different Applications

Recursion is a relatively new addition to the regular expression syntax. Even the first three popular regex engines to support it—Perl, PCRE, and Ruby—can't agree on the finer details of how recursion should behave. They've copied each other's syntax for the most part (leading to multiple syntax options for the same thing), but not their behavior. The developers of these regex engines likely didn't test enough corner cases when copying each other's features, or didn't think that these corner cases were common enough to worry about.

Fortunately for you, RegxBuddy does worry about these differences. The Insert Subroutine Call shows how the selected application behaves. The Create panel explains the exact behavior when in Detailed mode. The Test panel always correctly emulates each application's behavior.

The differences don't affect any of the examples on this page. They only use possessive quantifiers which never backtrack anyway. The regex that is recursed as a whole doesn't have any capturing groups, and the regexes with subroutine calls don't have any capturing groups inside those subroutine calls.

RegexBuddy

Perl 5.20 | Helpful | Match | Replace | Split | Copy | Paste | [Navigation icons] | [Help icon]

Case sensitive | Exact spacing | Dot doesn't match line breaks | ^\$ don't match at line breaks | Reset

`\A(\([^()]*+(?: (?1) \([^()]*+)*+))\z`

Create | Convert | Test | Debug | Use | Library | GREP | Forum

Detailed | Explain Token | Insert Token | Compare (no comparison) | Export | [Print icon]

- Assert position at the beginning of the string
- Match the regex below and capture its match into backreference number 1
 - Match the opening parenthesis character
 - Match any single character NOT present in the list "()"
 - Between zero and unlimited times, as many times as possible, without giving back (possessive)
 - Match the regular expression below
 - Between zero and unlimited times, as many times as possible, without giving back (possessive)
 - Match the regex inside capturing group number 1 (recursion; restore capturing groups upon exit)
 - Match any single character NOT present in the list "()"
 - Between zero and unlimited times, as many times as possible, without giving back (possessive)
 - Match the closing parenthesis character
- Assert position at the very end of the string

Insert Subroutine Call

Place the cursor inside the capturing group to which you want to insert a subroutine call:

`\A(\([^()]*+(?: (?1) \([^()]*+)*+))\z`

Chosen subroutine call

Numbered call:

Named call:

Backtracking if the overall regex fails after the call

Backtrack into the call, trying all permutations of the group to allow the overall regex to match

Atomic: do not backtrack into the call after it has matched, even if the overall regex fails to match

Handling of capturing groups

Do not capture the text matched by the subroutine call; groups inside the call capture normally

Capture the text matched by the subroutine call; groups inside the call capture normally

Isolate capturing groups: call does not affect capturing groups and cannot see previously captured text

Restore capturing groups: call does not affect capturing groups, but can see previously captured text

OK Cancel Help

18. Insert a Conditional into the Regular Expression

If you've added one or more numbered or named capturing groups to your regular expression then you can insert conditionals that reference those groups via Insert Token|Conditional. In the window that appears, click inside the capturing group which you want the conditional to be based on. RegxBuddy automatically inserts a named conditional when you select a named group and a numbered conditional when you select a numbered group. The inserted conditional will have two blank alternatives. You'll need to provide those to complete the conditional. The conditional will match the part to the left of the alternation operator when the capturing group has participated in the match. It will match the part to the right of the alternation operator when the capturing group has not participated in the match.

For example, the regular expression `(if)?(? $\{1\}$ then|else)` starts with an optional group that either matches and captures `if` or that does not participate in the match. The second part of the regex is a conditional that references this optional group. If the group matched `if` then the conditional tries to match `then`. If it did not participate in the match then the conditional tries to match `else`. So the only two possible matches of this regular expression are `ifthen` and `else`.

The screenshot shows the RegxBuddy interface. The main window displays the regular expression `(?:(<true>yes|no)=(<true>)yea|nay)`. The 'Insert Conditional' dialog box is open, showing the same regular expression with the cursor inside the `<true>` group. The dialog has two options: 'Numbered conditional' (set to 1) and 'Named conditional' (set to 'true').

RegxBuddy Main Window:

- Toolbar: Match, Replace, Split, Copy, Paste, Undo, Redo
- Options: Default flavor, Case sensitive, Exact spacing, Dot matches line breaks, ^\$ match at line breaks, Numbered capture, Allow zero-length matches, Reset
- Regex: `(?:(<true>yes|no)=(<true>)yea|nay)`
- Buttons: Create, Convert, Test, Debug, Use, Library, GREP, Forum
- Actions: Explain Token, Insert Token, Compare (no comparison), Export, Print
- Tree View:
 - Match the regular expression below
 - Match this alternative (attempting the next alternative only if this one fails)
 - Match the regex below and capture its match into a backreference named "true" (also backreference number 1)
 - Match the character string "yes" literally (case sensitive)
 - Or match this alternative (the entire group fails if this one fails to match)
 - Match the character string "no" literally (case sensitive)
 - Match the character "=" literally
 - Check whether named capturing group "true" matched when it was last attempted
 - If the group matched last time, then match the regular expression below
 - Match the character string "yea" literally (case sensitive)
 - If the group failed last time, then match the regular expression below
 - Match the character string "nay" literally (case sensitive)

Insert Conditional Dialog:

- Place the cursor inside the capturing group that you want to be checked by the conditional:
- Regex: `(?:(<true>yes|no)=(<true>)yea|nay)`
- Chosen conditional:
 - Numbered conditional: 1
 - Named conditional: true
- Buttons: OK, Cancel, Help

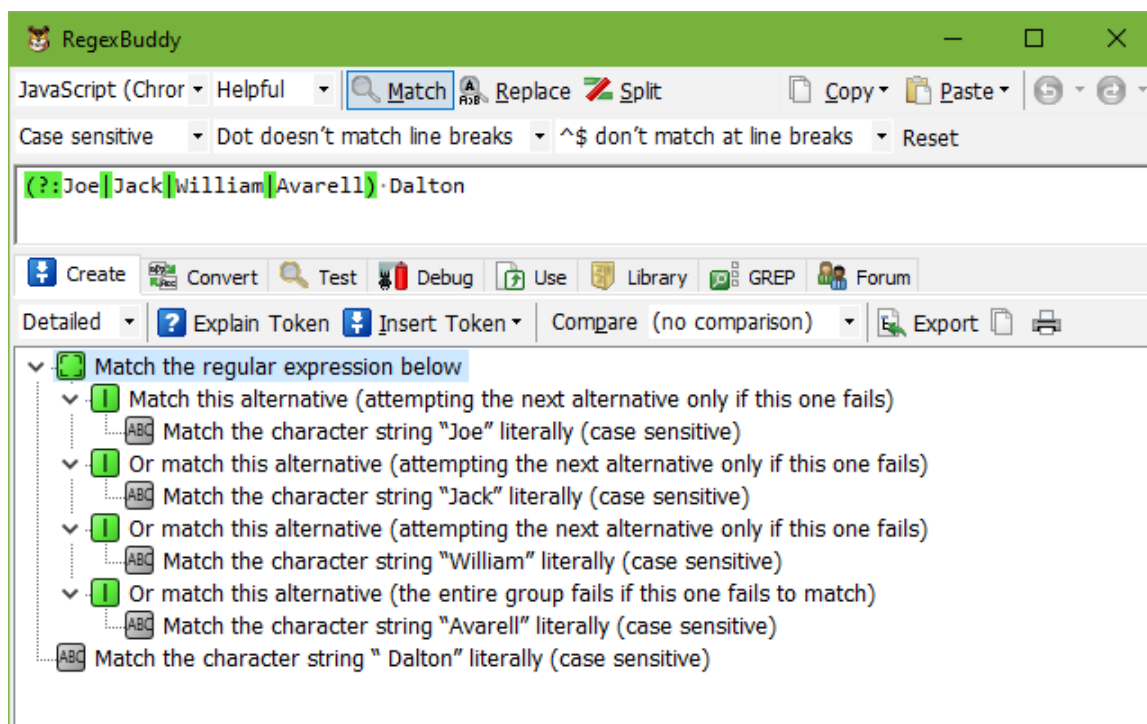
19. Insert a Grouping Regex Token

The Insert Token button on the Create panel makes it easy to group regular expression tokens together. Simply select the part of the regular expression you want to group, and then select the kind of group you want in the Insert Token menu.

Non-Capturing group

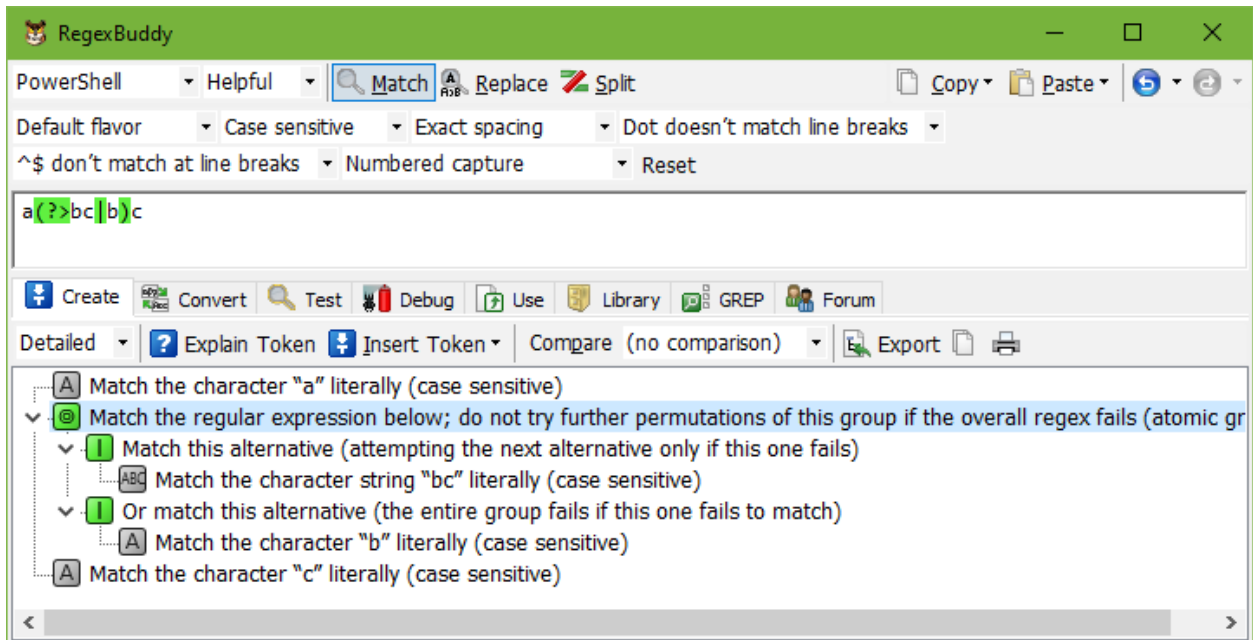
A non-capturing group groups the selected tokens together, so you can insert a quantifier behind it that will apply to the entire group. Non-capturing groups are also handy when you want to alternate only part of a regular expression. As its name indicates, a non-capturing group does not capture anything. It's merely for grouping. The benefit is that you can freely mix non-capturing groups with numbered capturing groups in a regular expression, without upsetting the backreference numbers of the capturing groups.

Some regex flavors do not support non-capturing groups. In that case, you'll need to use a capturing group instead.



Atomic Group

An atomic group is a non-capturing group that is atomic or indivisible. Once an atomic group has matched, the regex engine will not try different permutations of it at the same starting position in the subject string. You can use it to optimize your regular expression and to prevent catastrophic backtracking.

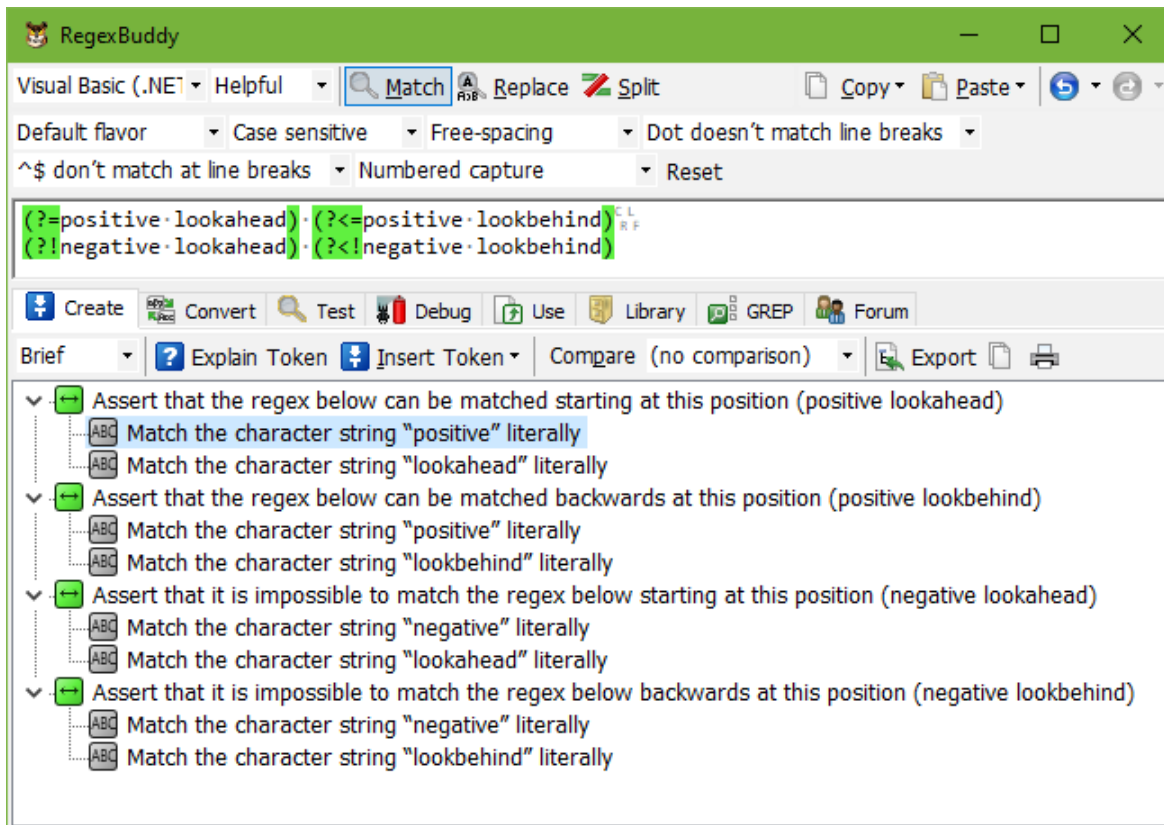


The screenshot shows the RegxBuddy application interface. The main text area contains the regular expression `a(?:bc|b)c`. Below the text area is a toolbar with buttons for 'Create', 'Convert', 'Test', 'Debug', 'Use', 'Library', 'GREP', and 'Forum'. Below the toolbar is a menu bar with options: 'Detailed', 'Explain Token', 'Insert Token', 'Compare (no comparison)', and 'Export'. The main content area displays a detailed explanation of the regex tokens:

- Match the character "a" literally (case sensitive)
- Match the regular expression below; do not try further permutations of this group if the overall regex fails (atomic group)
 - Match this alternative (attempting the next alternative only if this one fails)
 - Match the character string "bc" literally (case sensitive)
 - Or match this alternative (the entire group fails if this one fails to match)
 - Match the character "b" literally (case sensitive)
- Match the character "c" literally (case sensitive)

20. Insert Lookaround

Like anchors, lookaround groups match at a certain position rather than certain text. Lookahead will try to look forward at the current position in the string, while lookbehind will try to look backward. If the regex tokens inside the group can be matched at that position, positive lookahead will succeed, and negative lookahead will fail. If the regex tokens cannot be matched, positive lookahead fails and negative lookahead succeeds.



Lookaround is mainly used to check if something occurs before or after the match that you're interested in, without including that something in the regex match.

- Positive lookahead: Succeeds if the regular expression inside the lookahead can be matched starting at the current position.
- Negative lookahead: Succeeds if the regular expression inside the lookahead can NOT be matched starting at the current position.
- Positive lookbehind: Succeeds if the regular expression can be matched ending at the current position (i.e. to the left of it).
- Negative lookbehind: Succeeds if the regular expression can NOT be matched ending at the current position (i.e. to the left of it).

Since regular expressions normally cannot be applied backwards, most applications only allow you to use fixed-length regular expressions inside lookbehind. Some regex flavors don't allow any quantifiers, while others allow quantifiers as long as they're not "unlimited". The JGsoft and .NET regex flavors are the only ones that allow full regular expressions inside lookbehind.

21. Insert a Regex Token to Change a Matching Mode

The Insert Token button on the Create panel makes it easy to insert the following regular expression tokens to change how the regular expression engine applies your regular expression. These tokens are called mode modifiers.

The screenshot shows the RegexBuddy application interface. The main window displays a list of regex tokens and their corresponding matching options. The 'Insert Token' dialog box is open, showing the following options:

- Match the regex below with these options
 - Case insensitive
 - ABC Match the character string "case_insensitive" literally
- Match the regex below with these options
 - Case sensitive
 - ABC Match the character string "case_sensitive" literally
- Match the regex below with these options
 - Dot matches line breaks
 - Match any single character
 - ABC Match the character string "matches_line_breaks" literally
- Match the regex below with these options
 - Dot doesn't match line breaks
 - Match any single character that is NOT a line break character
 - ABC Match the character string "doesn't_match_line_breaks" literally
- Match the regex below with these options
 - ^\$ match at line breaks
 - Assert position at the beginning of a line
 - Assert position at the end of a line
 - ABC Match the character string "match_at_line_breaks" literally
- Match the regex below with these options
 - ^\$ don't match at line breaks
 - Assert position at the beginning of the string
 - Assert position at the end of the string, or before the line break at the end of the string, if any
 - ABC Match the character string "don't_match_at_line_breaks" literally
- Match the regex below with these options
 - Free-spacing
 - ABC Match the character string "free-spacing" literally
- Match the regex below with these options
 - Exact spacing
 - ABC Match the character string "exact spacing" literally

Mode modifiers are useful in situations where you can't set overall matching modes like you can with the combo boxes on RegexBuddy's toolbar. Mode modifiers are not supported by all applications that support matching modes. But in applications that do, mode modifiers always override modes set outside of the regex (combo boxes in RegexBuddy).

- **Turn on case insensitive:** Differences between uppercase and lowercase characters are ignored. `cat` matches `CAT`, `CaT`, or `cAt` or any other capitalization in addition to `cat`.
- **Turn on free-spacing:** Unescaped spaces and line breaks in the regex are ignored so you can use them to format your regex to make it more readable. In most applications this mode also makes `#` the start of a comment that runs until the end of the line.
- **Turn on dot matches line breaks:** The dot matches absolutely any character, whether it is a line break character or not. Sometimes this option is called "single line mode".
- **Turn on ^\$ match at line breaks:** The `^` and `$` anchors match after and before line breaks, or at the start and the end of each line in the subject string. Which characters are line break characters depends on the application and the line break mode. Sometimes this option is called "multi-line mode".

Some regular expression flavors also have mode modifiers to turn off modes, even though all modes are off by default. These flavors allow you to place mode modifiers in the middle of a regex. The modifier will then apply to the remainder of the regex to the right of the modifier, turning its mode on or off. With these flavors, if you select part of your regex before choosing a mode modifier item in the Insert Token menu, RegexBuddy will create a mode modifier span that sets the mode for the selected part of the regex only.

- **Turn off case insensitive:** Differences between uppercase and lowercase characters are significant. `cat` matches only `cat`. Same as selecting "case sensitive" in the combo boxes.
- **Turn off free-spacing:** Unescaped spaces, line breaks, and `#` characters in the regex are treated as literal characters that the regex must match. Same as selecting "exact spacing" in the combo boxes.
- **Turn off dot matches line breaks:** The dot matches any character that is not a line break character. Which characters are line break characters depends on the application and the line break mode. Same as selecting "dot doesn't match line breaks" in the combo boxes.
- **Turn off ^\$ match at line breaks:** The `^` and `$` anchors only match at the start and the end of the whole subject string. Depending on the application, `$` may still match before a line break at the very end of the string. Same as selecting "^\$ don't match at line breaks" in the combo boxes.

22. Insert a Comment

Click on the Insert Token button on the Create panel and select Comment to add a comment to your regular expression. You can use comments to explain what your regular expression does. Comments are ignored by the regex engine.

If you want to add a lot of comments to your regular expression, you should use turn on the free-spacing option if your regex flavor supports it. In free-spacing mode, spaces and line breaks are ignored, so you can lay out your regular expression and the comments freely.

The screenshot shows the RegexBuddy application interface. The main window displays a regular expression with several comments: `# Match a 20th or 21st century date in yyyy-mm-dd format`, `# year (group 1)`, `# separator`, `# month (group 2)`, `# separator`, and `# day (group 3)`. The expression is: `((19|20)\d\d) [./] (0|[1-9])1[012] [./] (0|[1-9])|[12][0-9]|3[01])`. The interface includes a menu bar with options like Match, Replace, Split, Copy, and Paste, and a toolbar with buttons for Create, Convert, Test, Debug, Use, Library, GREP, and Forum. A History panel on the right shows a previous search for "Date yyyy-mm-dd". The bottom panel, titled "Detailed", provides a tree view of the regex components, including comments and detailed descriptions of each token, such as "Match the character string '19' literally" and "Match a single character that is a 'digit' (ASCII 0-9 only)".

23. Using RegexMagic with RegexBuddy

RegexMagic is another product from Just Great Software. Just like RegexBuddy, it is designed to make it easy to create regular expressions. The key difference is that with RegexBuddy, you work directly with the regular expression syntax, or plain English building blocks that correspond directly with the regex syntax. With RegexMagic, you don't deal with the regular expression syntax at all. Instead you use RegexMagic's powerful patterns for matching characters, numbers, dates, times, email addresses, and much more to specify what you want, and RegexMagic generates the regular expression for you.

Suppose you want to match a number between 256 and 512. In RegexMagic, you simply select the "integer" pattern. In the pattern's properties, you set the minimum and maximum values to 256 and 512. RegexMagic automatically generates the regular expression `51[0-2][50[0-9]][34][0-9]{2}2[6-9][0-9]25[6-9]`. This regex, though conceptually simple, would be quite a chore to create by hand, even for regular expression experts.

If you own both RegexBuddy and RegexMagic, you can use the RegexMagic button on the Create panel in RegexBuddy to generate (part of) your regular expression with RegexMagic. The generated regex is automatically transferred to RegexBuddy so you can further edit and test it. The RegexMagic button offers three choices.

Click the **Generate New Regex** item under the RegexMagic button on the Create panel to launch RegexMagic. The Samples panel will have one sample with the text from the Test panel in RegexBuddy. The Match and Action panels will be blank. When you click the Send button on the Regex panel in RegexMagic, the regular expression in RegexBuddy, if any, will be replaced with the one you generated in RegexMagic. Nothing will happen if you close RegexMagic without clicking the Send button.

Click the **Regenerate Regex** item to launch RegexMagic with the same settings on the Samples, Match, and Action panels as you had them when you last clicked the Send button in RegexMagic after choosing Generate New Regex or Regenerate Regex in RegexBuddy. RegexMagic will not take over any changes you may have made to the previously generated regular expression in RegexBuddy. While you can edit any regular expression with RegexBuddy, RegexMagic can only edit regular expressions that it generated by itself, and for which you have saved the settings in a RegexMagic library. When you click the Send button in RegexMagic, the regular expression in RegexBuddy, if any, will be replaced with the one you generated in RegexMagic.

Click the **Insert Regex** item to launch RegexMagic. The first time you use this command, the Match and Action panels in RegexMagic will be blank. After that, the Match and Action panels will be as you left them last time you clicked the Send button in RegexMagic after choosing the Insert Regex command. When you click the Send button in RegexMagic, the regular expression generated by RegexMagic is inserted into the regular expression you have in RegexBuddy.

At the bottom of the Insert Token menu you'll also see a RegexMagic item. This item works exactly the same as the Insert Regex item in the RegexMagic menu.

24. Insert a Replacement Text Token












RegexBuddy makes it easy to build replacement texts without having to remember every detail of the complex replacement text syntax.

To insert a token into the replacement text when defining a Replace action, right-click in the editor box for the replacement text at the spot where you want to insert the token. This will move the cursor to that position, and show the context menu. In the context menu, select Insert Token.

When analyzing a replacement text on the Create panel, you can easily designate the spot where you want to insert the token. Click on a token in the replacement text tree, and the new token will be inserted right after it. Click on the Insert Token button on the toolbar or press Alt+I on the keyboard to access the Insert Token menu. The Insert Token button and Alt+I alternate between showing regular expression tokens and replacement text tokens depending on which of the editor boxes for the regular expression and replacement text or which of the trees on the Create panel most recently had keyboard focus.

List of Replacement Text Tokens

The Insert Token menu offers the following items. Note that depending on the replacement text flavor that you're working with, certain items may not be available, or may insert different tokens into the replacement text. Some replacement flavors don't offer certain features, or use a different syntax. The Subject left of match item, for example, is often disabled because many applications don't support this feature. Perl and Ruby do, but with different syntax. For Perl this item inserts `$'` while for Ruby it inserts `\'`.

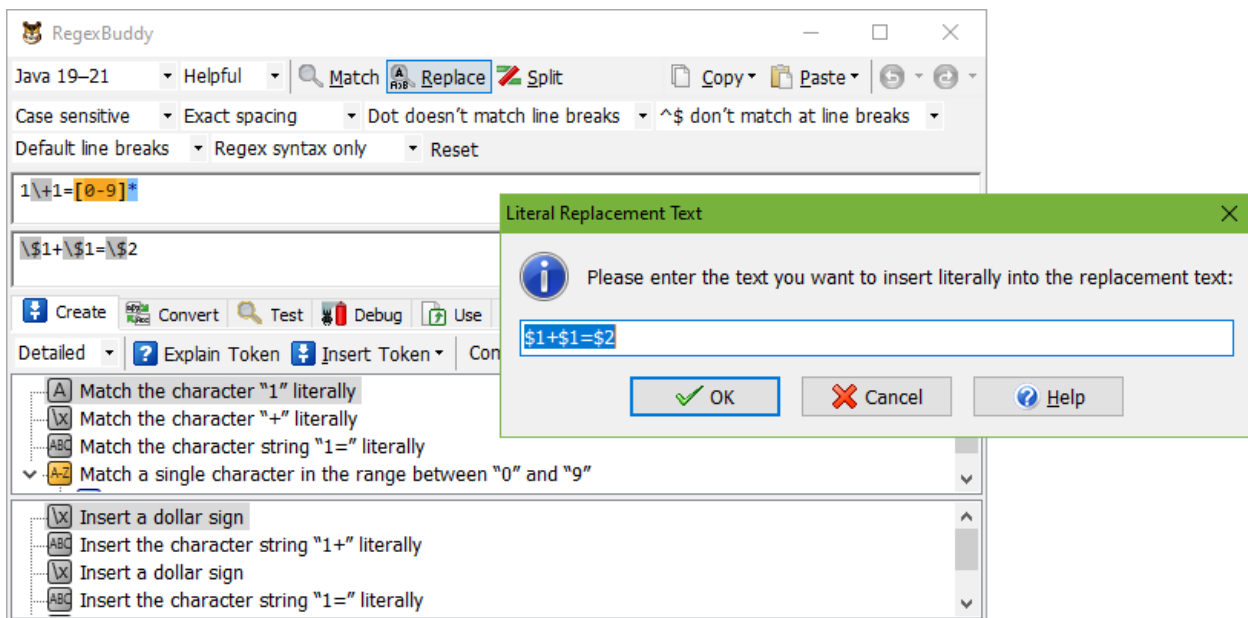
-  Literal text
-  Non-printable character
-  8-bit character
-  Unicode character
-  Matched Text
-  Backreference
-  Last Backreference
-  Conditional
-  Subject Left of Match
-  Subject Right of Match
-  Whole Subject

25. Insert Specific Characters into The Replacement Text

The Insert Token button on the Create panel makes it easy to insert the following replacement text tokens that represent specific characters. See the Insert Token help topic for more details on how to build up a replacement text via this menu.

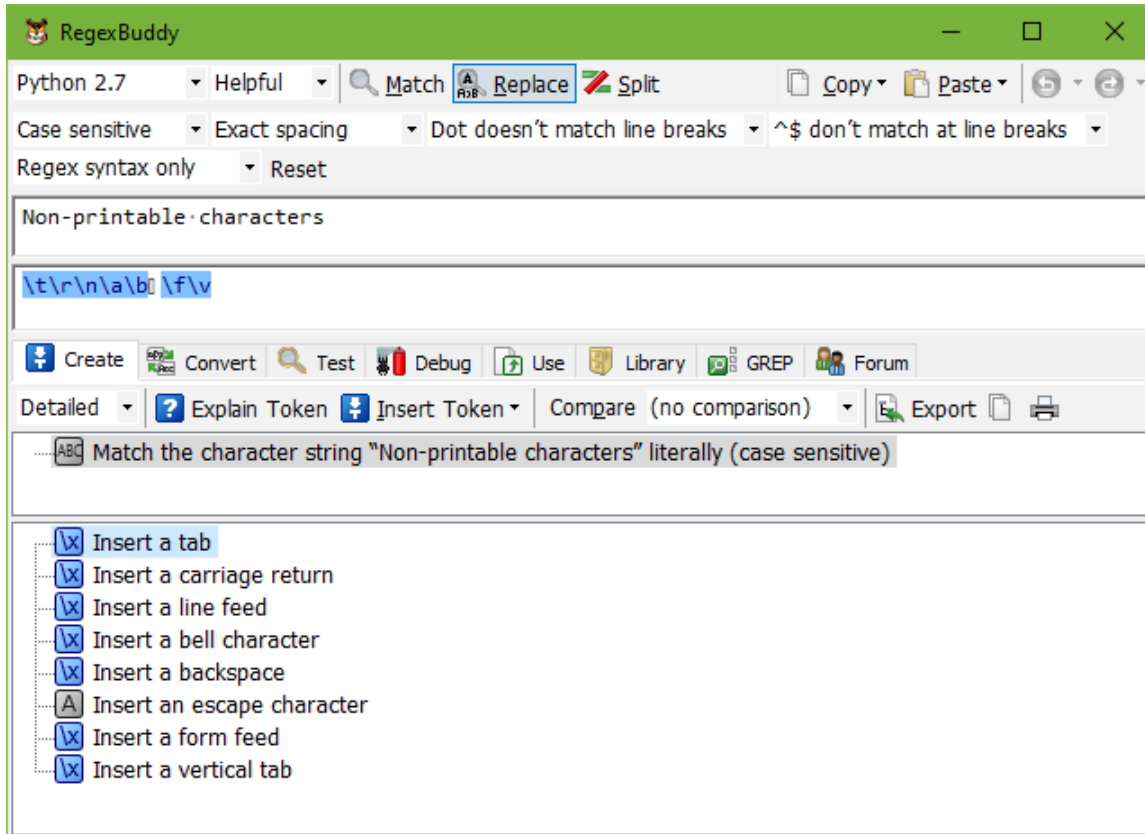
Literal Text

Enter one or more characters that will be inserted literally into the replacement text. RegxBuddy will escape backslashes, dollar signs, and possibly other characters if the selected application requires that.



Non-Printable Character

Match a specific non-printable character, such as a tab, line feed, carriage return, alert (bell), backspace, escape, form feed, or vertical tab. If the selected application supports any kind of escape sequence that represents the character you want to insert, then RegxBuddy inserts that escape sequence. Otherwise, RegxBuddy inserts the non-printable character directly.



8-bit Character

Inserts a specific character from an 8-bit code page. Use this to insert characters that you cannot type on your keyboard when working with an application or programming language that does not support Unicode.

In the screen that appears, first select the code page or encoding that you will be working with in the application where you'll implement your regular expression. The code pages labeled "Windows" are the Windows "ANSI" code pages. The default code page will be the code page you're using on the Test panel, if that is an 8-bit code page. To properly test your regular expression, you'll need to select the same code page on the Test panel as you used when inserting 8-bit characters into your replacement text.

RegexBuddy shows you a grid of all available characters in that code page. Click on the character you want to insert, and click OK.

RegexBuddy inserts a single hexadecimal character escape in the form of `\xFF` into your replacement text to match the character you selected. If your replacement text flavor does not support hexadecimal escapes, RegexBuddy inserts the characters literally.

RegexBuddy

Perl 5.8 Helpful Match Replace Split Copy Paste 5 6

`\xA9`

`\xAE`

Create Convert Test Debug Use Library GREP Forum

Detailed Explain Token Insert Token Compare (no comparison) Export

Match the character with position 0xA9 (169 decimal) in the character set

Insert the character with position 0xAE (174 decimal) in the character set

8-bit Character Map

8-bit code page:
Windows 1252: Western European

Match a single character
 Match one character out of a list of characters

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		☺	☻	☼	☽	☾	☿	♁	♂	♀	♃	♄	♅	♆	♇	♈
1	▶	◀	↑	↓	↶	↷	↸	↹	↺	↻	↼	↽	↾	↿	↰	↱
2	!	"	#	\$	%	&	'	()	*	+	,	-	.	/	
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	☐
8	€	×	,	f	"	...	†	‡	^	%	Š	<	Œ	×	Ž	×
9	×	'	'	"	"	•	—	—	~	™	š	>	œ	×	ž	ÿ
A		ı	ç	£	¥	ı	§	"	©	a	«	¬	-	®	¯	
B	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

OK Cancel Help

Unicode Character

Inserts a specific Unicode character or Unicode code point. Use this to insert characters that you cannot type on your keyboard when working with an application or programming language that supports Unicode.

In the screen that appears, RegexBuddy shows a grid with all available Unicode characters. Since the Unicode character set is very large, this can be a bit unwieldy. If you know what Unicode category the character you want belongs to, select it from the drop-down list at the top to see only characters of that category. If you move the mouse over the grid, you can see the hexadecimal and decimal representations of each character's code point in the Unicode standard.

If you see a great number of squares instead of characters in the grid, click the Select Font button to change the grid's font. The squares indicate the font cannot display the character. With the "all code points" character map option selected, certain squares will be crossed out with thin gray lines. These squares indicate unassigned Unicode code points. These are reserved by the Unicode standard for future expansion. With any other character map option selected, the last row of the grid may have squares that are crossed out with thin gray lines. This simply indicates the selected category doesn't have any more characters to fill up the last row.

RegexBuddy inserts a single Unicode character escape in the form of `\uFFFF` or `\x{FFFF}` into your replacement text to insert the character you selected. If your replacement text flavor does not support Unicode escapes, RegexBuddy inserts the characters literally.

RegxBuddy

Perl 5.30–5.32 | Helpful | Match | Replace | Split | Copy | Paste | [Refresh] | [Undo]

Case sensitive | Exact spacing | Dot doesn't match line breaks | ^\$ don't match at line breaks

Numbered capture | Reset

`\x{153}`

`\x{1E3}`

Create | Convert | Test | Debug | Use | Library | GREP | Forum

Detailed | Explain Token | Insert Token | Compare (no comparison) | Export | Print

Match the character "œ" which occupies Unicode code point U+0153 (case sensitive)

Insert the character "æ" which occupies Unicode code point U+01E3

Unicode Character Map

Match a single character:
 Match one character out of a list of characters

Character map: lowercase letters | Select Font

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p
q	r	s	t	u	v	w	x	y	z	ª	µ	°	β	à	á
â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	ð	ñ
ò	ó	ô	õ	ö	ø	ù	ú	û	ü	ý	þ	ÿ	ā	ă	ą
ć	ĉ	ċ	č	ď	đ	ē	ě	é	ẹ	ë	ĝ	ğ	ġ	ĥ	
ħ	ĩ	ī	ĵ	ĳ	ĵ	ķ	κ	ĺ	ļ	ł	ł	ł	ł	ł	ł
ŋ	ñ	ň	ŋ	ō	ö	ö	œ	ř	ŕ	ř	ś	ŝ	ş	š	ţ
ţ	ţ	ü	ü	ü	ü	ü	ü	w	ŷ	z	z	z	z	z	z
b	ċ	đ	q	f	h	k	ł	λ	η	σ	τ	β	ε	ı	ı
f	u	ý	z	z	z	z	z	p	dž	lj	nj	ă	ĩ	õ	ü
ü	ú	ü	ù	ə	ā	ā	æ	g	ğ	ķ	q	q̇	ž	ĵ	dz
g	ñ	á	æ	ó	ă	â	ē	ê	ĩ	î	ð	ó	ř	f	ü
ú	ş	ţ	ş	ħ	đ	ŝ	z	á	ę	õ	õ	ó	õ	ý	ı
ŋ	ł	e	o	o	b	o	o	o	o	o	o	o	o	o	o
e	ı	g	g	o	Y	Y	Y	Y	ı	ı	ı	ı	ı	ı	ı
ß	w	u	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı
r	ı	R	B	ş	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı

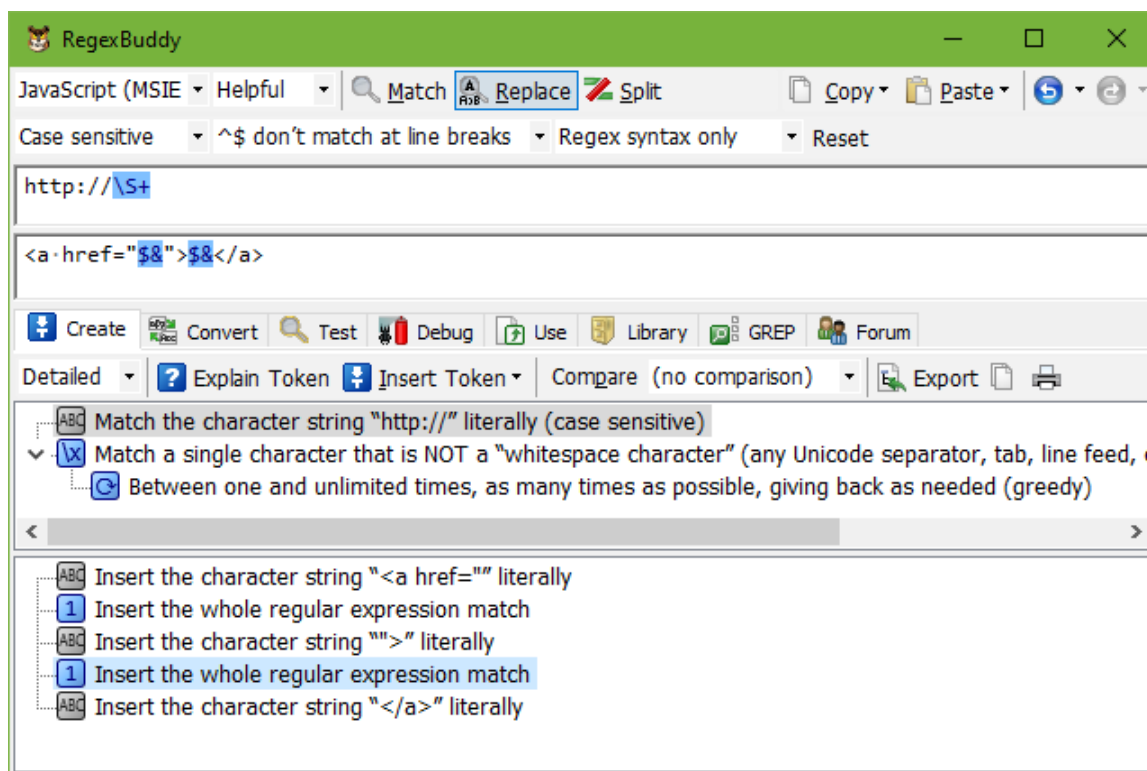
OK | Cancel | Help

26. Insert a Backreference into the Replacement Text

The Insert Token button on the Create panel makes it easy to insert the following replacement text tokens that reinsert (part of) the regular expression match. See the Insert Token help topic for more details on how to build up a replacement text via this menu.

Matched Text

Inserts a token such as `$&` or `$0` into the replacement text that will be substituted with the overall regex match. When using a regex that matches a URL, for example, the replacement text `$&` will turn the URL into an anchor tag.

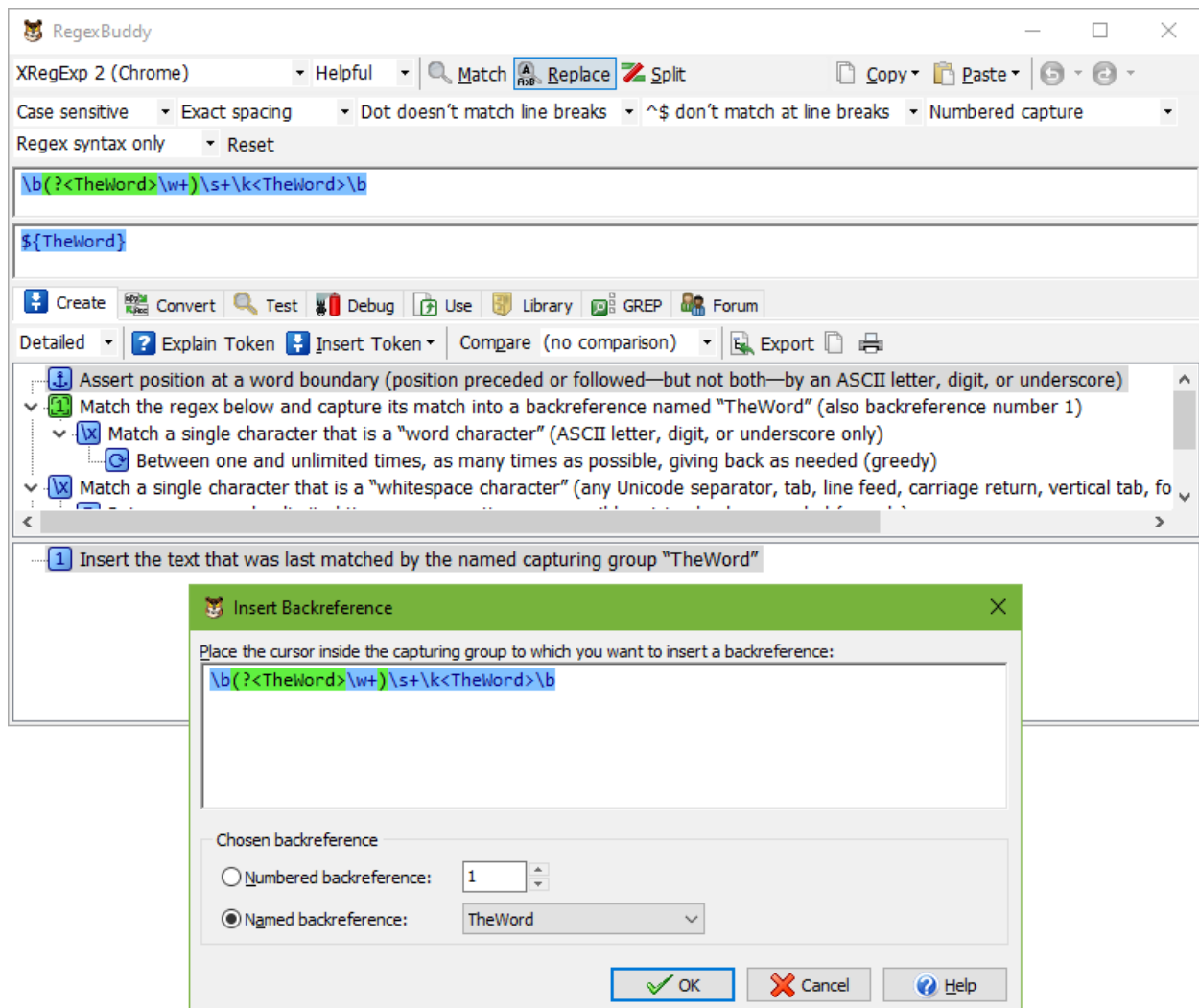


Backreference

If you've added one or more numbered or named capturing groups to your regular expression then you can insert backreferences to those groups via Insert Token|Backreference. In the window that appears, click inside the capturing group to which you want to insert a backreference. RegxBuddy automatically inserts a named backreference when you select a named group, and a numbered backreference when you select a numbered group, using the correct syntax for the selected application.

The backreference will be substituted with the text matched by the capturing group when the replacement is made. If the capturing group matched multiple times, perhaps because it has a quantifier, then only the text last matched by that group is inserted into the replacement text.

For example, the regular expression `\b(<TheWord>\w+)\s+\1\b` matches a doubled word and captures that word into numbered group #1. A search-and-replace using this regex and `$1` or `\1` as the replacement text will replace all doubled words with a single instance of the same word.



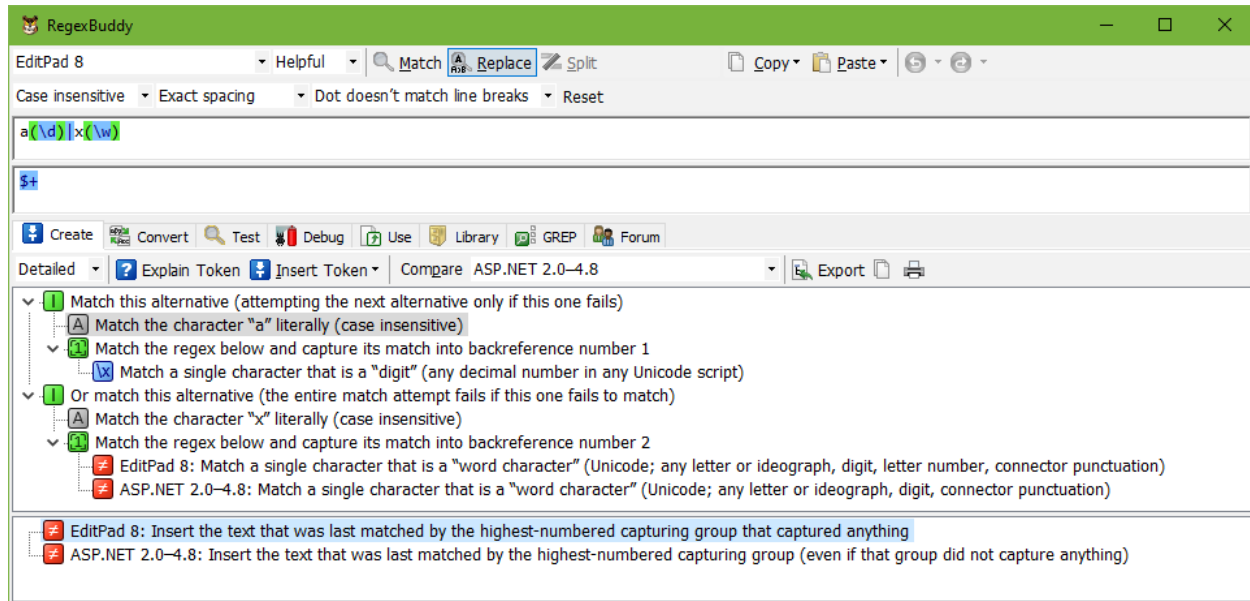
Last Backreference

Some flavors support the `$+` or `\+` token to insert the text matched by highest-numbered capturing group into the replacement text. Unfortunately, it doesn't have the same meaning in all applications that support it. In some applications, it represents the text matched by the highest-numbered capturing group that actually participated in the match. In other applications, it represents the highest-numbered capturing group, whether it participated in the match or not.

For example, in the regex `a(\d)|x(\w)` the highest-numbered capturing group is the second one. When this regex matches `a4`, the first capturing group matches `4`, while the second group doesn't participate in the match attempt at all. In some applications, such as EditPad or Ruby, `$+` or `\+` will hold the `4` matched by the first capturing group, which is the highest-numbered group that actually participated in the match. In other

applications, such as .NET, $\$+$ will be substituted with nothing, since the highest-numbered group in the regex didn't capture anything. When the same regex matches `xy`, EditPad, Ruby, and .NET all store `y` in $\$+$.

Also note that .NET numbers named capturing groups after all non-named groups. This means that in .NET, $\$+$ will always be substituted with the text matched by the last named group in the regex, whether it is followed by non-named groups or not, and whether it actually participated in the match or not. This is yet another reason why we recommend not mixing named and numbered capturing groups in a single regex.



27. Insert a Conditional into the Replacement Text

The Insert Token button on the Create panel makes it easy to insert the following replacement text tokens that reinsert (part of) the regular expression match. See the Insert Token help topic for more details on how to build up a replacement text via this menu.

Conditional

If you've added one or more numbered or named capturing groups to your regular expression then you can insert conditionals that reference those groups via Insert Token|Conditional. In the window that appears, click inside the capturing group which you want the conditional to be based on. RegexBuddy automatically inserts a named conditional when you select a named group and a numbered conditional when you select a numbered group.

The inserted conditional will have two blank alternatives. You'll need to provide those to complete the conditional. The conditional will insert the part to the left of the alternation operator into the replacement when the capturing group has participated in the match. It will insert the part to the right of the alternation operator when the capturing group has not participated in the match.

For example, the regular expression `0|(1)` matches `0` or `1`. The capturing group only participates in the match when the regex matches `1`. The replacement string conditional `(?{1>true:false)` thus inserts `true` when the regex matches `1` and `false` when it matches `0`.

RegexBuddy

boost::wregex 1.66–1.77 Helpful Match Replace Split Copy Paste ↶ ↷

Default flavor All flavor Case sensitive Exact spacing Dot matches line breaks

^\$ match at line breaks Numbered capture Allow zero-length matches Reset

(?:(<true>yes|no))

(?{true}yea|nay)

Create Convert Test Debug Use Library GREP Forum

Detailed Explain Token Insert Token Compare (no comparison) Export Print

- Match the regular expression below
 - Match this alternative (attempting the next alternative only if this one fails)
 - Match the regex below and capture its match into a backreference named "true" (also backreference number 1)
 - Match the character string "yes" literally (case sensitive)
 - Or match this alternative (the entire group fails if this one fails to match)
 - Match the character string "no" literally (case sensitive)
- Check whether named capturing group "true" was matched
 - If the group was matched then insert the following
 - Insert the character string "yea" literally
 - If the group was not matched then insert the following
 - Insert the character string "nay" literally

Insert Conditional

Place the cursor inside the capturing group that you want to be checked by the conditional:

(?:(<true>yes|no))

Chosen conditional

Numbered conditional: 1

Named conditional: true

OK Cancel Help

28. Insert The Subject String into The Replacement Text

The Insert Token button on the Create panel makes it easy to insert the following replacement text tokens that reinsert (part of) the subject string. See the Insert Token help topic for more details on how to build up a replacement text via this menu.

The three subject string tokens are only supported by a few replacement text flavors. They're generally not very useful in search-and-replace operations. However, they can be quite handy when collecting or extracting text based on regex matches.

Subject Left of Match

Insert a token into the replacement text that will be substituted with the text in the subject string to the left of the regular expression match. For example, when `\d+` matches `123` in `abc123def`, then the part of the subject to the left of the match is `abc`.

Subject Right of Match

Insert a token into the replacement text that will be substituted with the text in the subject string to the right of the regular expression match. For example, when `\d+` matches `123` in `abc123def`, then the part of the subject to the right of the match is `def`.

Whole Subject

Insert a token into the replacement text that will be substituted with the whole string that you applied the regular expression to, regardless of which part of it the regular expression actually matched.

The screenshot shows the RegxBuddy application window. The title bar reads "RegxBuddy". The menu bar includes "PowerGREP 5", "Helpful", "Match", "Replace", and "Split". Below the menu bar are options for "Case insensitive", "Exact spacing", and "Dot doesn't match line breaks", along with a "Reset" button. The main input field contains the regex pattern `\d+`. Below this, the tokenized result is shown as `Left:·$;·Right:·$';·Whole:·$`. A toolbar contains buttons for "Create", "Convert", "Test", "Debug", "Use", "Library", "GREP", and "Forum". Below the toolbar, there are options for "Detailed", "Explain Token", "Insert Token", "Compare (no comparison)", and "Export". The main content area shows a tree view of the regex tokens:

- Match a single character that is a "digit" (any decimal number in any Unicode script)
 - Between one and unlimited times, as many times as possible, giving back as needed (greedy)
- Insert the character string "Left" literally
- Insert a colon
- Insert the character "·" literally
- Insert the part of the subject string to the left of the regex match
- Insert the character string "; Right" literally
- Insert a colon
- Insert the character "·" literally
- Insert the part of the subject string to the right of the regex match
- Insert the character string "; Whole" literally
- Insert a colon
- Insert the character "·" literally
- Insert the whole subject string

29. Analyze and Edit a Regular Expression

RegexBuddy takes away all the obscurity and confusion caused by the rather cryptic regular expression syntax when you click on the Create panel.

When you enter a regular expression or paste in an existing regex, the Create panel shows you a tree outline of the regular expression, called the “regex tree”. The regex tree describes the regular expression in plain English, using a structure that you can easily navigate.

The small drop-down list at the left hand of the Create toolbar allows you to switch the regex tree between brief and detailed mode. In brief mode, the regex tree tells you the basic purpose of each part of the regular expression. In detailed mode, the regex tree provides additional detail about the current application’s specific interpretation. For example, when `\w` is explained as matching a word character, brief mode will tell you nothing further, while detailed mode will also tell you which characters the current application treats as word characters. Brief mode makes it easier to grasp the structure of a complex regex by reducing the amount of information thrown at you. Detailed mode gives you a full understanding by explaining many subtle details that you might not be aware of.

Click on a node in the regex tree, and RegexBuddy highlights the corresponding token in the regular expression. While you edit the regular expression, RegexBuddy keeps the regex tree synchronized with the updated regular expression. As you move the text cursor while editing, RegexBuddy highlights the node in the tree that describes the token around or immediately to the left of the text cursor.

To get more information about a particular node, click on the node and then click the Explain Token button. This opens the regex tutorial in RegexBuddy’s help file, right at the page that describes the node you selected. When the regex tree or the edit box for the regex has keyboard focus, you can also press F1 on the keyboard to open the regex tutorial at the page that describes the selected node or the regex token under the cursor.

After clicking on a node, you can press the Delete key on the keyboard to delete it, or the Insert key to insert a regex token. You can rearrange parts of the regular expression by dragging and dropping nodes with the mouse. Note that moving a node this way actually moves the part of the regular expression that the node represents, rather than the node itself. The regex tree is rebuilt after you moved the node. This can have side effects. E.g. if you move a quantifier to the very start of the regular expression, the node turns from a quantifier node into an error node.

You can edit certain nodes by double-clicking on them. If you double-click the node for a character class, for example, then the window for inserting a character class appears. The window shows the elements in the character class that you double-clicked on. When you click OK, the character class is replaced with the edited one.

Not all tokens can be edited by double-clicking. If you double-click on the regex tree node for a dot, for example, nothing happens. The dot simply matches any character and there’s nothing you can edit about it, other than to delete it or replace it with something totally different.

If some of the syntax used in your regular expression is not supported by the application you selected, then the regex tree shows errors that the selected application doesn’t support such and such. If you double-click on the flavor error node, RegexBuddy automatically replaces it with equivalent syntax that is supported by the selected flavor. This way you can manually convert a regex from one flavor to another. If no equivalent syntax is supported, double-clicking the flavor error node does nothing. Automatic conversion of the entire regex is possible on the Convert panel.

RegexBuddy gives you full freedom to edit the regular expression. It does not get into your way by preventing the regular expression from becoming invalid. Instead, RegexBuddy assumes you know what you are doing, and helps you with that by clearly analyzing your regular expression, whatever state it is in.

If your regular expression will be used in multiple applications or multiple versions of the same application or programming language, use the Compare drop-down list on the Create panel. This expands the regex tree to cover multiple applications, allowing you to instantly compare the behavior of a regex across multiple applications without having to run lots of tests.

The regex tree is not only handy to analyze a regular expression while editing it. The regex tree is perfect for explaining to others how a regular expression works. You can include the regex tree in your documentation or teaching materials by clicking the Export button. You can include it in the source code of software you're developing via the Use panel.

The screenshot shows the RegexBuddy application window. The main text area contains the following regular expression and its breakdown:

```
# Match a 20th or 21st century date in yyyy-mm-dd format
((?:19|20)\d\d).....# year (group 1)
[-/.].....# separator
(0[1-9]|1[012]).....# month (group 2)
[-/.].....# separator
(0[1-9]|[12][0-9]|3[01])...# day (group 3)
```

The interface includes a menu bar with options like Match, Replace, Split, Copy, Paste, and a toolbar with buttons for Create, Convert, Test, Debug, Use, Library, GREP, and Forum. A History panel on the right shows a previous search for "Date yyyy-mm-dd". The bottom section displays a detailed tree structure for the regex, explaining each token and its function, such as "Match the regex below and capture its match into backreference number 1" and "Match this alternative (attempting the next alternative only if this one fails)".

Analyze and Edit a Replacement Text

If you put RegexBuddy in Replace mode, then the Create panel is split into two parts. The top part shows the regex tree as usual. The bottom part uses a very similar structure to describe the replacement text in plain English. You can use it to analyze, edit, and compare the replacement text in the same way as you can use the regex tree for the regular expression. The Insert Token menu switches to show replacement text tokens whenever you click on the replacement text tree. The menu switches back to show regex tokens when you click on the regex tree.

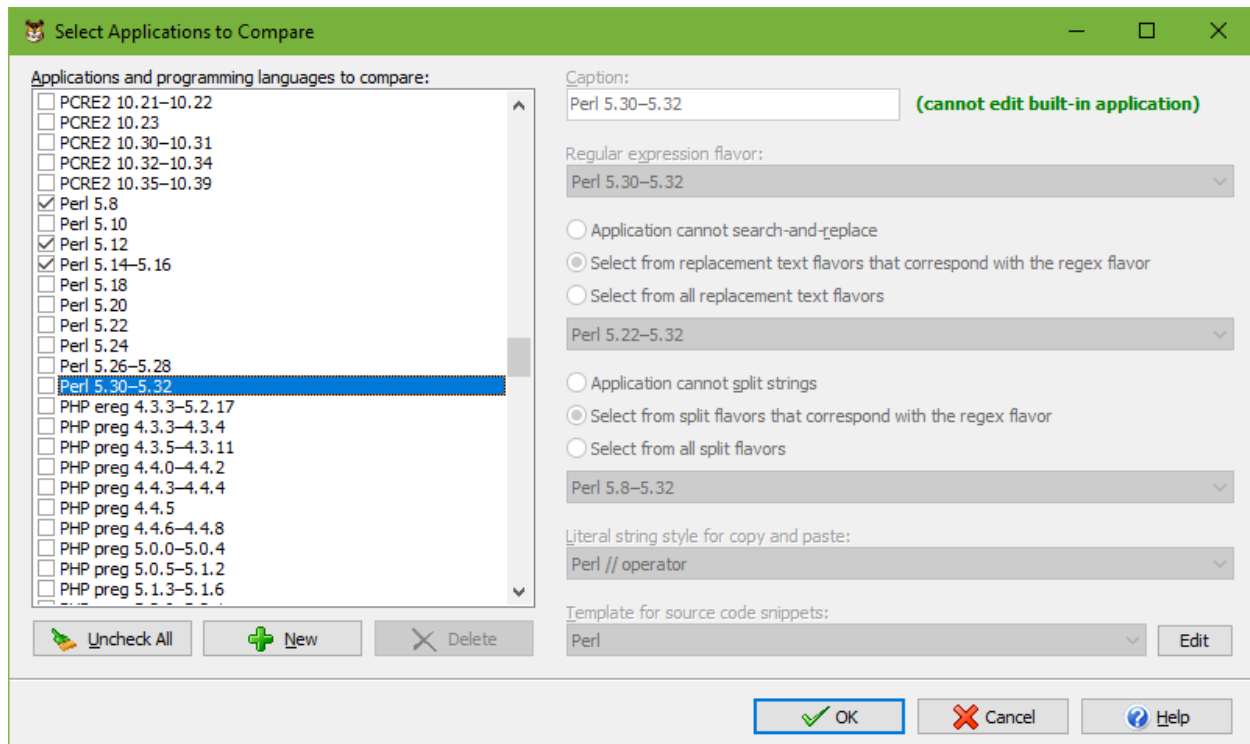
30. Compare a Regex Between Multiple (Versions of) Applications


RegexBuddy has the unique ability to emulate the features and limitations of all the popular tools and languages for working with regular expressions, including many different versions of the same applications and programming languages. Many of the differences that RegexBuddy is aware of are quite subtle. You wouldn't necessarily discover them just by testing your regular expressions in your actual applications, because many subtle differences only reveal themselves with specific test subjects. Therefore, you should always check RegexBuddy's analysis on the Create panel, so you'll know what to test for.


There may be times when you need a regular expression that works with multiple applications, or multiple versions of the same application. If you're developing a code library that targets multiple versions of a programming language, then any regexes your code uses need to work the same way in all those versions. If you're writing an article on how to use regexes to solve a particular problem in a particular application, then you'll want to alert your readers if different versions of that application may produce different results.


With RegexBuddy you can easily deal with this. RegexBuddy can instantly compare your regular expressions between any number of applications.

To choose the applications you want to compare, select "More applications and languages" in the Compare drop-down list on the Create panel, or press Ctrl+F3 on the keyboard. The dialog box that pops up is very similar to the one you used for selecting the (primary) target application. The difference is that the checkboxes in the list of applications now allow you to select the applications you want to compare, instead of marking favorite applications. You can select as many applications to compare as you like in any combination. You can select multiple versions of the same application, or totally different applications, or a combination of different versions of different applications.



After you clicking OK, RegexBuddy will immediately compare your regular expression (and replacement text, if any) among all the selected applications. Parts of the regular expression that are handled in the same way by all applications are added normally to the regex tree with their usual icons. In the screen shot below, that includes the comments and `\p{Ll}`. If certain parts of the expression are handled differently, the regex tree will show multiple  nodes for those parts. There will be one node for each unique interpretation of that part of the regex, listing all the applications that follow that interpretation. The screen shot shows that Perl 5.12 and prior handled the “lowercase letter” and “uppercase letter” Unicode categories inconsistently in case insensitive regexes.

The regex tree on the create panel will indicate exactly which parts of the regex are handled differently by some of the selected applications. If a group node in the tree is handled differently, then the comparison adds additional group nodes with the  icon. These nodes restrict the comparison of their part of the regex to certain applications. If you click on such a node, RegexBuddy selects the node’s part of the regular expression. In the screen shot, you can see that Perl 5.8 does not support named capture, but that later versions do. The nodes below the Perl 5.14–5.16||Perl 5.10||Perl 5.12 group node node are added with their usual icons because these versions of Perl handle the named capturing group and its content in exactly the same way. Double-clicking on regex tree nodes does not have any effect if the regex tree indicates any differences.

If the entire regular expression or replacement text is handled in the same way by all applications you’ve chosen to compare, then  nodes at the top of the regex tree or replacement tree will tell you so. Then double-clicking tree nodes then works as usual.

Switching the regex tree between brief and detailed mode affects the outcome of the comparison. Suppose you’re comparing two different applications that both support `\w` to match word characters, but one supports only ASCII while the other supports Unicode. In brief mode, the regex tree will tell you that all selected applications *basically* interpret the regex in the same way, with an added note that detailed mode may reveal subtle differences. In detailed mode, the regex tree will show that the two applications handle `\w` differently.

When creating a regex that needs to work in multiple applications, you can initially work in brief mode to make sure there aren’t any major syntactic differences between the two applications. If there aren’t, you can then switch to detailed mode to make sure there aren’t any (subtle) differences in behavior.

If there are too many incompatibilities between the applications that your regex needs to work with, then you may need to create multiple regular expressions. RegexBuddy’s Convert panel can help you with that.

RegexBuddy

Perl 5.30-5.32

Helpful Match Replace Split Copy Paste

Case insensitive Exact spacing Dot doesn't match line breaks ^\$ don't match at line breaks Numbered capture Reset

`(?#LL)\p{L1}(?#Lu)\p{Lu}(?#^LL)\p{^L1}(?#^Lu)\p{^Lu}{<named>group}`

History

Date yyyy-mm-dd

Create Convert Test Debug Use Library GREP Forum

Detailed Explain Token Insert Token Compare Compare 4 applications Export RegexMagic

- Comment: Ll
 - Match a character from the Unicode category "letter with case" (a letter that exists in lowercase and uppercase variants)
- Comment: Lu
 - Perl 5.30-5.32 & Perl 5.14-5.16: Match a character from the Unicode category "letter with case" (a letter that exists in lowercase and uppercase variants)
 - Perl 5.8 & Perl 5.12: Match a character from the Unicode category "uppercase letter" (an uppercase letter that has a lowercase variant)
- Comment: ^L
 - Perl 5.30-5.32 & Perl 5.14-5.16: Match any character that is NOT in the Unicode category "letter with case" (a letter that exists in lowercase and uppercase variants)
 - Perl 5.8 & Perl 5.12: Match any character that is NOT in the Unicode category "lowercase letter" (a lowercase letter that has an uppercase variant)
- Comment: ^Lu
 - Perl 5.30-5.32 & Perl 5.14-5.16: Match any character that is NOT in the Unicode category "letter with case" (a letter that exists in lowercase and uppercase variants)
 - Perl 5.8 & Perl 5.12 does not correctly handle negating the "Lu" Unicode category when the regex is case insensitive; `\P{Lu}` will match any character
 - Perl 5.30-5.32 & Perl 5.12 & Perl 5.14-5.16 interprets this part differently
 - Match the regex below and capture its match into a backreference named "named" (also backreference number 1)
 - Match the character string "group" literally (case insensitive)
 - Perl 5.8 interprets this part differently
 - Perl 5.8 does not support named capture
 - Match the character string "group" literally (case insensitive)

Perl 5.30-5.32 & Perl 5.8 & Perl 5.12 & Perl 5.14-5.16

All selected applications handle your replacement text in the same way

Convert all text until the next `\L` or `\E` to uppercase

Insert the whole regular expression match

31. Export The Analysis of a Regular Expression

The regex tree RegexBuddy shows on the Create panel is perfect for explaining to others how a regular expression works. You can include the regex tree in your documentation or teaching materials by clicking the Export button. A small menu will pop up below the button, giving you the choice between exporting to a plain text file, exporting to an HTML file, and exporting to the clipboard.

Export to a Plain Text File

When you export the regex tree to a plain text file, RegexBuddy will ask you for the name of the file, and for a caption. The caption is placed at the top of the text file, followed by the regular expression and then the regex tree. Each node in the tree will occupy one line in the text file, indented with spaces. When viewing the text file in a text editor, turn off word wrap to clearly see the tree structure.

Export to an HTML File

When you export the regex tree to an HTML file, RegexBuddy will ask you for the name of the file, and for a caption. The caption is used for the title and header tags in the HTML file. You can also choose whether you want the HTML file to link to <https://www.regular-expressions.info> or not. If you do, each node in the regex tree will link to the page in the tutorial on this web site that explains the node. If you plan to upload the HTML file to your web site, you should definitely include the links. This makes it much easier for your site's visitors to learn more about regular expressions. The tutorial at <https://www.regular-expressions.info> is the same tutorial as the one in RegexBuddy's documentation. It was written by Jan Goyvaerts, who also designed and developed RegexBuddy.

When you view the HTML file exported by RegexBuddy, you will see the regular expression at the top of the page, and the regex tree as a bulleted list below it. Moving the mouse pointer over the regular expression or the regex tree highlights the corresponding part of the regular expression in the regex tree. This makes it very convenient to grasp which part of the regex does what, just like clicking on regex tree nodes does on the Create panel in RegexBuddy.

Export to Clipboard

Exporting the regex tree to the clipboard generates the same textual representation as exporting to a plain text file does. The text is placed on the clipboard instead of saved into a file.

If you plan to include the regex tree as documentation in software source code, you can generate a comment with the regex tree on the Use panel instead of exporting it on the Create panel. The Use panel will generate a source code snippet that you can paste into your source code directly.

32. Convert a Regex to Work with Another Application

Exactly what your regular expression does depends a lot on the application or programming language you'll use it in. A regular expression that works perfectly in one application may find different matches or not work at all in another application. You should never paste a regex that you've found on the Internet into your own code and hope that it'll all work.

Instead, paste the regular expression into RegexBuddy. If the regex was formatted as a string in source code, paste with the correct string style. Select the application or programming language that the regular expression was originally intended for in the toolbar at the top. This way, you can be sure that RegexBuddy is interpreting the regular expression as it was intended and (hopefully) tested by its original creator.

Now you can easily convert the regular expression to the application or programming language you want to use it with. On the Convert panel, select your target application in the drop-down list with the mouse or by pressing Alt+V on the keyboard. Your favorite applications are shown directly in the list. Choose "More applications and languages" in the list or press Ctrl+F4 to select another application or to change which applications are your favorites. The conversion is instant and fully automatic. The converted regex is updated continuously if you edit the original regex. In Replace mode, both the regular expression and replacement text are converted.

RegexBuddy is very strict when converting regular expressions. You can see this in the screen shot. Both .NET and Delphi support the dot to match any character except line breaks. But the .NET Regex class only treats `\n` as a line break, while Delphi's TRegex class recognizes all Unicode line breaks. So RegexBuddy converts the dot into a negated character class `[^\n]` that does exactly the same thing in Delphi as `.` does in C#.

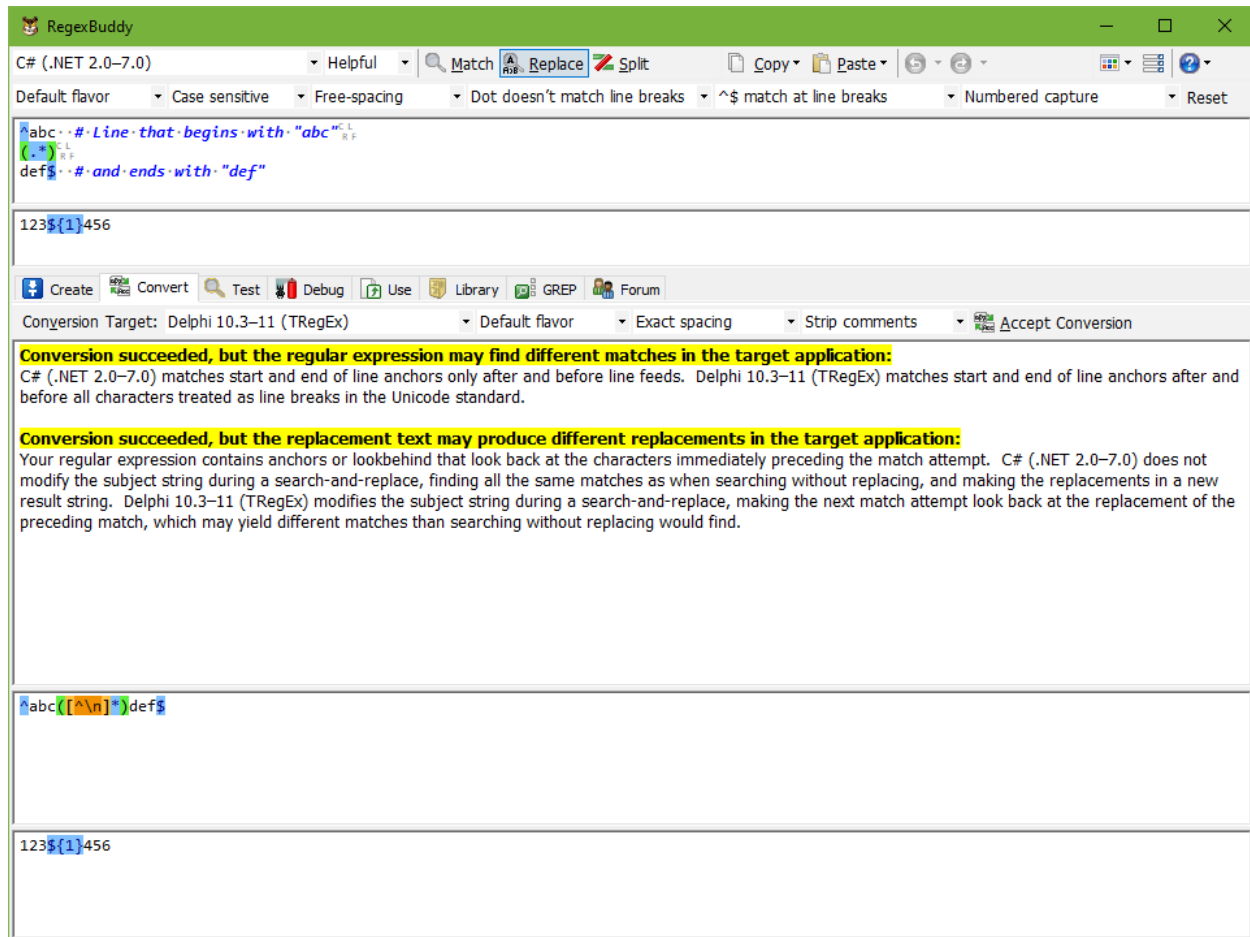
But RegexBuddy can't always use alternative syntax to preserve the exact behavior of the regex. Both .NET's Regex class and Delphi's TRegex class use `^` and `$` to match at the start and the end of a line. But again the interpretation of what constitutes a line is different. In this case, the converted regex uses the same syntax, and RegexBuddy warns that the behavior of the regex may be different. Whether it will be different depends on the strings or files you'll be using this regex on. If they don't contain any line break characters other than `\n`, then you can ignore the warning.

The top half of the Convert panel shows all the warnings and errors, if any. The bottom half shows the converted regex. In Replace mode, the bottom half is split into two once more to show both the converted regex and replacement.

To use the converted regular expression and replacement text, click the Accept Conversion button or press Alt+A. This replaces the original regex with the converted regex. If you want to keep the original regex, first click the green plus button on the History panel to duplicate the original regex, and then click Accept Convert to replace the duplicate regex with the converted regex. Either way, the Convert panel becomes empty to give you extra visual confirmation that you accepted the conversion. As soon as you edit the regex

You should always use the Accept Conversion button. Do not copy and paste the converted regex or replacement directly into your target application. First of all, accepting the conversion allows you to use the Copy button or the Use panel to correctly format the regular expression for pasting into your source code. But more importantly, the regular expression conversion also handles the regular expression options. The target application may require different options to be set. These options aren't shown on the Convert panel. But the Accept Conversion button does change the regex options on the top toolbar as needed.

For example, converting the same regex shown in the screen shot from C# to Java will say “conversion succeeded” without any warnings or any changes to the regex. But the converted regex will only do the same thing in Java as it did in C# when setting the UNIX_LINES option in Java. If you click Accept Conversion, then RegxBuddy automatically selects “LF only” in the drop-down list for the line break handling regex option. Then the Use panel will automatically add UNIX_LINES to the Java snippets it generates.



Converting Whitespace and Comments

If the target application supports both free-spacing and exact spacing modes, then you can choose which mode the converted regular expression should use. Converting a free-spacing regex to exact spacing will strip all insignificant whitespace and line breaks from the regex. Converting a regex from exact spacing to free-spacing escapes all literal whitespace and line breaks in the regex, so they will be seen as significant in the converted regex. RegxBuddy will not automatically add whitespace to beautify your regex. You can do that yourself after clicking Accept Conversion. If the target application supports only either mode, then RegxBuddy automatically converts your regex to that mode.

If the target application allows comments to be added to the regex, then you can choose whether to keep comments in the converted regex or strip comments from the converted regex. Some applications only support comments in free-spacing mode, so that choice affects whether the option to keep comments

appears. If the target application does not support comments, RegexBuddy strips comments without warning. If your original regex doesn't have any comments, then this option has no effect.

Converting to The Same Application

You can select the same application in the application drop-down list at the top and on the Convert panel. One reason to do this is to change the spacing mode of a regex. If you've stored a nicely formatted and commented regex in your RegexBuddy library but your application only provides a tiny one-line input box for the regex, then you can make your regex usable by converting it to exact spacing and stripping comments. Or, if you've inherited some source code with obtuse regular expressions, you can make those more readable by first converting them to free-spacing mode, and then adding whitespace and comments as you like.

Another reason is to automatically deal with unsupported syntax. For example, `\x{20AC}` is a syntax error in C#. But RegexBuddy is smart enough to guess that you've been trying to use the Perl syntax to match the Unicode code point for the euro symbol. When converting this regex from C# to Ce, the Convert panel says "conversion succeeded" and gives you `\uFFFF` as a fully functional regex that does what you intended.

Manual Conversion

The Convert panel is a powerful tool. But sometimes you may want more control. To manually convert a regular expression, switch to the Create panel. Select both the original application and the target application as the two flavors to compare your regular expression between. The comparison indicates all differences between the two applications that apply to your regex. You can then edit your regex by hand to work around those differences, or decide that certain differences can be ignored because they don't apply to the strings or files you'll be using the regex on.

33. Testing Regular Expression Actions

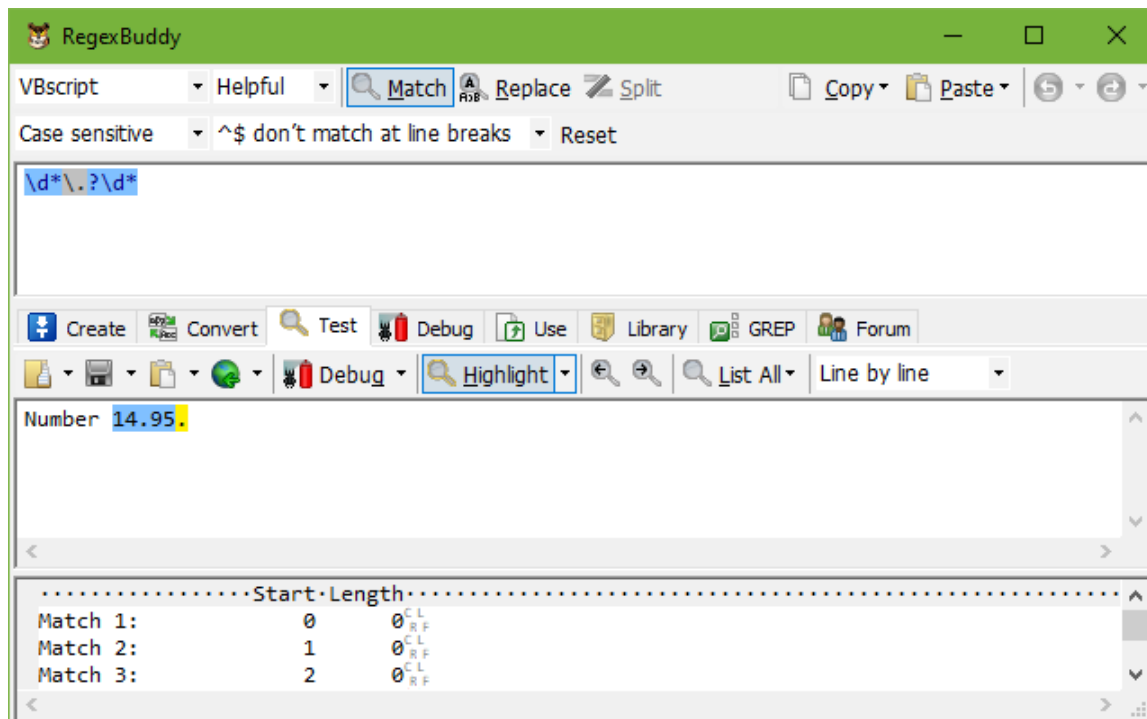
It is always a good idea to test your regular expression on sample data before using it on valuable, actual data, or before adding the regex to the source code for the application you are developing. You can debug a regular expression much easier with RegexBuddy.

It is obvious that you should check that the regex matches what it should match. However, it is far more important that you verify that it does *not* match anything that you do *not* want it to match. Testing for false positives is far more important, something which people new to regular expressions often do not realize.

An example will make this clear. Suppose we want to grab floating point numbers in the form of 123, 123.456 and .456. At first sight, the regex `\d*\.\?\d*` appears to do the trick.

To test this, type `Number 14.95.` in the edit box just below the toolbar on the Test panel. Then click the Find First button. RegexBuddy reports a zero-length match at offset 0. If you click Find Next, RegexBuddy reports a zero-length match at offset 1. Another click on Find Next gives a match at offset 2, and so on until `14.95.` is matched at offset 7. To top it off, the full stop at the end of the string is the final match.

While our regular expression successfully matches the floating point numbers we want, it also matches an empty string. Everything in the regex is optional. The solution is to make the regex enforce the requirement that a floating point number must contain at least one digit. See the floating point number example for a detailed discussion of this problem.



Preparing Sample Data to Test Your Regex

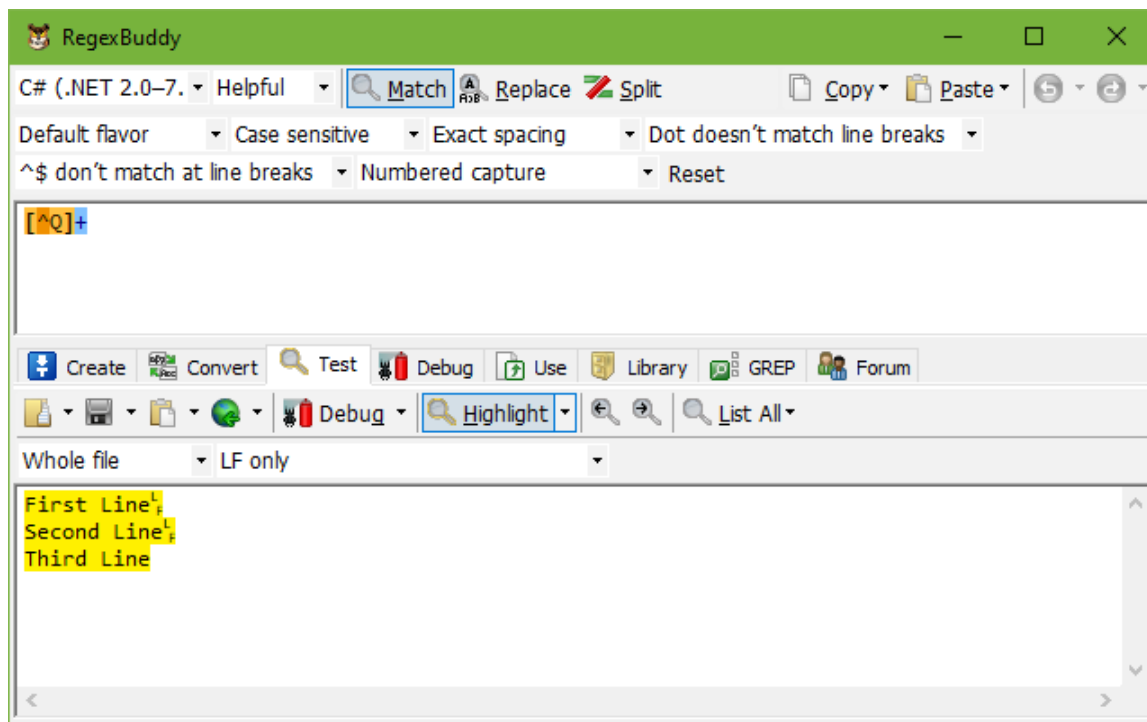
The Test panel has two edit boxes. The top one is where you enter the subject string to test the regex on. You can simply type in some text, or click the Open File button to use a text file as the subject string. You can access recently used test files by clicking the downward pointing arrow of the Open File button.

If you copied some text to the Windows clipboard, you can use that as the test subject by clicking the Paste Subject button. Paste “as is” to paste the text on the clipboard unchanged. If you copied a string literal from your source code, including the single or double quotes delimiting it, select one of the programming language string options when pasting. Select C-style for C++, Java, C#, etc., Pascal-style for Delphi, Basic-style for VB, etc.

If you get question marks instead of characters when you try to type or paste in some text, or if a file you’ve just opened displays with the wrong characters, RegexBuddy is using the wrong encoding for your text. To fix this, right-click on the text subject and select Encoding in the context menu. Pick the “reinterpret” option and select the correct encoding. This fixes incorrect characters in a file you’ve opened. Question marks that you typed or pasted in are not fixed automatically. Delete the question marks and type or paste the text again.

When working with binary data, right-click on the test subject and select Hexadecimal in the context menu to switch to hexadecimal mode.

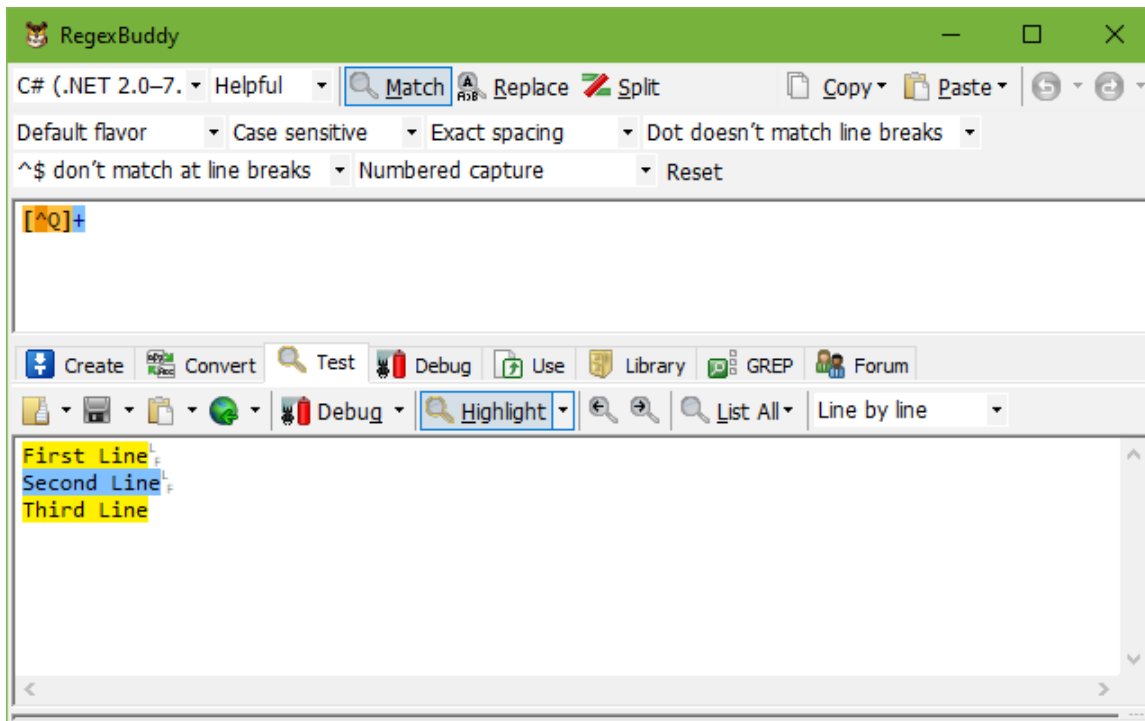
Test Scope



RegexBuddy provides only one big edit control for entering your test subject. By default, the “whole file” scope is selected in the toolbar. RegexBuddy then treats the test subject text as one long string, possibly consisting of multiple lines. The effect is the same as if you loaded the whole file into a single variable in a

programming language, and then passed that variable as the test subject to the function or class that does regex matching. Depending on your regular expression, regex matches may span across lines.

If you select the “line by line” scope, RegexBuddy treats each line in the edit control as a separate subject string. This gives you the same results as when you read a file line by line in a loop in a programming language, and do the regex match separately on each line inside the loop. Regex matches can never span across lines. The regex engine never sees more than one line at a time. RegexBuddy also won’t include the line break at the end of each line in the test subject. The “line by line” scope is a handy way of entering multiple test subjects if you know that your actual subject strings in your actual application never have line break characters in them.



The “page by page” scope doesn’t have an everyday programming analog. RegexBuddy provides this option to make it straightforward to specify multiple test subjects that each can consist of multiple lines. Press Ctrl+Enter in the edit control to insert a page break. The page break appears as a horizontal line in RegexBuddy. If you save the file, RegexBuddy saves a form feed ASCII character (`\f` or `\x0C`) at each page break position. If you were to enter page breaks with the “whole file” scope selected, RegexBuddy’s regex engine would also see the form feed characters when you test your regular expression. With the “page by page” scope selected, RegexBuddy treats the page breaks as hard delimiters that the regex engine can’t see. The regex engine is fed each block of text, page by page.

Changing the test scope does not change how your regular expression works. It can change the resulting matches, however, because it does change the test subject your regex is applied to. If your test subject is three lines of text that don’t include the character Q, for example, then the regex `[^Q]+` finds one match that spans the entire scope. With the scope set to “whole file”, there is only one test subject with three lines. RegexBuddy finds one match spanning all three lines. With “line by line” selected, there are three totally separate test subjects, each consisting of a single line. RegexBuddy then finds three matches, one for each test subject, each matching the whole line each test subject consists of. The above two screen shots show the effect.

Line Break Style

Windows text files normally use CRLF or `\r\n` line breaks. Linux and OS X use UNIX-style LF only or `\n` line breaks. Classic Mac used CR only or `\r` for line breaks. All editor controls in RegexBuddy handle all these line break styles, as well as all other Unicode line breaks even though they are not commonly used in plain text files.

Most regular expression engines are not so smart, however. Many recognize only `\n` as a line break character. This mainly affects how the dot and anchors behave.

In many cases, you'll never notice this. For example, the regex engines in Perl, Python, and Ruby only recognize `\n` as a line break. Yet, when you read a Windows text file in its entirety into a variable in any of these languages, a regex like `(?m) ^.*$` seems to work just fine. The reason is that these languages open files in "text mode" by default. In this mode, when your script runs on Windows, reading from a file automatically converts CRLF line breaks into LF only, and writing to a file automatically does the opposite conversion.

If you set the test scope to "line by line", then you don't need to worry about this. Then the regular expression never sees any line breaks and RegexBuddy won't show its line break handling option on the Test toolbar.

If you set the test scope to "whole file" or "page by page", then right after that, you can choose how RegexBuddy deals with line breaks. The default is "automatic line breaks", which converts line breaks to fit the regex engine of the selected application whenever you open a file or select another application. The parentheses after "automatic line breaks" indicate what the option does for the current application. For Perl, it will say (LF only) which means all CR and CRLF line breaks will be converted into LF only. For EditPad, it will say (no change) because EditPad's regex engine intelligently handles all line breaks, so no conversion is needed.

You can also select CRLF pair, LF only, or CR only to convert the active file to a particular line break style. Doing this disables the automatic line break conversion. The conversion is only applied once. If you open another file that uses a different line break style, the drop-down list will change to indicate that file's line break style. The file won't be converted unless you gain choose the line break style you want from the drop-down list. To re-enable automatic conversion, simply select that option again.

Some files that you open may use a mixture of different line break styles. If automatic conversion is off or requires no change, then RegexBuddy preserves the mixed line breaks. The line break style drop-down list will then have an additional option saying "mixed" along with the file's primary (most common) line break style. RegexBuddy uses the primary line break style for line breaks that you enter via the keyboard or paste from the clipboard. Selecting the "mixed" option in the drop-down list disables automatic line break conversion without converting the file to a particular line break style.

Line break conversion is permanent. With automatic line break conversion enabled and Perl as the active application, opening a Windows text file converts all the line breaks to LF only. If you then select EditPad as the application, the line breaks will remain as LF only, even though EditPad could handle all the file's original line breaks. When an application requires "no change" to the line breaks, RegexBuddy really does nothing to the line breaks. To revert to the file's original line break style, you'll need to reload it.

In most situations, "automatic line breaks" is the only option you need, as most regex engines handle line breaks in a way that is consistent with what the overall environment does. One major exception among the applications supported by RegexBuddy are the applications and programming languages based on the .NET

framework. Since .NET is a Windows development library, you'll most likely be reading text files with Windows line breaks. The .NET classes for reading files do not do any line break conversion. But the .NET Regex class treats only the line feed character `\n` as a line break. This means that if you have a string that contains `\r\n` line breaks, then the regex `(?m)^\.*$` includes `\r` at the end of each match. The regex `(?m)z$` will not find any matches. It cannot match `z` at the end of each line because the dollar will not match between `z` and `\r`.

Because the .NET Regex class only supports line feeds, and because RegexBuddy emulates only regex engines, not complete programming languages, the “automatic line breaks” option in RegexBuddy uses “LF only” for .NET. To match up the behavior between RegexBuddy and your .NET application, you'll either need to select “CRLF only” in RegexBuddy to disable the line break conversion, or you'll need to do the same conversion in your .NET application by replacing all matches of the literal string “`\r`” with nothing.

If your regex doesn't need to find matches that span multiple lines, then a less confusing solution may be to make your .NET application read files one line at a time, and pass each line separately to the regex. You can make RegexBuddy do the same by selecting “line by line” as the test scope. This way your subject strings will never contain any line breaks, so you don't need to worry how your regex deals with them.

Highlighting Matches

Click the Highlight button to highlight all regex matches in the test subject. The highlight feature makes it very easy to spot which text your regex matches. The highlighting is updated nearly instantly whenever you edit the regular expression or the test subject. If you're already quite experienced at creating regular expressions, typing in your regular expression with the Test panel visible and highlighting active is a very powerful, quick and hands-on approach to creating regular expressions.

The matches are highlighted with alternating colors. That makes it easier to differentiate between adjacent matches. If you put capturing groups into the regex, you can highlight the part of each match captured into a particular backreference. That way you can test whether you have the right number for the backreference, and whether it properly captures what you want to extract from each match. Click on the downward pointing arrow on the Highlight button to select the number of the backreference to highlight. The overall matches are still highlighted. The highlighting for the backreference alternates between two colors, along with the overall match. You can change the colors in the preferences.

When matches are highlighted, click the Previous button to jump to the match before the current position of the text cursor. Click the Next button to jump to the first match after the cursor. The Find First and Find Next buttons are not available when matches are being highlighted. To get more information about a particular match, double-click it. The bottommost text box shows the offset and the length of the overall match and all capturing groups. The number of matches is also indicated.

Note that some regular expressions, like the floating point example above, can result in zero-width matches. These matches won't be visualized by the highlighting, since they have no characters to highlight. If you double-click on a highlighted match, however, then the “match number of total” label counts those zero-width matches too. Double-clicking on the highlighted 14.95 in the screen shot at the top, for example, would say “match 8 of 10”, as there are 7 zero-width matches before it.

Match highlighting is done in a background thread. It is updated with each keystroke as you edit the test subject or regular expression. With long test subjects or complex regular expressions, the highlighting

temporarily disappears while RegexBuddy applies it in the background. You don't have to wait for it to appear in order to continue editing.

Find First and Find Next

When matches are not being highlighted, the Find First button applies the regular expression once to the entire subject string. The matched part of the string is selected in the edit box, and the text cursor is moved to the end of the match. Details about the overall match, and details about each capturing group, are shown in the text box at the bottom. If the regular expression cannot match the subject string at all, the bottommost text box will tell you so.

The Find Next button works just like the Find First button, except that it applies the regex only to the part of the subject string after (i.e. to the right and below) the text cursor's position in the subject edit box. If you do not move the text cursor yourself between clicking on the Find First and/or Find Next buttons, the effect is that Find Next continues after the previous match. This is just like a regular expression engine would do when you call its "find next" function without resetting the starting position, which is also what the Highlight and List All buttons do.

If you want to test the match from a particular starting position, just click at that position to move the text cursor there. Then click Find Next. This can lead to different match results than indicated by the Highlight and List All buttons.

List All

When defining a match action, the List All button will be visible. When you click this button, three options to list all matches will appear. When you select one of them, RegexBuddy will perform the search through the entire subject string. The list of matches is shown in the text box at the bottom. Double-click a highlighted match in the list of matches to select it in the subject text.

If you select "List all regex matches", then the matches are shown one after the other on separate lines. No other information is shown. If you select "List all matches in columns", the matches will also shown one after the other on separate lines. In addition, the text matched by all of the capturing groups will be listed in columns next to the overall matches. "List all matches with full details" list all matches with all capturing groups, if any, below each match. The capturing groups will be folded with the match. To see them, click the + button in the left margin to expand the match. Two columns, Start and Length, indicate the zero-based character offset of the start of the match (i.e. the number of characters before the match), and the length of the match (i.e. the number of characters in the match).

If your regular expression has capturing groups, there will be additional items "List all matches of group X" at the bottom of the List All menu. These items list matches one after the other on a line just like "List all regex matches", except that they'll show the text matched by a particular capturing group rather than the overall regex matches.

The option to show non-participating groups determines how RegexBuddy displays capturing groups that did not participate in the match attempt at all. E.g. when `a(b)?` matches `a`, the optional capturing does not participate at all, because there's no `b` to match. However, in the regex `a(b?)`, the group will always participate because it's not optional. It's contents are optional, so when the regex matches `a`, the group will participate without capturing anything. When showing non-participating groups, RegexBuddy will display

“n/a” in the results for the group in the former regex, and nothing for the group in the latter regex. If you turn off this option, RegexBuddy will show nothing for either group, as neither captured anything. In some programming languages, non-participating groups are set to NULL, while participating groups without a match are set to the empty string. For these languages, it can be important to have RegexBuddy make the distinction. However, if you plan to save or copy and paste the test results, the “n/a” strings may get in the way. Then you should turn this option off.

Replace All

When defining a replace action, the Replace All button will be visible. When you click this button, two menu items will appear. If you choose “Search-and-replace all matches”, RegexBuddy will perform the search-and-replace across the entire subject string. The result is shown in the text box at the bottom. Replacements will be highlighted in the result. Double-click a replacement to select the text that was replaced in the test subject.

If you select “List all replacements”, then RegexBuddy will search through the entire subject string. For each match, RegexBuddy substitutes backreferences in the replacement text. The replacements will be listed one after the other on separate lines. Essentially, what you get is the result of a search-and-replace with all the text that wasn’t matched by the regular expression removed from the results. Instead there will be a line break between each match.

Split

When defining a split action, the Split button is visible instead. Click it to make RegexBuddy split the subject string, and show the resulting list of strings in the edit box at the bottom. The strings in the list are separated by horizontal lines. If the subject string consisted of multiple lines, it is possible that some of the strings in the resulting list span more than one line.

Update Automatically

At the bottom of the List All and Replace All menus, there’s an option to make the results update automatically when you change the regex or test subject. Turning this on has the same effect as selecting the same List All, Replace All or Split command after each keystroke.

Turning on this option will allow you to see the effects of your changes as soon as RegexBuddy can calculate them. The List All, Replace All and Split commands run in a background thread. They won’t slow you down while you edit your regular expression or test subject.

Invoking The Debugger

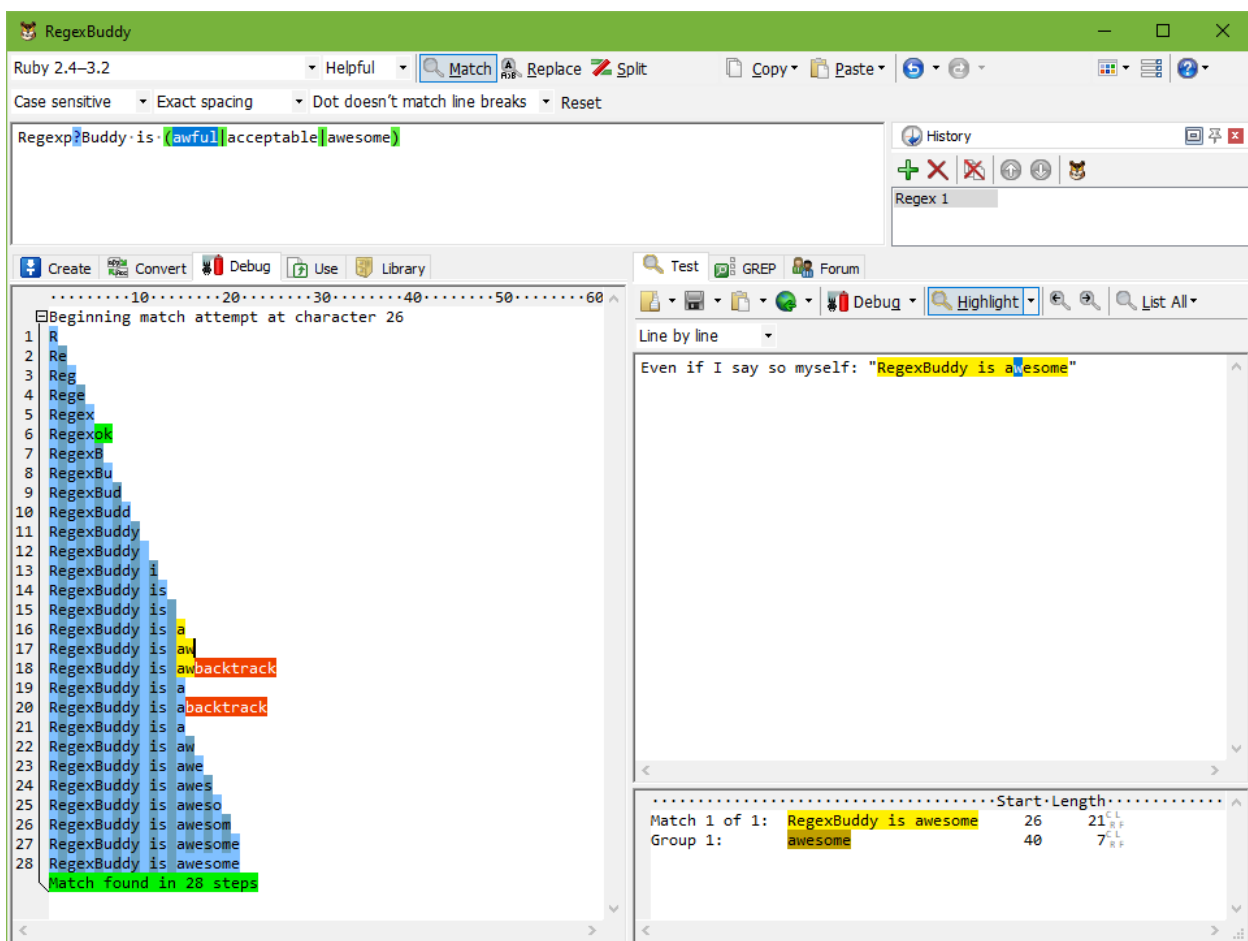
Click the Debug button to invoke the RegexBuddy debugger at the position of the text cursor in the test subject. RegexBuddy will automatically switch to the Debug panel to give you a fully detailed analysis of how your regular expression matches or doesn’t match your test subject.

34. Debugging Regular Expressions

RegexBuddy's regular expression debugger provides a unique view inside a regular expression engine. To invoke the debugger, switch to the Test panel. Place the text cursor in the test subject at the position you want to debug the regex match attempt. Then click the Debug button. If regex matches are highlighted in the test subject, placing the cursor in the middle of a highlighted match debugs the match attempt that yielded the match.

When you click the Debug button directly, RegexBuddy only debugs one match attempt. This is the attempt that starts at the position of the text cursor in the test subject at the time you clicked the Debug button. However, if the text cursor was in the middle of a highlighted match, the debugger starts at the beginning of the match rather than in the middle of the match.

If you want to debug all match attempts throughout the test subject, use the downward pointing arrow next to the Debug button and select "Debug everywhere". To make sure it doesn't run out of memory, the debugger stops after one million steps. If your regular expression is complex or your test subject very long, this may not be enough to step through it all. In that case, place the cursor at the position where the debugger gave up. Click the downward arrow next to the Debug button and invoke "Debug till end". This makes the debugger start at the position of the text cursor, just like "Debug here". However, at the end of the match attempt, the debugger advances through the test subject like "Debug everywhere".



What The Debugger Displays

When debugging more than one match attempt, RegxBuddy collapses all match attempts. To expand a match attempt, click on the + symbol in the left margin or press the plus key on the keyboard. To collapse it again, click the - symbol or press the minus key. Press Ctrl+Up and Ctrl+Down on the keyboard to jump from one match attempt to the next.

After you click the Debug button, the Debug panel displays each step in the match attempt. Each step is the result of one of two possible events. Either a token was successfully matched. The step indicates the text matched so far by the overall regular expression, including the match added by the last token. The engine then continues with the next token. In the screen shot, this is what happens at steps 1 through 5, 7 to 17, and 21 to 28.

The other event is a token that failed to match. The engine then backtracks to a previous position in the regex. The step indicates the text matched thus far, followed by “backtrack” highlighted in red which indicates the token that fails to match. So steps with “backtrack” indicate that the regex engine is going to backtrack. The next step shows where the engine ended up after backtracking.

In the screen shot, the regex engine backtracked twice. In step 18, the **f** in **awful** in the regex failed to match the **e** in **awesome** in the subject. In step 20, the first **c** in **acceptable** failed to match the **w** in **awesome**.

Backtracking means to go back in the regular expression to a position where the regex engine has remembered an alternative path. It does not necessarily mean going back in the text being matched. Such alternative paths exist wherever your regex uses alternation or quantifiers. In case of alternation, backtracking simply tries the next alternative. In case of greedy quantifiers, backtracking makes the quantifier give up one of its iterations. In case of lazy quantifiers, backtracking requires the quantifier to match another iteration. If the next alternative fails too, or if the quantifier can't give up or expand its match, the engine keeps backtracking until it finds an alternative path that leads to a match, or until there are no alternative paths left to try, failing the match attempt.

Sometimes, a token can match successfully without matching any text. These are called zero-length matches. Since the debugger doesn't have any text to display for such matches, it shows “ok” highlighted in green. In the screen shot, step 6 indicates that **p?** matched successfully. There's no **p** in the test subject, but since it's optional, we still have a successful match where the **p** is repeated zero times.

How to Inspect the Debugger's Output

To see which text was matched by a particular token at each step, click on the token in the regular expression. If the token consists of a single character, place the text cursor to the right of the token. Alternatively, you can click on the token in the regex tree on the Create panel. This both selects the token in the regular expression and highlights it in the debugger. You can arrange tabs side by side by dragging and dropping them with the mouse.

The text matched by the selected regex token is highlighted in yellow in the debugger's output. You can change the colors in the preferences. If the token is inside a group, its matches are only highlighted in the steps where the regex engine is processing the tokens inside the group. When the regex engine leaves the group, the text matched by the tokens inside the group is considered to have been matched by the group rather than the tokens inside it. If you place the text cursor after the closing round bracket of a group, the

situation is reversed. This way you can see the dynamics between the tokens in the group, and the group itself.

To find out which regex token matches a particular piece of text or character, simply click on that text in the debugging output. The token then becomes selected in the regular expression. This also causes all the text matched by that token to be highlighted in yellow. If you then switch to the Create panel, the token's regex block in the regex tree are also selected. The text you clicked on also becomes selected on the Test panel, so you can easily check its context.

In the screen shot above, I placed the cursor in the debug output at the end of step 17. RegexBuddy automatically highlighted “awful” in the regular expression. This is the regex token that matched “aw” in step 17. RegexBuddy also highlighted the “w” on the Test pane that corresponds with the “w” that’s highlighted in yellow on the Debug pane.

Differences Between The Debugger and a Real Regex Engine

The good news is that RegexBuddy’s regular expression debugger is an extension of a real regular expression engine. You get exactly the same (final) results on the Debug panel as you do on the Test panel. This also means that the debugger takes into account all the quirks of the regular expression flavor you’re working with.

However, to make it easier to follow the debugger’s output, the optimizations that the regex engine makes are disabled in the debugger. This is done to maintain a one-on-one relationship between the matching process and the regular expression you typed in. These optimizations do not change the final result, but often reduce the number of steps the engine needs to achieve the result.

For example, like most other regex engines, RegexBuddy’s regex engine optimizes the regular expression `(?:abcd|abef)` into `ab(?:cd|ef)`. The leading `ab` is then further optimized into a fast literal text search. This reduces the amount of needless backtracking quite a bit, while keeping the final result the same.

35. Comparing the Efficiency of Regular Expressions

The efficiency of a regular expression depends on a lot of factors. Each regular expression engine has different optimizations. The text you're applying the regular expression to also has a great impact. The more positions in the text where the regular expression can be partially matched, the more time the regex engine spends on those failed attempts. All this means that there's no straightforward way to benchmark regular expressions using the traditional (virtual) stopwatch outside of your actual application.

Still, RegexBuddy's debugger can give you a good view of the overall complexity of a regular expression. Essentially, the more steps the debugger needs to either find the match or declare failure, the more complex the regular expression. Particularly steps marked as "backtrack" are expensive.

When comparing the efficiency of regular expressions, you should run the debugger both on highlighted matches, as well as at positions where the regular expression cannot be matched. When comparing regular expressions to match a particular HTML tag, for example, place the text cursor on the Test panel before the < of an HTML tag that should not be matched. Click the Debug button to see how many steps it takes to figure out that tag shouldn't match.

Sometimes, the performance difference between two regular expressions can be quite severe. In the regular expressions tutorial, there's an example illustrating what is called catastrophic backtracking. The faulty regular expression needs 60,315 steps to fail a short string, a number that grows exponentially with the string's length. The improved regular expression needs only 36 linear steps to match, and another constant 12 to fail.

When comparing two regular expressions, first observe how the number of steps RegexBuddy needs grows. If one regex uses a constant number of steps while the other needs more steps for longer strings, the constant regex is by far the best, even if it needs more steps. However, don't grind your teeth trying to make it constant. It's often impossible for complex text patterns.

If neither regex is constant, compare their growth rate relative to the length of the string. You'll need to test at least three string lengths. First, check if the growth is linear. It is when the number of steps roughly doubles when you double the length of the string. If one regex grows linearly and the other grows exponentially, the linear regex is always better, even if it needs more steps. As the length of the string grows, the exponential regex will soon exhaust the capabilities of any regular expression engine.

Comparing the actual number of steps only makes sense if both regular expressions are either constant or linear. The one with fewer steps wins. If both are exponential, start with trying to make one of them linear. It's not always possible, but certainly worth trying.

Benchmark Both Success and Failure

You should always benchmark your regular expression both on test subjects that match, and test subjects that don't. In fact, the performance killer is usually slow failure rather than slow matching. Often, a regular expression is used to extract small bits from a larger file. In that case, the regular expression's performance at positions in the file where it cannot match is far more important than its performance at matching. If you want to extract 10 strings of 100 characters from a file that's a million characters long, the regular expression has to match 10 times, and fail 999,000 times.

Writing a regular expression that matches linearly is much easier than writing one that fails linearly. When a regex fails, the regular expression engine does not give up until it has tried all possible permutations of the all the alternations and quantifiers in your regular expression. You'll be surprised how numerous those are.

Making Regular Expressions More Efficient

There are two important techniques to make an exponential regex linear. The easiest and most important one is to make adjacent regex tokens mutually exclusive whenever possible. When writing a regex that locates delimited content, and the delimiters cannot appear (in escaped form) in the content, specify that in your regular expression. Doing so makes sure that the regex engine does not attempt to include the delimiter as part of the content, which significantly reduces the number of pointless permutations the regex engine tries.

As a test, compare the regular expressions "[^"]*" and ".*?" on the test subject "test". Both regexes match, but the former needs 3 steps in the debugger, while the latter needs 11. Now test "this is a test". The former still needs 3 steps, but the latter needs 31. Clearly, a greedy negated character class is more efficient than a lazy dot. In actual search time, the impact of this isn't measurable when your input files only contain short strings. It can be significant when you're searching thousands of files with strings thousands of characters long.

The second technique is the use of atomic grouping and/or possessive quantifiers. Both these features are a fairly recent addition to the regular expression culture. RegexBuddy supports both, but your programming language might not. These regex tokens come in handy when you can't use negated character classes. Basically, an atomic group locks in the part of the text the regex matched so far. It says: when you've reached this point, don't bother going back to try more permutations. If you can't find a match, fail right away.

In many situations, you cannot replace a lazy dot with a negated character class to prevent the delimiter from being included in the content. Then you can use an atomic group to achieve the same result. Simply place the atomic group around the lazy dot (or whatever repeated regex token that shouldn't match the following delimiter) and the delimiter that follows it. Then, as soon as the delimiter is matched, the atomic group is locked down. If the remainder of the regex fails, the lazy dot won't get the chance to expand itself and gobble up the delimiter.

See The Difference in RegexBuddy

The tutorial topic on catastrophic backtracking includes a detailed example. You can easily see the effect for yourself in RegexBuddy. The two regular expressions are included in RegexBuddy's library as "HTML file" and "HTML file (atomic)" along with identical test data. Click the Use button on the Library panel and pick "Use regex and test data".

Both regexes highlight the whole test subject. (If not, you cheated and used copy and paste rather than the "Use regex and test data" command, which doesn't turn on the "dot matches line breaks" option these regexes need.) If you click on the highlighted match and then click the Debug button, RegexBuddy finds the match in 401 steps for the first regex, and 407 steps for the atomic regex.

The screenshot shows the RegExBuddy interface with the following details:

- Regex:** `<html>.*<head>.*<title>.*</title>.*</head>.*<body.*>.*</body>.*</html>`
- Test Subject:** A large block of HTML code, including `<html>...<head>...<title>This is a test</title>...</head>...<body bgcolor=white>...<h1>This is a test</h1>...<p>First paragraph</p>...<p>a href="http://www.regxbuddy.com/">Link paragraph</p>...</body>...</html>`
- Match:** The entire HTML document is highlighted in yellow, indicating a successful match.
- Match 1 of 1:** Shows the full HTML document structure.
- Match Found:** A green box at the bottom left indicates "Match found in 401 steps".

The screenshot shows the RegExBuddy interface with the following details:

- Regex:** `<html>(?!.*<head>)(?!.*<title>)(?!.*</title>)(?!.*</head>)(?!.*<body.*>)(?!.*</body>).*</html>`
- Test Subject:** The same HTML code as in the previous screenshot, but with the final `</html>` tag removed.
- Match:** No match is found, as the document is not properly closed.
- Match 1 of 1:** Shows the HTML code without the closing tag.
- Match Found:** A green box at the bottom left indicates "Match found in 407 steps", which represents the failure.

Now, delete the very last `>` character from the test subject. The match highlighting disappears immediately, as the regex no longer matches the test subject. Move the text cursor immediately to the left of the very first `<` character in the file. Then click the Debug button. The first regex needs 2,555 steps to conclude failure. The regex using atomic grouping needs only 426 steps.

If you look closely at the debugger output, you'll see that the first regex produces some kind of vertical sawtooth, extending all the way, backing up a little, extending all the way, backing up some more, etc. The second regex, however, gradually matches the whole file, and then drops everything in just a few steps.

In the screen shots below, you can clearly see that in the one but last step of the first regex, the first `.*?` in the regex has “eaten up” the whole file. The second regex only has one “backtrack” for each `.*?`.

RegExBuddy (Visual Basic .NET 2.0-7.0) interface showing a match attempt that fails after 2555 steps. The regex is `<html>.*?<head>.*?<title>.*?</title>.*?</head>.*?<body>.*?</body>.*?</html>`. The debugger output shows a vertical sawtooth pattern of "backtrack" messages as the first `.*?` in the regex consumes the entire file and then backtracks.

```

Step 0 .....190 .....200 .....210 .....220 .....230 .....240 .....250 .....260 .....270 .....280 .....290 .....300 .....310 .....320 .....330 .....340 .....350 .....360 .....370 .....380 .....390 .....400 .....410 .....420 .....430 .....440 .....450 .....460 .....470 .....480 .....490 .....500 .....510 .....520 .....530 .....540 .....550 .....560 .....570 .....580 .....590 .....600 .....610 .....620 .....630 .....640 .....650 .....660 .....670 .....680 .....690 .....700 .....710 .....720 .....730 .....740 .....750 .....760 .....770 .....780 .....790 .....800 .....810 .....820 .....830 .....840 .....850 .....860 .....870 .....880 .....890 .....900 .....910 .....920 .....930 .....940 .....950 .....960 .....970 .....980 .....990 .....1000 .....1010 .....1020 .....1030 .....1040 .....1050 .....1060 .....1070 .....1080 .....1090 .....1100 .....1110 .....1120 .....1130 .....1140 .....1150 .....1160 .....1170 .....1180 .....1190 .....1200 .....1210 .....1220 .....1230 .....1240 .....1250 .....1260 .....1270 .....1280 .....1290 .....1300 .....1310 .....1320 .....1330 .....1340 .....1350 .....1360 .....1370 .....1380 .....1390 .....1400 .....1410 .....1420 .....1430 .....1440 .....1450 .....1460 .....1470 .....1480 .....1490 .....1500 .....1510 .....1520 .....1530 .....1540 .....1550 .....1560 .....1570 .....1580 .....1590 .....1600 .....1610 .....1620 .....1630 .....1640 .....1650 .....1660 .....1670 .....1680 .....1690 .....1700 .....1710 .....1720 .....1730 .....1740 .....1750 .....1760 .....1770 .....1780 .....1790 .....1800 .....1810 .....1820 .....1830 .....1840 .....1850 .....1860 .....1870 .....1880 .....1890 .....1900 .....1910 .....1920 .....1930 .....1940 .....1950 .....1960 .....1970 .....1980 .....1990 .....2000 .....2010 .....2020 .....2030 .....2040 .....2050 .....2060 .....2070 .....2080 .....2090 .....2100 .....2110 .....2120 .....2130 .....2140 .....2150 .....2160 .....2170 .....2180 .....2190 .....2200 .....2210 .....2220 .....2230 .....2240 .....2250 .....2260 .....2270 .....2280 .....2290 .....2300 .....2310 .....2320 .....2330 .....2340 .....2350 .....2360 .....2370 .....2380 .....2390 .....2400 .....2410 .....2420 .....2430 .....2440 .....2450 .....2460 .....2470 .....2480 .....2490 .....2500 .....2510 .....2520 .....2530 .....2540 .....2550 .....2555
Match attempt failed after 2555 steps
  
```

RegExBuddy (Visual Basic .NET 2.0-7.0) interface showing a match attempt that fails after 426 steps. The regex is `<html>(.*?)<head>(.*?)<title>(.*?)</title>(.*?)</head>(.*?)<body>(.*?)</body>(.*?)</html>`. The debugger output shows a single "backtrack" message as the second `.*?` in the regex consumes the whole file and then backtracks.

```

.....10 .....20 .....30 .....40 .....50 .....60 .....70 .....80 .....90 .....100 .....110 .....120 .....130 .....140 .....150 .....160 .....170 .....180 .....190 .....200 .....210 .....220 .....230 .....240 .....250 .....260 .....270 .....280 .....290 .....300 .....310 .....320 .....330 .....340 .....350 .....360 .....370 .....380 .....390 .....400 .....410 .....420 .....430 .....440 .....450 .....460 .....470 .....480 .....490 .....500 .....510 .....520 .....530 .....540 .....550 .....560 .....570 .....580 .....590 .....600 .....610 .....620 .....630 .....640 .....650 .....660 .....670 .....680 .....690 .....700 .....710 .....720 .....730 .....740 .....750 .....760 .....770 .....780 .....790 .....800 .....810 .....820 .....830 .....840 .....850 .....860 .....870 .....880 .....890 .....900 .....910 .....920 .....930 .....940 .....950 .....960 .....970 .....980 .....990 .....1000 .....1010 .....1020 .....1030 .....1040 .....1050 .....1060 .....1070 .....1080 .....1090 .....1100 .....1110 .....1120 .....1130 .....1140 .....1150 .....1160 .....1170 .....1180 .....1190 .....1200 .....1210 .....1220 .....1230 .....1240 .....1250 .....1260 .....1270 .....1280 .....1290 .....1300 .....1310 .....1320 .....1330 .....1340 .....1350 .....1360 .....1370 .....1380 .....1390 .....1400 .....1410 .....1420 .....1430 .....1440 .....1450 .....1460 .....1470 .....1480 .....1490 .....1500 .....1510 .....1520 .....1530 .....1540 .....1550 .....1560 .....1570 .....1580 .....1590 .....1600 .....1610 .....1620 .....1630 .....1640 .....1650 .....1660 .....1670 .....1680 .....1690 .....1700 .....1710 .....1720 .....1730 .....1740 .....1750 .....1760 .....1770 .....1780 .....1790 .....1800 .....1810 .....1820 .....1830 .....1840 .....1850 .....1860 .....1870 .....1880 .....1890 .....1900 .....1910 .....1920 .....1930 .....1940 .....1950 .....1960 .....1970 .....1980 .....1990 .....2000 .....2010 .....2020 .....2030 .....2040 .....2050 .....2060 .....2070 .....2080 .....2090 .....2100 .....2110 .....2120 .....2130 .....2140 .....2150 .....2160 .....2170 .....2180 .....2190 .....2200 .....2210 .....2220 .....2230 .....2240 .....2250 .....2260 .....2270 .....2280 .....2290 .....2300 .....2310 .....2320 .....2330 .....2340 .....2350 .....2360 .....2370 .....2380 .....2390 .....2400 .....2410 .....2420 .....2430 .....2440 .....2450 .....2460 .....2470 .....2480 .....2490 .....2500 .....2510 .....2520 .....2530 .....2540 .....2550 .....2560 .....2570 .....2580 .....2590 .....2600 .....2610 .....2620 .....2630 .....2640 .....2650 .....2660 .....2670 .....2680 .....2690 .....2700 .....2710 .....2720 .....2730 .....2740 .....2750 .....2760 .....2770 .....2780 .....2790 .....2800 .....2810 .....2820 .....2830 .....2840 .....2850 .....2860 .....2870 .....2880 .....2890 .....2900 .....2910 .....2920 .....2930 .....2940 .....2950 .....2960 .....2970 .....2980 .....2990 .....3000 .....3010 .....3020 .....3030 .....3040 .....3050 .....3060 .....3070 .....3080 .....3090 .....3100 .....3110 .....3120 .....3130 .....3140 .....3150 .....3160 .....3170 .....3180 .....3190 .....3200 .....3210 .....3220 .....3230 .....3240 .....3250 .....3260 .....3270 .....3280 .....3290 .....3300 .....3310 .....3320 .....3330 .....3340 .....3350 .....3360 .....3370 .....3380 .....3390 .....3400 .....3410 .....3420 .....3430 .....3440 .....3450 .....3460 .....3470 .....3480 .....3490 .....3500 .....3510 .....3520 .....3530 .....3540 .....3550 .....3560 .....3570 .....3580 .....3590 .....3600 .....3610 .....3620 .....3630 .....3640 .....3650 .....3660 .....3670 .....3680 .....3690 .....3700 .....3710 .....3720 .....3730 .....3740 .....3750 .....3760 .....3770 .....3780 .....3790 .....3800 .....3810 .....3820 .....3830 .....3840 .....3850 .....3860 .....3870 .....3880 .....3890 .....3900 .....3910 .....3920 .....3930 .....3940 .....3950 .....3960 .....3970 .....3980 .....3990 .....4000 .....4010 .....4020 .....4030 .....4040 .....4050 .....4060 .....4070 .....4080 .....4090 .....4100 .....4110 .....4120 .....4130 .....4140 .....4150 .....4160 .....4170 .....4180 .....4190 .....4200 .....4210 .....4220 .....4230 .....4240 .....4250 .....4260
Match attempt failed after 426 steps
  
```

As a final test, type `1234567890` at the end of the test subject, so it ends with `</html1234567890`. Put the text cursor back before the very first `<` character in the file, and click Debug. The regex without atomic grouping now needs 2,695 steps, while the atomic regex needs only 446. The 10 additional characters add 140 steps for the first regex, but only 20 for the second. That's 14-fold the number of characters we added for the first regex, but only 2-fold for the second.

RegxBuddy Visual Basic (.NET 2.0-7.0) Match Replace Split Copy Paste

Default flavor Case insensitive Exact spacing Dot matches line breaks ^\$ don't match at line breaks Numbered capture Reset

<html>.*<head>.*<title>.*</title>.*</head>.*<body>.*</body>.*</html>

History HTML file (lazy) HTML file (atomic)

Step 0.....200.....210.....220.....230.....240.....2

2666 boring stuff omitted-></body>.*backtrack

2667 boring stuff omitted-></body>.*h

2668 boring stuff omitted-></body>.*hbacktrack

2669 boring stuff omitted-></body>.*ht

2670 boring stuff omitted-></body>.*hbacktrack

2671 boring stuff omitted-></body>.*htm

2672 boring stuff omitted-></body>.*htmbacktrack

2673 boring stuff omitted-></body>.*html

2674 boring stuff omitted-></body>.*htmlbacktrack

2675 boring stuff omitted-></body>.*html1

2676 boring stuff omitted-></body>.*html1backtrack

2677 boring stuff omitted-></body>.*html12

2678 boring stuff omitted-></body>.*html12backtrack

2679 boring stuff omitted-></body>.*html123

2680 boring stuff omitted-></body>.*html123backtrack

2681 boring stuff omitted-></body>.*html1234

2682 boring stuff omitted-></body>.*html1234backtrack

2683 boring stuff omitted-></body>.*html12345

2684 boring stuff omitted-></body>.*html12345backtrack

2685 boring stuff omitted-></body>.*html123456

2686 boring stuff omitted-></body>.*html123456backtrack

2687 boring stuff omitted-></body>.*html1234567

2688 boring stuff omitted-></body>.*html1234567backtrack

2689 boring stuff omitted-></body>.*html12345678

2690 boring stuff omitted-></body>.*html12345678backtrack

2691 boring stuff omitted-></body>.*html123456789

2692 boring stuff omitted-></body>.*html123456789backtrack

2693 boring stuff omitted-></body>.*html1234567890

2694 boring stuff omitted-></body>.*html1234567890backtrack

2695 Match attempt failed after 2695 steps

Whole file LF only

```
<html>
<head>
<title>This is a test</title>
</head>
<body bgcolor=white>
<h1>This is a test</h1>
<p>First paragraph</p>
<p><a href="http://www.regxbuddy.com/">Link</a> paragraph</p>
<!--More boring stuff omitted-->
</body>
</html1234567890
```

The regular expression does not match the test subject

RegxBuddy Visual Basic (.NET 2.0-7.0) Match Replace Split Copy Paste

Default flavor Case insensitive Exact spacing Dot matches line breaks ^\$ don't match at line breaks Numbered capture Reset

<html>(<?>.*<head>)(?>.*<title>)(?>.*</title>)(?>.*</head>)(?>.*<body>.*)(?>.*</body>).*</html>

History HTML file (lazy) HTML file (atomic)

Step 0.....10.....20.....30.....40.....50.....6

417 <html><head><title>This is a test</title></head><body bg

418 <html><head><title>This is a test</title></head><body bg

419 <html><head><title>This is a test</title></head><body bg

420 <html><head><title>This is a test</title></head><body bg

421 <html><head><title>This is a test</title></head><body bg

422 <html><head><title>This is a test</title></head><body bg

423 <html><head><title>This is a test</title></head><body bg

424 <html><head><title>This is a test</title></head><body bg

425 <html><head><title>This is a test</title></head><body bg

426 <html><head><title>This is a test</title></head><body bg

427 <html><head><title>This is a test</title></head><body bg

428 <html><head><title>This is a test</title></head><body bg

429 <html><head><title>This is a test</title></head><body bg

430 <html><head><title>This is a test</title></head><body bg

431 <html><head><title>This is a test</title></head><body bg

432 <html><head><title>This is a test</title></head><body bg

433 <html><head><title>This is a test</title></head><body bg

434 <html><head><title>This is a test</title></head><body bg

435 <html><head><title>This is a test</title></head><body bg

436 <html><head><title>This is a test</title></head><body bg

437 <html><head><title>This is a test</title></head><body bg

438 <html><head><title>This is a test</title></head><body bg

439 <html><head><title>This is a test</title></head><body bg

440 <html><head><title>This is a test</title></head><body bg

441 <html><head><title>This is a test</title></head><body bg

442 <html><head><title>This is a test</title></head>backtrack

443 <html><head><title>This is a test</title>backtrack

444 <html><head><title>backtrack

445 <html><head>backtrack

446 <html>backtrack

Match attempt failed after 446 steps

Whole file LF only

```
<html>
<head>
<title>This is a test</title>
</head>
<body bgcolor=white>
<h1>This is a test</h1>
<p>First paragraph</p>
<p><a href="http://www.regxbuddy.com/">Link</a> paragraph</p>
<!--More boring stuff omitted-->
</body>
</html1234567890
```

The regular expression does not match the test subject

By using atomic grouping, we achieved a 7-fold reduction in the regular expression's complexity. Both regular expressions are in fact linear. But two steps per character is a huge savings from fourteen steps per character. If you're scanning through a megabyte worth of invalid HTML files, the optimized regex has 2 MB of work to do, while the original regex has 14 MB of work to do.

If you wonder where these numbers come from, the regex has seven lazy dots. The lazy dot matches a character (one step), proceed with the next token, and then backtrack when the next token fails (second step). When using atomic grouping, each lazy dot matches each character in the file only once. None of the dots can match beyond their delimiting HTML tag. This yields two steps per character. In the original regex, the dots can backtrack to include their delimiters and match everything up to the end of the file. Since there are seven lazy dots, all characters at the end of the file are matched by all seven, taking fourteen matching steps.

Had you only compared the performance of these regexes on a valid HTML file, you might have been fooled into thinking the original is a fraction faster. In reality, their performance at matching is the same, since neither regex does any more backtracking than needed. It's only upon failure that backtracking gets out of hand with the first regex.

36. Generate Source Code to Use a Regular Expression

Once you have created and tested a regex action, you are ready to use it. One way is to simply tell RegexBuddy to copy the regular expression to the clipboard, so you can paste it into tool or editor where you want to use the regex. If you do this often, you may want to see if it is possible to integrate RegexBuddy and the software you use regular expressions with.

If you want to use the regular expression in the source code for an application you are developing, switch to the Use panel in RegexBuddy. RegexBuddy can generate code snippets in a variety of programming languages that have built-in support for regular expressions, or that have a popular regular expression library available for them.

Benefits of RegexBuddy's Code Snippets

Using RegexBuddy's code snippets gives you several major benefits. You don't have to remember all the details on how to use a particular language's or library's regex support. Though most languages and libraries have similar capabilities, the actual implementation is quite different. With RegexBuddy, you simply select the task you want to accomplish from the list, and RegexBuddy generates source code you can readily copy and paste into your code editor or IDE.

Another key benefit is that when using a regular expression in a programming language, certain characters need special treatment, up and above the special treatment they may need in the regular expression itself. The regex `\\` to match a single backslash, for example, is automatically inserted as `"\\\\\\\\"` in a Java or C code snippet, and as `/\\\\/` in a JavaScript or Perl snippet. Never again mess around with pesky backslashes!

For languages that support exception handling, RegexBuddy's code snippets also contain `try..catch` or equivalent code blocks to handle any exception that could be thrown by any of the methods the code snippet calls.

How to Generate a Code Snippet

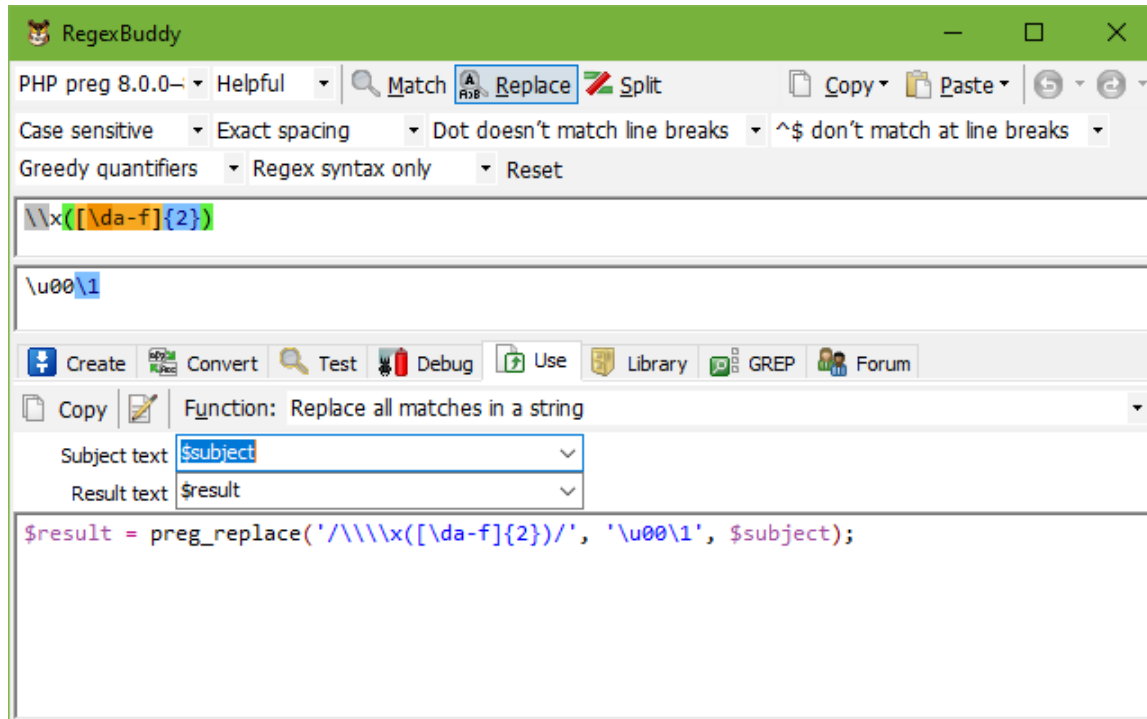
To generate a code snippet, first select your programming language from the list of applications in the top left corner. Then select the type of function you want to implement. Some functions are only available for certain languages. The functions that create an object for repeatedly applying a regular expression, for example, are only available in languages with object-oriented regex libraries. No functions are available for applications that aren't programming languages.

The available functions also depend on the kind of action you defined. Search and extraction functions are listed for match actions, search-and-replace functions are listed for replace actions, and splitting functions are listed for split actions.

Most functions allow you to specify certain parameters, such as the variable or string constant you want to use as the subject for the regex operation, the variable to store the results in, the name of the regex or matcher object, etc. RegexBuddy does not verify whether you enter a valid constant or variable. Whatever you enter is inserted into the code snippet unchanged. The parameters you enter are automatically remembered. For each parameter, the same values are carried over across all the functions. The parameters are remembered separately for each language. So you need to enter the names you typically use only once.

RegexBuddy keeps the code snippet in sync with your regular expression at all times, even when the regex is syntactically incorrect. Use the Create and Test panels to make sure your regex works correctly with your programming language. The snippet on the Use panel is ready for copying and pasting at all times. If you want, you can edit the snippet in RegexBuddy. Just remember that your changes are lost as soon as you edit the regular expression.

You can transfer the snippet into your code editor or IDE by clicking the Copy button on the Use panel's toolbar. This copies the entire snippet to the Windows clipboard, ready for pasting. If you only want to use part of the snippet, either select it and press Ctrl+C on the keyboard to copy it, or select it and then drag-and-drop it with the mouse into your code editor.



Available Languages

RegexBuddy ships with source code templates for all of the world's most popular programming languages. Each template implements a comprehensive set of functions, or things that you can do with your regular expression in the selected programming language.

If the provided functions don't match your coding style, you can easily edit any of the templates by clicking the Edit button on the toolbar on the Use panel. You can edit or delete the provided functions, and add as many new functions as you like.

You can also create new templates. These can be alternative sets of functions for a programming language already supported by RegexBuddy, such as a set of functions specific to a particular development project. You can also create templates for languages not supported by RegexBuddy. The only limitation is that you'll be limited to languages that use one of the regex flavors and string styles already supported by RegexBuddy. To use a new template, first save the new template in the template editor. Then add a custom application and select the new template when adding the application.

37. Available Functions in Source Code Snippets

To make it easy for you to apply the same regex techniques in a variety of programming languages, RegexBuddy uses the same function descriptions for all languages it supports. Some functions are not available for all languages, depending on the functionality provided by the language or its regular expression library.

Import regex library: If the regular expression support is not a core part of the language, you need to reference the regex library in your code, before you can use any of the other code snippets.

Test if the regex matches (part of) a string: Tests if the regular expression can be matched anywhere in a subject string, and stores the result in a boolean variable.

Test if the regex matches a string entirely: Tests if the regular expression matches the subject string entirely, and stores the result in a boolean variable. This function is appropriate when validating user input or external data, since you typically want to validate everything you received. If the language does not have a separate function for testing if a string can be matched entirely, RegexBuddy places anchors around your regular expression to obtain the same effect.

If/else branch whether the regex matches (part of) a string: Tests if the regular expression can be matched anywhere in a subject string. Add code to the *if* section to take action if the match is successful. Add code to the *else* section to take action if the match fails.

If/else branch whether the regex matches a string entirely: Tests if the regex matches a subject string in its entirety, adding anchors to the regex if necessary.

Create an object with details about how the regex matches (part of) a string: Applies the regular expression to a subject string, and returns an object with detailed information about the part of the subject string that could be matched. The details usually include the text that was matched, the offset position in the string where the match started, and the number of characters in the match. If the regex contains capturing groups, the details also contain the same information for each of the capturing groups. The groups are numbered starting at one, with group number zero being the overall regex match. The Find First button on the Test panel performs the same task as this function.

Note that for regular expressions that consist only of anchors or lookaround, or where everything is optional, it is possible for a successful match to have a length of zero. Such regexes can also match at the point right after the end of the string, in which case the offset points to a non-existent character.

Get the part of a string matched by the regex: Applies the regular expression to a subject string, and returns the part of the subject string that could be matched. This function is appropriate if you are only interested in the first (or only) possible match in the subject string.

Get the part of a string matched by a numbered group: Applies the regular expression to a subject string, and returns the part of the subject string that was matched by a capturing group in the regex. This function is appropriate if you are only interested in a particular part of the first (or only) possible match in the subject string. One of the parameters for this function is the number of the capturing group. You can easily obtain this number from the regex tree on the Create panel.

Get the part of a string matched by a named group: Applies the regular expression to a subject string, and returns the part of the subject string that was matched by a named capturing group in the regex. This

function is appropriate if you are only interested in a particular part of the first (or only) possible match in the subject string. One of the parameters for this function is the name of the capturing group.

Iterate over all matches in a string: Applies and re-applies a regular expression to a subject string, iterating over all matches in the subject, until no further matches can be found. Insert your own code inside the loop to process each match. Match details such as offset and length of the match, the matched text, the results from capturing groups, etc. are available inside the loop.

Iterate over all matches and capturing groups in a string: Iterates over all matches in the string like the previous function, and also iterates over all capturing groups of each match. Note that the number of available capturing groups may be different for each match. If certain capturing groups did not participate at all in the overall match, no details will be available for those groups. In practice, you will usually use the previous function, and only work with the capturing groups that you are specifically interested in. This function illustrates how you can access each group.

Validate input with a regular expression: Applies the regular expression to the text entered in a field on a form to validate the field before processing the form.

Replace Functions

These functions are available for languages that can use a regular expression to search-and-replace through a string. These functions only appear in the Function drop-down list when you're defining a replace action.

Replace all matches in a string: Executes a replace action. All regex matches in the subject string are substituted with the replacement text. References to capturing groups in the replacement text are expanded. Unless otherwise indicated in the function's name, the "replace all" function does not modify the original subject string. It returns a new string with the replacements applied.

Search and replace through a string: Executes a replace action using a callback function that returns the replacement text. You can edit this callback function to use procedural code to modify or build the replacement for each regex match.

Split Functions

These functions are available for languages that can use a regular expression to split a string. These functions only appear in the Function drop-down list when you're defining a split action.

Split a string: Executes a split action. Returns an array or list of strings from a given subject string. The first string in the array is the part of the subject before the first regex match. The second string is the part between the first and second matches, the third string the part between the second and third matches, etc. The final string is the remainder of the subject after the last regex match. The text actually matched by the regex is not added to the array.

Object-Oriented Functions

These functions are available for programming languages that use an object-oriented regular expression library.

Create an object to use the same regex for many operations: Before a regular expression can be applied to a subject string, the regex engine needs to convert the textual regex that you (or the user of your software) entered into a binary format that can be processed by the engine's pattern matcher. If you use any of the functions listed above, the regex is (re-)compiled each time you call one of the functions. That is inefficient if you use the same regex over and over. Therefore, you should create an object that stores the regular expression and its associated internal data for the regex engine. Your source code becomes easier to read and maintain if you create regex objects with descriptive names.

Create an object to apply a regex repeatedly to a given string: Some regex libraries, such as Java's `java.util.regex` package, use a separate pattern matcher object to apply a regex to a particular subject string, instead of using the regex object created by the function above. Explicitly creating and using this object, instead of using one of the convenience functions that create and discard it behind the scenes, makes repeatedly applying the same regex to the same string more efficient. This way you can find all regex matches in the string, rather than just the first one.

Apply the same regex to more than one string: Illustrates how to use the regex object to apply a given regular expression to more than one subject string.

Use regex object to ...: This series of functions creates and uses a regex object to perform its task. The results of these functions are identical to the results of the similarly named functions already described above. The advantage of these functions over the ones above, is that you can reuse the same regex object to use a given regular expression more than once. For the second and following invocations, you obviously only copy and paste the part of the code snippet that uses the regex object into your source code.

Database Functions

These functions are available for database languages (various flavors of SQL).

Select rows in which a column is/cannot (partially) matched by the regex: SQL code for selecting rows from a database checking for a regex match on a certain column. Only those rows where the column does/doesn't match the regex are returned.

Select rows in which a column is/cannot entirely matched by the regex: SQL code for selecting rows from a database checking for a regex match on a certain column. Anchors are added to the regex to require it to match the column's value entirely. Only those rows where the column does/doesn't match the regex are returned.

Select the part of a column matched by a regex: SQL code for selecting rows from a database checking for a regex match on a certain column. Returns only the part of the column that was matched by the regular expression, and only for those rows where the column matched at all.

Regex Tree Functions

Comment with RegexBuddy's regex tree: A series of comment lines with the regular expression “as is” (not converted to a string or operator) and the regex tree from the Create panel in RegexBuddy.

Comment inside a region with RegexBuddy's regex tree: A series of comment lines with the regular expression “as is” (not converted to a string or operator) and the regex tree from the Create panel in RegexBuddy. The comment lines are placed inside a region to make it easy to fold the whole comment in your development tool.

XML comment with RegexBuddy's regex tree: A series of comment lines with the regular expression “as is” (not converted to a string or operator) and the regex tree from the Create panel in RegexBuddy. The comment has added XML tags that allow it to be included in documentation that is automatically generated from your source code.

String literal with RegexBuddy's regex tree: For regex flavors that support the free-spacing regular expression syntax, RegexBuddy converts your regex to use free-spacing mode (if it doesn't already) and add the regex tree as comments to the regular expression. For regex flavors that do not support free-spacing, RegexBuddy splits the regex into a concatenation of strings, adding the regex tree as source code comments next to the string parts. Either way, the resulting string holds the actual regular expression, and can be passed to a class or function to actually match the regular expression to a subject string.

Replacement Tree Functions

Comment with RegexBuddy's replacement tree: A series of comment lines with the replacement text “as is” (not converted to a string or operator) and the replacement tree from the Create panel in RegexBuddy.

Comment inside a region with RegexBuddy's replacement tree: A series of comment lines with the replacement text “as is” (not converted to a string or operator) and the replacement tree from the Create panel in RegexBuddy. The comment lines are placed inside a region to make it easy to fold the whole comment in your development tool.

XML comment with RegexBuddy's replacement tree: A series of comment lines with the replacement text “as is” (not converted to a string or operator) and the replacement tree from the Create panel in RegexBuddy. The comment has added XML tags that allow it to be included in documentation that is automatically generated from your source code.

String literal with RegexBuddy's replacement tree: RegexBuddy splits the replacement into a concatenation of strings, adding the replacement tree as source code comments next to the string parts. The resulting string holds the actual replacement text, and can be passed to a class or function to actually replace regex matches.

38. Copying and Pasting Regular Expressions

Copy and paste is the simple and easy way to transfer a regular expression between RegexBuddy and your searching, editing and coding tools. You can use the regular Ctrl+C and Ctrl+V shortcut keys to copy and paste the selected part of the regex as is. Or, you can use the Copy and Paste buttons on RegexBuddy's toolbar to transfer the regular expression in different formats. If you often use RegexBuddy in conjunction with a particular tool or application, you may want to see if it is possible to integrate RegexBuddy with that tool.

The search boxes of text editors, grep utilities, etc. do not require the regular expression to be formatted in any way, beyond being a valid regex pattern. This is exactly the way you enter the regular expression in RegexBuddy, whether you type it in or insert tokens on the Create panel. So you can simply copy and paste the regex "as is".

If you want to use the regular expression in the source code of an application or script you are developing, the regex needs to be formatted according to the rules of your programming language. Some languages, such as Perl and JavaScript use a special syntax reserved for regular expressions. Other languages rely on external libraries for regular expression support, requiring you to pass regexes to their function calls as strings. This is where things get a bit messy.

Flavors and Matching Modes

Matching modes like "dot matches newline" and "case insensitive" are generally not included as part of the regular expression. That means they aren't included when copying and pasting regular expressions. The only exceptions are the JavaScript, Perl, PHP preg, and Ruby operators. For all the other string styles, you'll need to make sure by yourself that you're using the same matching modes in your actual tool or source code as in RegexBuddy. If you want RegexBuddy to take care of this for you, use the source code snippets on the Use panel rather than direct copy and paste.

In particular, free-spacing mode is something you need to pay attention to. Many string styles, including Basic, C, Java, JavaScript, and Pascal, do not support multi-line strings. When you tell RegexBuddy to copy a regular expression to the clipboard using one of these string styles, RegexBuddy will copy a concatenation of multiple strings to the clipboard, one for each line in your regular expression. Each string will end with a line break, so comments in the regex are terminated correctly. That way your regex will be formatted the same way in your source code as in RegexBuddy.

When pasting the regular expression back into RegexBuddy, RegexBuddy cannot determine from the string alone whether the line breaks in the string are line breaks that your regular expression should match, or whether the line breaks are free-spacing line breaks. To solve this, RegexBuddy simply looks whether you've selected free-spacing or exact spacing in the drop-down list on the main toolbar. If you selected "exact spacing", line breaks in the string are included as `\r\n` in the regex. If you selected "free-spacing", line breaks are treated as whitespace that split the regex into multiple lines in RegexBuddy. To make sure RegexBuddy pastes your regular expression correctly, set the free-spacing mode in RegexBuddy to match the free-spacing option in your source code *before* you paste the regex.

Copying a regular expression doesn't convert it into the correct regular expression flavor. For example, if you're editing a regular expression with the PCRE flavor selected and you select the Copy as JavaScript Operator command, you'll get a PCRE-style regular expression formatted as a JavaScript operator. If you want the regular expression to be fully converted to JavaScript, first convert the regular expression on the

Convert panel. After clicking Accept Conversion, you can then select Copy as JavaScript Operator to copy the converted regex as a JavaScript operator.

Copying The Regex as a String or Operator

In regular expressions, metacharacters must be escaped with a backslash. In many programming languages, backslashes appearing in strings must be escaped with another backslash. This means that the regex `\\` which matches a single backslash, becomes `\\\\` in Java or C/C++. The regex `"\\"` which matches a single backslash between double quotes, becomes `"\\\\"`. How's that for clarity?

When you generate code snippets on the Use panel, RegexBuddy automatically inserts your regexes correctly formatted as a strings or operators as required by the programming language. To use a regex you prepared in RegexBuddy without creating a code snippet, click the Copy button on the main toolbar. You can copy the regular expression to the clipboard in one of several formats. Directly under the Copy button, you can find the string style used by the programming language you've selected in the list of applications. If the source code template for that application uses a different string style, that one will also be listed directly under the Copy button. All other string styles will be listed under the "Copy Regex As" submenu.

As Is: Copies the regex unchanged. Appropriate for tools and applications, but not for source code.

Basic-style string: The style used by programming languages derived from Basic, including Visual Basic and Xojo (REALbasic). A double-quoted string. A double quote in the regex becomes two double quotes in the string.

C string: A string of char in C and C++. A double-quoted string. Backslashes and double quotes are escaped with a backslash. Supports escapes such as `\t`, `\n`, `\r`, and `\xFF` at the string level.

C Wide string: A string of wchar_t in C and C++. A double-quoted string prefixed with the letter L. Backslashes and double quotes are escaped with a backslash. Supports escapes such as `\t`, `\n`, `\r`, `\xFF`, and `\uFFFF` at the string level.

C++11 Raw string: A string of char in C++11 quoted as a Raw string. Raw strings can contain literal line breaks and unescaped quotes and backslashes. Does not support any character escapes. If the regex contains the characters `)"` then RegexBuddy automatically uses a longer custom delimiter to make sure the delimiter does not occur in the regex. If the regex does not contain any line breaks, quotes, or backslashes a normal double-quoted string is copied as then there is no benefit to using a raw string.

C++11 Wide Raw string: A string of wchar_t in C++11 quoted as a Raw string.

C# string: If the regex contains backslashes, it will be copied as a verbatim string for C# which doesn't require backslashes to be escaped. Otherwise, it will be copied as a simple double-quoted string.

Delphi string: The style used by Delphi and other programming languages derived from Pascal. A single-quoted string. A single quote in the regex becomes two single quotes in the string.

Delphi Prism string: The style used by Delphi Prism, formerly known as Oxygene or Chrome. Either a single-quoted string on a single line, or a double-quoted string that can span multiple lines. A quote in the regex becomes two quotes in the string.

Groovy string: The Groovy programming language offers 5 string styles. Single-quoted and double-quoted strings require backslashes and quotes to be escaped. Using three single or three double quotes allows the string to span multiple lines. For literal regular expressions, the string can be delimited with two forward slashes, requiring only forward slashes to be escaped.

Java string: The style used by the Java programming language. A double-quoted string. Backslashes and double quotes are escaped with a backslash. Unicode escapes `\uFFFF` allowed.

JavaScript string: The string style defined in the ECMA-262 standard and used by its implementations like JavaScript, JScript and ActionScript. A single-quoted or double-quoted string. Backslashes and quotes are escaped with a backslash. Unicode escapes `\uFFFF` and Latin-1 escapes `\xFF` allowed.

JavaScript operator: A Perl-style `//` operator that creates a literal `RegExp` object in the ECMAScript programming language defined in the ECMA-262 standard, and its implementations like JavaScript, JScript and ActionScript. ECMA-262 uses mode modifiers that differ from Perl's.

Perl-style string: The style used by Perl, where a double-quoted string is interpolated, but a single-quoted string is not. Quotes used to delimit the string, and backslashes, are escaped with a backslash.

Perl operator: A Perl `m//` operator for match and split actions, and an `s///` operator for replace actions.

PHP string: A string for use with PHP's `ereg` functions. Backslashes are only escaped when strictly necessary.

PHP `'/'` preg string: A Perl-style `//` operator in a string for use with PHP's `preg` functions.

PostgreSQL string: A string for PostgreSQL, delimited by double dollar characters.

PowerShell string: A string for PowerShell. Uses "here strings" for multi-line strings. Quotes and non-printables are escaped with backticks.

Python string: Unless the regex contains both single and double quote characters, the regex is copied as a Python "raw string". Raw strings do not require backslashes to be escaped, making regular expressions easier to read. If the regex contains both single and double quotes, the regex is copied as a regular Python string, with quotes and backslashes escaped.

R string: The string style used by the R programming language. A single-quoted or double-quoted string. Backslashes and quotes are escaped with a backslash. Unicode escapes `\U0010FFFF`, basic Unicode escapes `\uFFFF`, and Latin-1 escapes `\xFF` allowed.

Ruby operator: A Perl-style `//` operator for use with Ruby. Ruby uses mode modifiers that differ from Perl's.

Scala string: Copies the regex as a triple-quoted Scala string which avoids having to escape backslashes and allows line breaks which is handy for free-spacing regular expressions.

SQL string: The string style used by the SQL standard. A single-quoted string. A single quote in the regex becomes two single quotes in the string. The string can span multiple lines. Note that not all databases use this string style. E.g. MySQL uses C-style strings, and PostgreSQL uses either C-style strings or dollar-delimited strings.

Tcl word: Delimits the regular expression with curly braces for Tcl. In Tcl parlance, this is called a word.

XML: Replaces ampersands, angle brackets and quotes with XML entities like & suitable for pasting into XML files.

Pasting a String or Operator as The Regular Expression

RegexBuddy can do the opposite conversion when you want to edit a regular expression that is already part of an application's source code. In your source code editor or IDE, select the entire string or operator that holds the regular expression. Make sure quotes and other delimiters are included. Copy it to the clipboard.

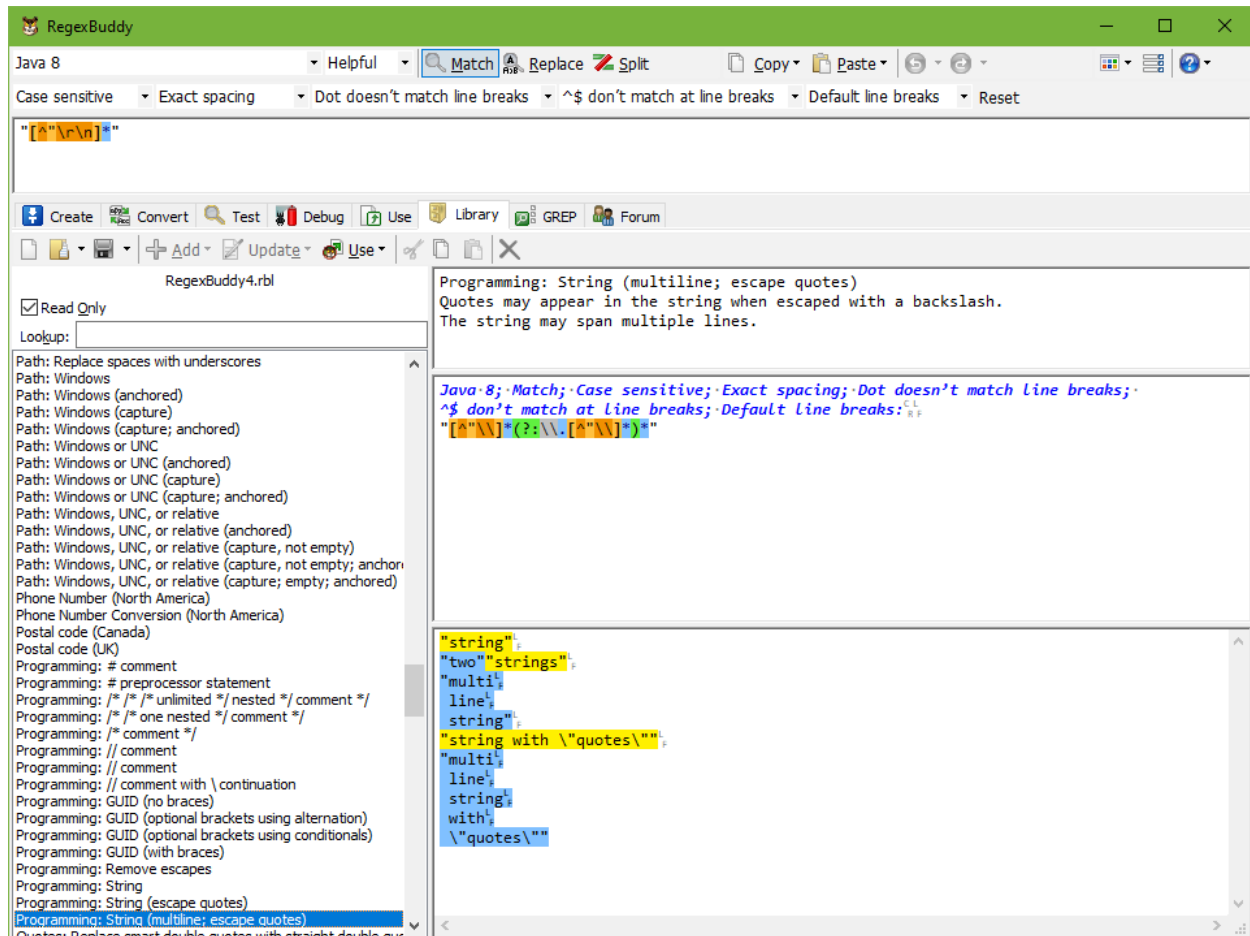
Then switch to RegexBuddy, and click the Paste button in the main toolbar. The Paste button's menu offers the same options as the Copy menu, except that they work the other way around. If the clipboard holds the Java string `"\\\""`, select "Regex from Java string" and RegexBuddy will properly interpret the string as the regular expression `"\""`, which matches a single backslash between a pair of double quote characters. When you're done editing the regex, select "Copy as Java string" and you can paste the updated regex as a Java string into your source code.

There is only one item for pasting C strings. It automatically recognizes the L, R, and LR prefixes. You can use this one item to paste C strings, wide C strings, raw C++11 strings, and wide raw C++11 strings.

Just like when copying regular expressions, you need to take care that you've set the same matching modes in RegexBuddy as in the application or source code you've copied the regex from. Pasting a regular expression also doesn't change the programming language or application selected in RegexBuddy, nor does it convert the regular expression to the selected application. Make sure to select the correct programming language or application before or after pasting the regular expression.

39. Storing Regular Expressions in Libraries

When you have gone through the effort of creating and testing a regular expression, it would be a waste to use it just once. It makes a lot of sense to store the regexes you create for later reuse, by yourself or by friends and colleagues. RegxBuddy makes storing and sharing regex actions easy.



To store a regular expression, switch to the Library panel in RegxBuddy. To create a new library, first click the New button on the Library toolbar. Then click the Save button to store the library into a new RegxBuddy Library file.

If you want to store it into an existing library, click the Open button to select the RegxBuddy Library file. Or click the small downward pointing arrow on the Open button to reopen a library that you recently worked with.

Any changes you make to a RegxBuddy Library are saved automatically. RegxBuddy makes sure that you never lose any regexes you worked so hard on. If you open the same library in more than one instance of RegxBuddy, RegxBuddy makes sure all the instances are kept in sync, by automatically saving and reloading the library. In other words: you don't have to worry at all about making sure your work is saved. As a precaution against inadvertent modifications, all libraries are opened as "read only" by default. You can toggle the read only state with the check box just below the New and Open buttons on the Library panel's toolbar.

One exception is the sample RegexBuddy library that is included with RegexBuddy. When you open it, RegexBuddy turns on the “read only” state permanently. The reason is that when you upgrade RegexBuddy, the sample library is upgraded too. If you want to edit the sample library, click the Save As button to make a copy.

Since RegexBuddy automatically and regularly saves libraries, there is no button or command for you to do so manually. If you want to edit a library, and preserve the original for safekeeping, click the Save As button before turning off the read only option. Save As tells RegexBuddy to create a copy of the library under a new name. Any subsequent changes are automatically saved into the new file. The original file stays the way it is.

Adding a Regex Action to The Library

First, you need to define the regex action in the topmost part of RegexBuddy’s window. Then, click on the Library panel’s Add button. If the button is disabled, you forgot to turn off the “read only” checkbox on the Library panel, or you did not type in a regular expression yet. You can either add the regular expression by itself, or you can add it along with the text you’ve typed or loaded into the Test panel.

The entire regex action then appears in the lower right part of RegexBuddy’s window. If your regular expression contains parameters like %PARAM%, a grid appears where you can edit the parameters. If you stored a test subject with the regular expression, its text also appears, with regex matches highlighted.

Enter a clear description in the edit box just below the Library panel’s toolbar. You can enter as much text as you want. A good description greatly helps to look up the regex later and to differentiate it from other similar regexes.

The first line of the description is used as the label for the regex in the list at the left hand side of the Library panel. If a regex does not have a description, the list shows the first line of the regex itself.

Updating Regex Actions in The Library

You cannot directly edit regular expressions and test data stored in the library. To change a regular expression, click the Update button and select to update the regular expression. This replaces the library item’s regular expression with the one you’re currently editing in RegexBuddy. Click the Update button and select to update the test subject to replace the library item’s test subject with the text currently on the Test panel. Both updates leave the description unchanged.

Copying and Pasting Regexes Between Libraries

Each instance of RegexBuddy can open only a single library. If you want to move or duplicate regex actions between two libraries, start a second instance of RegexBuddy. You can easily do that by clicking the RegexBuddy button in the History. Then use the Cut, Copy and Paste buttons to move or copy a regex action from one instance of RegexBuddy to the other.

If you open the same library in more than once RegexBuddy instance, both instances are automatically kept in sync. If you make changes in one instance, they automatically appear in the other instance when you switch to it.

40. Parameterizing a Regular Expression Stored in a Library

When storing a regular expression in a RegexBuddy Library for later reuse, it is sometimes obvious that many variants of the regex should also be in the library. E.g. the regex `^.{7}(.{3})` captures the 8th through 10th characters of a line into backreference 1. Though you may want to store this regex for later reuse, it is quite likely that in the future, you will use the same technique to capture a different range of characters.

Rather than storing a large number of similar regexes into a library, you can parameterize a regular expression. Replace all parts of the regex that should be variable, with a parameter. A parameter is a sequence of one or more letters A..Z and digits 0..9, delimited by a pair of percentage signs. E.g. turn `^.{7}(.{3})` into `^{%SKIPAMOUNT%}(.{%CAPTUREAMOUNT%})`. Then add the regex to the library. It doesn't matter if adding the parameters makes the regular expression invalid on the Create panel. The library will accept it and detect the parameters.

When you've selected a regex with parameters in the library, a new grid appears on the Library panel. This grid has three columns. It has one row for each parameter you inserted into the regex. In the second column, briefly describe the parameter. For lengthy descriptions, use the edit box just below the Library panel's toolbar instead. There, you can enter as much text as you want.

Provide a default value for each parameter in the third column. Substituting each parameter with its default value should result in a valid regular expression. That makes it easier to reuse the regex later.

When substituting a regular expression, RegexBuddy does not interpret a parameter's value in any way. The parameter's tag is simply replaced with the value, whatever it is. Metacharacters in the value are not automatically escaped. Though this means you have to be a bit careful when entering a value for a parameter, it also enables you to substitute a parameter with a complete regular expression. The above example could be generalized into `^{%SKIPAMOUNT%}(%REGEX%)`. Rather than capturing a fixed number of characters, this regex template matches whichever regular expression you want, starting at a specific column on a line.

When adding a replace action to a library, you can use parameters in the replacement text just like you can use them in the regular expression. If you specify the same parameter more than once, in the regex and/or the replacement text, all occurrences of that parameter are replaced with the same value.

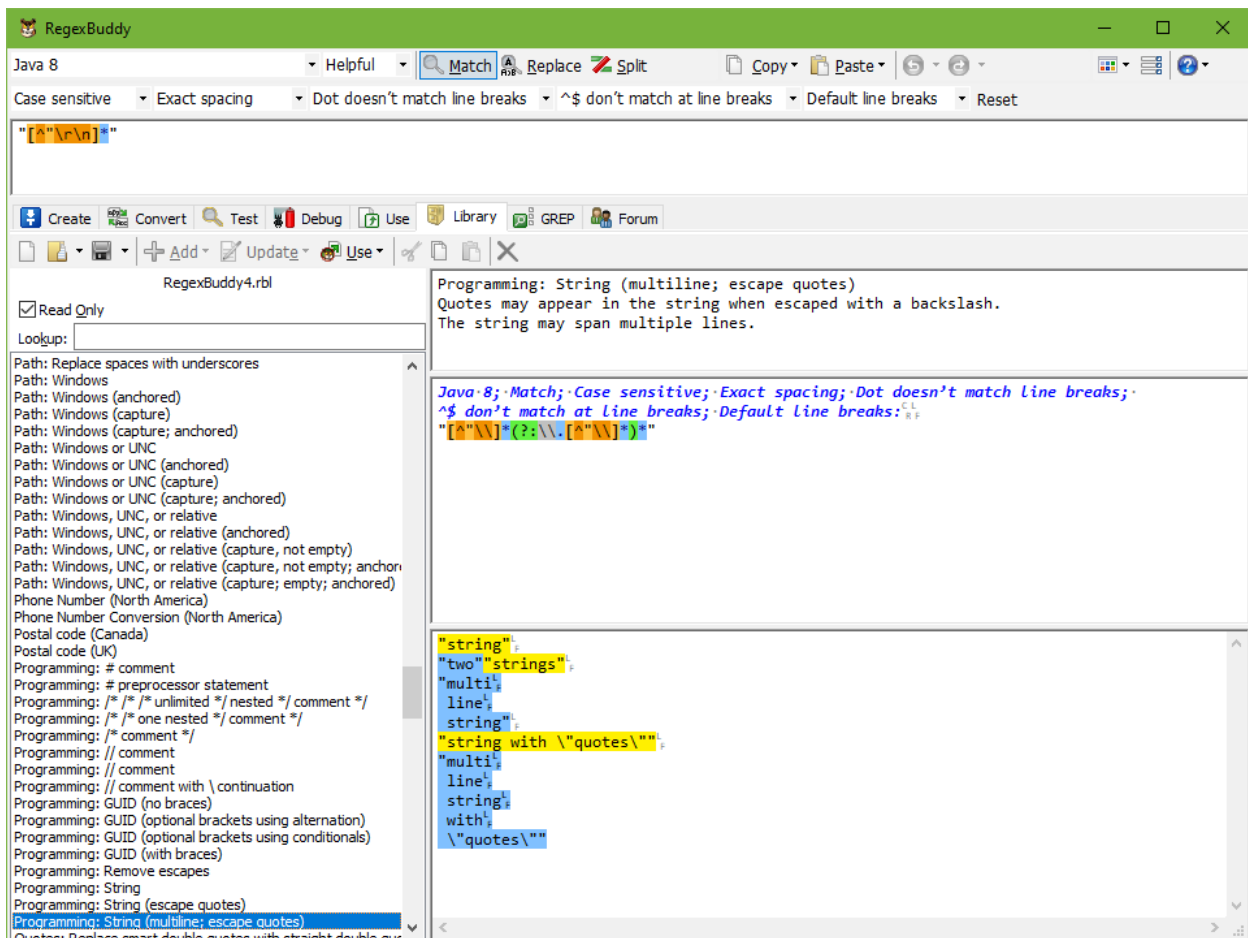
41. Using a Regular Expression from a Library

RegexBuddy makes it very easy to create regular expressions, which is nice. But it is even nicer to use ready-made regular expressions provided by others, or reuse regexes that you saved yourself in the past.

On the Library panel, click the the Open button to open the RegexBuddy Library from which you want to reuse a regular expression. Or click the small downward pointing arrow on the Open button to reopen a library that you recently worked with. If this is the first time you are using RegexBuddy's Library panel, the list of libraries that you recently worked with shows the standard RegexBuddy library that is included with RegexBuddy itself. This library contains a wide range of sample regex actions that you can use for various purposes.

To use a regex action from the library, select it from the list at the left, and click the Use button and select to use the regular expression. This copies the regex action to the regular expression area at the top, where you define regex actions in RegexBuddy. Then you can analyze, test and implement the regular expression as usual.

Some library items also have test data associated with them. If you click the Use button and select to use the test data, then the test data is copied to the Test panel in RegexBuddy.



Correct Application and Mode Settings

Each regular expression in a library is intended for a particular application with a particular set of matching modes. All this is stored with the regex in the library. When you use a regex from a library, those settings are copied over to the application and option drop-down lists on the top toolbars.

If a regex from a library targets a different application than the one you want to use the regex with, first click the Use button on the library toolbar to load the regex and the settings it was saved with. Then, select your target application on the Convert panel. If the conversion goes well, click the Accept Conversion button to finish converting the library regex to your target application. You can now copy the regex into your application or generate a code snippet.

To make sure the regex works as intended, make sure that you set the same matching modes in your actual application. Code snippets generated on the Use panel do this automatically. But if you just copy and paste the regex, you'll need to set those options yourself. You can see them in the drop-down lists on the top toolbar. You can also see them as a comment above the regex on the Library panel.

Substituting Parameters

When storing a regular expression in a RegexBuddy Library, it can be parametrized. When you select a parametrized regular expression and click the Use button, RegexBuddy asks you what you want to substitute the parameters with. Each parameter has a description that explains what you should enter as the value. A default value is also provided. If you can't read the complete description because the column is too narrow, make the parameter window wider. The description column stretches along with the window.

RegexBuddy does not interpret a parameter's value in any way. The parameter's tag is simply replaced with the value, whatever it is. Metacharacters in the value are not automatically escaped. Though this means you have to be a bit careful when entering a value for a parameter, it also enables you to substitute a parameter with a complete regular expression. You can see a preview of the final regular expression, and the replacement text in case of a replace action, as you type in values for the parameters.

42. GREP: Search and Replace through Files and Folders

If you've defined a match action, the GREP panel enables you to search through any number of files and folders using the regular expression you created. If you defined a replace action, you can use it to search-and-replace through files and folders. This way you can test your regular expression on a larger number of files (rather than just one file on the Test panel), and even perform actual search or file maintenance tasks.

Type the folder you want to search through in the Folders field, or click the ellipsis button next to it to select the folder from a folder tree. If you want to search through multiple folders, simply enter them all delimited by semicolons or commas. To search through a folder that has a semicolon or comma in its name, enclose the entire folder with double quotes. Turn on "recurse subfolders" to make RegexBuddy search through all subfolders of the folder(s) you specified.

If you don't want to search through all files in the folder, you can restrict the search using file masks. In a file mask, the asterisk (*) represents any number (including none) of any character, similar to `.*` in a regular expression. The question mark (?) represents one single character, similar to `?` in a regular expression. E.g. the file mask `*.txt` tells RegexBuddy to search through any file with a .txt extension. You can delimit multiple masks with semicolons or commas. To search through all C source and header files, use `*.c;*.h`. To include a literal semicolon or comma in a file mask, enclose that file mask with double quotes.

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`.

Sometimes it is easier to specify what you do not want to search through. To do so, mark the "invert mask" option. With this option, a file mask of `*.c;*.h` tells RegexBuddy to search through all files except C source and header files.

If you turn on "include binary files", then RegexBuddy searches through all files that can be found with your folder and file mask settings. If you turn it off, then RegexBuddy checks all those files to see whether they are text files or binary files and only searches through the text files. A file is seen as binary if it contains NULL bytes within the first 64K of its contents that don't fit the pattern of a UTF-16 or UTF-32 file.

By default, RegexBuddy searches through a whole file at once, and allows regex matches to span multiple lines. Traditional grep tools, however, search through files line by line. You can make RegexBuddy do the same by turning on the "line-based" option. When grepping line by line, you can turn on "invert results" to make RegexBuddy list all lines in each file which the regular expression does *not* match.

Target and Backup Files

When searching, RegexBuddy does not modify any files by default. The search matches are simply displayed on screen. To save all the search matches into a single file, select the "save results into a single file" option. Click the ellipsis button to the far right of the target setting to select the file you want to save the results into. To create one file for each file searched through, select "save one file for each searched file". Click the ellipsis button to select the folder to save the files into. Each saved file will have the same name as the file searched through.

When executing a search-and-replace, RegexBuddy can either modify the files it searched through, or create copies of those files and modify the copies. If you select to “copy only modified files” or “copy all searched files”, click the ellipsis button to select the folder to save the target files into. Again, each target file will have the same name as the original.

RegexBuddy does not prompt you when the GREP action overwrites one or more files. Therefore, you should make sure the backup option is set to anything except “no backups”. The “single .bak” and “single .~??” options create a backup file for each target file that is overwritten. If the backup file already existed, the previous backup is lost. The “multi .bak” and “multi backup N of” options create numbered backup files without overwriting existing backup files. The “same file name” option keeps one backup for each target file, and gives the backup copy the same name. The “hidden history” option creates a hidden folder named “__history” in each folder where files are overwritten which keeps backups with numbered extensions.

The “same file name” backup option requires you to specify a folder into which backup files should be stored. The other backup options except “hidden history” allow you to choose. If you specify a backup folder, all backups will be stored in that folder. If not, backups are saved in the same folders as the target files that were overwritten. Keeping backups in a separate folder makes it easy to get rid of the backups later, but makes it harder to find a backup file if you want to restore a file manually.

The screenshot shows the RegexBuddy application window. The title bar reads "RegexBuddy". The menu bar includes "Python 3.11-3.12", "Helpful", "Match", "Replace", "Split", "Copy", "Paste", and "Reset". The toolbar contains "Case sensitive", "Exact spacing", "Dot doesn't match line breaks", "^[^\$ don't match at line breaks", "Regex syntax only", and "Reset". The main window displays the following search results:

```

TOTAL: 19623 matches in 134 files
118 matches in S:\JGsoft\RegexBuddy4\Clients\Delphi\RegxBuddy_TLB.pas
133 matches in S:\JGsoft\RegexBuddy4\Clients\Delphi\RegxBuddyClientUnit.pas
15 matches in S:\JGsoft\RegexBuddy4\Clients\Delphi\RegxBuddyROT.pas
5 { Copyright (c) 2004-2013 Jan Goyvaerts }
8 { Design & implementation, by Jan Goyvaerts, 2007 }
9 { Updated for RegexBuddy 4, by Jan Goyvaerts, 2013 }
83 if ShellExecute(0, nil, PChar(RegxBuddyEXE), PChar('/rot ' + IntToStr(ID)), nil, SW_SHOW) <= 32 then Exit;
84 // Try to connect to RegexBuddy for up to 5 seconds
85 Count := 100;
87 OleCheck(GetRunningObjectTable(0, ROT));
88 StrPtr := CoTaskMemAlloc(260);
90 Sleep(50);
94 while EnumMoniker.Next(1, Moniker, nil) = S_OK do begin
95 OleCheck(CreateBindCtx(0, BindCtx));
114 until Result or (Count <= 0);
184 matches in S:\JGsoft\RegexBuddy4\Cross\RXB.Actions.pas
894 matches in S:\JGsoft\RegexBuddy4\Cross\RXB.Convert.pas
5 { Copyright (c) 2004-2023 Jan Goyvaerts }
8 { Design & implementation, by Jan Goyvaerts, 2007-2011 }
9 { Version 4 design & impl., by Jan Goyvaerts, 2011-2023 }
66 const LateNumber = 0;
67 const NoNumber = -1;
68 const NoNumberLookahead = -2;
69 const NoNumberLookbehind = -3;
70 const NoNumberLookaroundRemoved = -4;
71 const NoNumberEmptyError = -5;
72 const NoNumberExplicitCapture = -6;

```

Preview, Execute and Quick Execute

RegexBuddy can GREG in three modes: preview, execute and quick execute. You can access these modes through the GREG button. A preview displays detailed results in RegexBuddy, without modifying any files. Execute an action to create or modify target files as you specified, and to see detailed results afterwards to verify the results.

Use “quick execute” when you know the GREG action will do what you want, to quickly update files without displaying detailed results in RegexBuddy. The “quick execute” option can be much faster than “execute” when performing a search-and-replace through large numbers of files, since it doesn’t require RegexBuddy to keep track of each individual search match. On a typical computer, “preview” and “execute” can handle about 100,000 search matches, while “quick execute” has no practical limits.

Backup Files and Undo

If you followed my advice to have RegexBuddy create backup files, you can instantly undo the effects of a GREG action. Click on the GREG button, and then select Undo from the menu. This replaces all files that were overwritten with their backup copies. The backups are removed in the process. Only the very last action can be undone by RegexBuddy, and only if you do not close RegexBuddy. If you have multiple instances of RegexBuddy open, each instance can undo its own very last GREG action.

If you’ve verified the results, and all target files are in order, click the GREG button and select Delete Backup Files to delete the backup files created by the very last GREG action.

Opening and Saving Actions and Results

The Clear button clears the GREG action and results. It is not necessary to click the Clear button before starting a new action, but it may make things easier for you by reducing clutter.

Click the Open button to open a GREG action previously saved with the Save button. Note that only the action (regular expression with folder, mask, target and backup settings) is saved into rbg files.

To save the results of an action, click the Export button. When prompted, enter a file name with a .txt or .html extension. If you enter a .html extension, the results are saved into an HTML file. When you view the HTML file in a web browser, it will look just like the results are displayed in RegexBuddy. If you enter any other extension, the results are saved as plain text. If you want to save search results without additional information such as the number of matches, set the Target option to save the results to file *before* executing the action. Target files only contain search matches, one on each line.

PowerGREG: The Ultimate GREG for Windows

Since RegexBuddy’s focus is on creating and testing regular expressions, it only offers basic GREG functionality. While certainly useful, you may want to invest in a powerful GREG tool such as PowerGREG. PowerGREG can search using any number of regexes at once, post-process replacement texts during search-and-replace, arbitrarily section files using an extra set of regular expressions, search through proprietary file

formats such as PDF, MS Word and Excel, etc. PowerGREP also keeps a permanent undo history, a feature that can save your day all by itself.

Edit Files

When the GREP panel is showing search results and the cursor is on a file name or on match results that have a file header, then the Edit button (rightmost button) on the GREP toolbar provides various options to open the file. Clicking the button itself opens the file on the Test panel in RegexBuddy. Double-clicking a file name in the GREP results does the same. Double-clicking a search match opens the file on the Test panel and also selects that search match on the Test panel.

The drop-down menu of the Edit button allows you to open the file in external applications. If the file has Open and Edit file associations, then those are shown at the top of the menu. These open the file in its default application just like double-clicking in Windows Explorer does.

The first of the two EditPad items opens just the selected file in EditPad. The second EditPad item opens all files with search matches in EditPad. The Explore Folder command opens the folder that contains the selected file in Windows Explorer.

The bottommost item in the menu opens the file on the Test panel just like clicking the Edit button directly does.

43. History: Experiment with Multiple Regular Expressions

RegexBuddy built-in regex history makes it easy to compare the effects of different variations of a regular expression, or just to work with more than one regex at a time. History items store the regular expression, the replacement text, the application, the emulation mode, and the matching modes.

To create a variant of the current regular expression, click the green plus button on the History toolbar. This duplicates the current regular expression, giving you two identical items in the history. The only difference is that the new regex gets a new label saying “Regex X”. If you click once on the label in the history while it is already selected, an edit box appears allowing you to type in a new name. This works just like renaming files in Windows Explorer.

If you click the Use button on the Library panel, the regular expression you selected in the library is copied into the History as a new entry. This way you can easily compare various regular expressions from the library.

Any changes you make to the regular expression, replacement text or its options are automatically updated in the history. There’s no need to click a button to “save” your changes into the history. Changes you make to a regular expression you selected from the library are not automatically saved back into the library. Only the history is updated automatically. To update the library as well, click the Update button.

There are two buttons with a red X. The button with just a red X deletes the selected regex from the history. The button with the red X over a couple of document sheets clears the entire history. You’ll be left with just one blank entry labeled “Regex 1”.

Use the arrow buttons to change the order of the regular expressions in the history.

If you want to compare two regular expressions side by side, you can start a second instance of RegexBuddy. Simply click the button showing the RegexBuddy logo on the History toolbar. If you set any preferences to preserve various items (they’re all on by default), those things are taken over by the new instance. The history is not automatically synchronized between the two instances. If you close all running instances and then restart RegexBuddy, it will come up with the history as it was in the instance that you closed last.

RegexBuddy’s History panel is intended as a scratch pad. It is *not* intended as a place to preserve your regular expressions. If you have a regex that you want to keep, add it to a library.

The screenshot shows the RegxBuddy application interface. At the top, the title bar reads "RegxBuddy". The menu bar includes "Java 8", "Helpful", "Match", "Replace", and "Split". Below the menu bar, there are several options: "Case sensitive", "Exact spacing", "Dot doesn't match line breaks", "^\$ don't match at line breaks", "Default line breaks", and "Reset".

The main text area contains the following text:

```
"string";  
"two" strings";  
"multi  
line  
string";  
"string with \"quotes\"";
```

The word "string" in the first line is highlighted in yellow. The search bar at the top contains the regex pattern: `"[^\"]*(?:\\. "[^\"]*")*`. The "History" panel on the right shows the following entries:

- Programming: String
- Programming: String (escape quotes)
- Programming: String (multiline; escape quotes)

The bottom toolbar includes "Create", "Convert", "Test", "Debug", "Use", "Library", "GREP", and "Forum". Below the toolbar, there are icons for "Debug", "Highlight", and "List All". The "Highlight" dropdown is set to "Whole file" and "LF only".

The results panel at the bottom shows the following match:

Match	Text	Start	Length
Match 1 of 6:	"string"	0	8

44. Share Experiences and Get Help on The User Forums

When you click the Login button you will be asked for a name and email address. The name you enter is what others see when you post a message to the forum. It is polite to enter your real, full name. The forums are private, friendly and spam-free, so there's no need to hide behind a pseudonym. While you can use an anonymous handle, you'll find that people (other RegexBuddy users) are more willing to help you if you let them know who you are. Support staff from Just Great Software will answer technical support questions anyhow.

The email address you enter is used to email you whenever others participate in one of your discussions. The email address is never displayed to anyone, and is never used for anything other than the automatic notifications. RegexBuddy's forum system does not have a function to respond privately to a message. If you don't want to receive automatic email notifications, there's no need to enter an email address.

If you select "never email replies", you'll never get any email. If you select "email replies to conversations you start", you'll get an email whenever somebody replies to a conversation that you started. If you select "email replies to conversations that you participate in", you'll get an email whenever somebody replies to a conversation that you started or replied to. The From address on the email notifications is forums@jgsoft.com. You can filter the messages based on this address in your email software.

RegexBuddy's forum system uses the standard HTTP protocol which is also used for regular web browsing. If your computer is behind an HTTP proxy, click the Proxy button to configure the proxy connection.

If you prefer to be notified of new messages via an RSS feed instead of email, log in first. After RegexBuddy has connected to the forums, you can click the Feeds button to select RSS feeds that you can add to your favorite feed reader.

Various Forums

Below the Login button, there is a list where you can select which particular forum you want to participate in. The "RegexBuddy" forum is for discussing anything related to the RegexBuddy software itself. This is the place for technical support questions, feature requests and other feedback regarding the functionality and use of RegexBuddy.

The "regular expressions" forum is for discussing regular expressions in general. Here you can talk about creating regular expressions for particular tasks, and exchange ideas on how to implement regular expressions with whatever application or programming language you work with.

Searching The Forums

Before starting a new conversation, please check first if there's already a conversation going on about your topic. In the top right corner of the Forum panel, there is a box on the toolbar that you can use to search for messages. When you type something into that box, only conversations that include at least one message containing the word or phrase you typed in are shown. The filtering happens in real time as you type in your word or phrase.

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

If you find a conversation about your subject, start with reading all the messages in that conversation. If you have any further comments or questions on that conversation, reply to the existing conversation instead of starting a new one. That way, the thread of the conversation stays together, and others can instantly see what you’re talking about. It doesn’t matter if the conversation is a year old. If you reply to it, it moves to the top automatically.

Conversations and Messages

The left hand half of the Forum pane shows two lists. The one at the top shows conversations. The bottom one shows the messages in the selected conversation. You can change the order of the conversations and messages by clicking on the column headers in the lists. A conversation talks about one specific topic. In other forums, a conversation is sometimes called a thread.

If you want to talk about a topic that doesn’t have a conversation yet, click the New button to start a new conversation. A new entry appears in the list of conversations with an edit box. Type in a brief subject for your conversation (up to 100 characters) and press Enter. Please write a clear subject such as “scraping an HTML table in Perl” rather than “need help with HTML” or just “help”. A clear subject significantly increases the odds that somebody who knows the answer will actually click on your conversation, read your question and reply. A generic scream for help only gives the impression you’re too lazy to type in a clear subject, and most forum users don’t like helping lazy people.

After typing in your subject and pressing Enter, the keyboard focus moves to the empty box where you can enter the body text of your message. Please try to be as clear and descriptive as you can. The more information you provide, the more likely you’ll get a timely and accurate answer. If your question is about a particular regular expression, don’t forget to attach your regular expression or test data. Use the forum’s attachment system rather than copying and pasting stuff into your message text.

If you want to reply to an existing conversation, select the conversation and click the Reply button. It doesn’t matter which message in the conversation you selected. Replies are always to the whole conversation rather than to a particular message in a conversation. RegexBuddy doesn’t thread messages like newsgroup software tends to do. This prevents conversations from veering off-topic. If you want to respond to somebody and bring up a different subject, you can start a new conversation, and mention the new conversation in a short reply to the old one.

When starting a reply, a new entry appears in the list of messages. Type in a summary of your reply (up to 100 characters) and press Enter. Then you can type in the full text of your reply, just like when you start a new conversation. However, doing so is optional. If your reply is very brief, simply leave the message body blank. When you send a reply without any body text, the forum system uses the summary as the body text, and automatically prepends [nt] to your summary. The [nt] is an abbreviation for “no text”, meaning the summary is all there is. If you see [nt] on a reply, you don’t need to click on it to see the rest of the message. This way you can quickly respond with “Thank you” or “You’re welcome” and other brief courtesy messages that are often sadly absent from online communication.

When you're done with your message, click the Send button to publish it. There's no need to hurry clicking the Send button. RegexBuddy forever keeps all your messages in progress, even if you close and restart RegexBuddy, or refresh the forums. Sometimes it's a good idea to sleep on a reply if the discussion gets a little heated. You can have as many draft conversations and replies as you want. You can read other messages while composing your reply. If you're replying to a long question, you can switch between the message with the question and your reply while you're writing.

Directly Attach Regexes, Test Data, etc.

One of the greatest benefits of RegexBuddy's built-in forums is that you can directly attach regular expressions, test data, regex libraries, source code templates and GREP actions. Simply click the Attach button and select the item you want to attach. You can add the same item more than once. E.g. to add two regular expressions, click on the first one in the History, click the Attach button and choose Regular Expression. Then click on the second regex and click Attach, Regular Expression again.

To attach a screen shot, press the Print Screen button on the keyboard to capture your whole desktop. Or, press Alt+Print Screen to just capture the active window (e.g. RegexBuddy's window). Then switch to the Forum panel, click the Attach button, and select Clipboard. You can also attach text you copied to the clipboard this way.

It's best to add your attachments while you're still composing your message. The attachments appear with the message, but won't be uploaded until you click the Send button to post your message. If you add an attachment to a message you've written previously, it is uploaded immediately. You cannot attach anything to messages written by others. Write your own reply, and attach your data to that.

To check out an attachment uploaded by somebody else, click the Use or Save button. The Use button loads the attachment directly into RegexBuddy. This may replace your own data. E.g. the Test panel can hold only one test subject, so using an attachment with test data replaces whatever you typed or loaded into the Test panel. If you click the Save button, RegexBuddy prompts for a location to save the attachment. RegexBuddy does not automatically open attachments you save.

RegexBuddy automatically compresses attachments in memory before uploading them. So if you want to attach an external file, there's no need to compress it using a zip program first. If you compress the file manually, everybody who wants to open it has to decompress it manually. If you let RegexBuddy compress it automatically, decompression is also automatic.

Taking Back Your Words

If you regret anything you wrote, simply delete it. There are three Delete buttons. The one above the list of conversations deletes the whole conversation. You can only delete a conversation if nobody else participated in it. The Delete button above the edit box for the message body deletes that message, if you wrote it. The Delete button above the list of attachments deletes the selected attachment, if it belongs to a message that you wrote.

If somebody already downloaded your message before you got around to deleting it, it won't vanish magically. The message will disappear from their view of the forums the next time they log onto the forums or click Refresh. If you see messages disappear when you refresh your own view of the forums, that means the author of the message deleted it. If you replied to a conversation and the original question disappears,

leaving your reply as the only message, you should delete your reply too. Otherwise, your reply will look silly all by itself. When you delete the last reply to a conversation, the conversation itself is also deleted, whether you started it or not.

Changing Your Opinion

If you think you could better phrase something you wrote earlier, select the message and then click the Edit button above the message text. You can then edit the subject and/or body text of the message. Click the Send button to publish the edited message. It will replace the original. If you change your mind about editing the message, click the Cancel button. Make sure to click it only once! When editing a message, the Delete button changes its caption to Cancel and when clicked reverts the message to what it was before you started editing it. If you click Delete a second time (i.e. while the message is no longer being edited), you'll delete the message from the forum.

If other people have already downloaded your message, their view of the message will magically change when they click Refresh or log in again. Since things may get confusing if people respond to your original message before they see the edited message, it's best to restrict your edits to minor errors like spelling mistakes. If you change your opinion, click the Reply button to add a new message to the same conversation.

Updating Your View

When you click the Login button, RegexBuddy automatically downloads all new conversations and message summaries. Message bodies are downloaded one conversation at a time as you click on the conversations. Attachments are downloaded individually when you click the Use or Save button.

RegexBuddy keeps a cache of conversations and messages that persists when you close RegexBuddy. Attachments are cached while RegexBuddy is running, and discarded when you close RegexBuddy. By caching conversations and messages, RegexBuddy improves the responsiveness of the forum while reducing the stress on the forum server.

If you keep RegexBuddy running for a long time, RegexBuddy does not automatically check for new conversations and messages. To do so, click the Refresh button.

Whenever you click Login or Refresh, all conversations and messages are marked as "read". They won't have any special indicator in the list of conversations or messages. If the refresh downloads new conversation and message summaries, those are marked "unread". This is indicated with the same "people" icon as shown next to the Login button.

45. Forum RSS Feeds

When you're connected to the user forum, you can click the Feeds button to select RSS feeds that you can add to your favorite feed reader. This way, you can follow RegexBuddy's discussion forums as part of your regular reading, without having to start RegexBuddy. To participate in the discussions, simply click on a link in the RSS feed. All links in RegexBuddy's RSS feeds will start RegexBuddy and present the forum login screen. After you log in, wait a few moments for RegexBuddy to download the latest conversations. RegexBuddy will automatically select the conversation or message that the link points to. If RegexBuddy was already running and you were already logged onto the forums, the conversation or message that the link points to is selected immediately.

You can choose which conversations should be included in the RSS feed:

- All conversations in all groups: show all conversations in all the discussion groups that you can access in RegexBuddy.
- All conversations in the selected group: show the list of conversations that RegexBuddy is presently showing on the Forum panel.
- All conversations you participated in: show all conversations that you started or replied to in all the discussion groups that you can access in RegexBuddy.
- All conversations you started: show all conversations that you started in all the discussion groups that you can access in RegexBuddy.
- Only the selected conversation: show only the conversation that you are presently reading on the Forum panel in RegexBuddy.

In addition, you can choose how the conversations that you want in your RSS feed should be arranged into items or entries in the feed:

- One item per group, with a list of conversations: Entries link to groups as a whole. The entry titles show the names of the groups. The text of each entry shows a list of conversation subjects and dates. You can click the subjects to participate in the conversation in RegexBuddy. Choose this option if you prefer to read discussions in RegexBuddy itself (with instant access to attachments), and you only want your RSS reader to tell you if there's anything new to be read.
- One item per conversation, without messages: Entries link to conversations. The entry titles show the subjects of the conversations. The entry text shows the date the conversation was started and last replied to. If your feed has conversations from multiple groups, those will be mixed among each other.
- One item per conversation, with a list of messages: Entries link to conversations. The entry titles show the subjects of the conversations. The entry text shows the list of replies with their summaries, author names, and dates. You can click a message summary to read the message in RegexBuddy. If your feed has conversations from multiple groups, those will be mixed among each other.
- One item per conversation, with a list of messages: Entries link to conversations. The entry titles show the subjects of the conversations. The entry text shows the list of replies, each with their full text. If your feed has conversations from multiple groups, those will be mixed among each other. This is the best option if you want to read full discussions in your RSS reader.
- One item per message with its full text: Entries link to messages (responses to conversations). The entry titles show the message summary. The entry text shows the full text of the reply, and the conversation subject that you can click on to open the conversation in RegexBuddy. If your feed lists multiple conversations, replies to different conversations are mixed among each other. Choose this option if you want to read full discussions in your RSS reader, but your RSS reader does not mark old entries as unread when they are updated in the RSS feed.

46. Keyboard Shortcuts

Most of the commands available in RegexBuddy have keyboard shortcuts associated with them. The editor boxes where you can edit your regular expressions, replacement strings, and test subjects recognize many additional text editing keyboard shortcuts not listed here.

Main Toolbar

These keyboard shortcuts work regardless of which panel has keyboard focus in RegexBuddy.

Alt+F	Application drop-down list
Alt+F1	Application dialog box
F3	Toggle Helpful or Strict emulation
Alt+M	Switch to Match mode
Alt+R	Switch to Replace mode
Alt+S	Switch to Split mode
Alt+C	Copy regex or replacement as a string literal
Alt+P	Paste regex or replacement from a string literal
Alt+0	Toggle matching modes (follow up with underlined letters in the popup menu)
Alt+1	Move keyboard focus to the editor box for the regular expression
Alt+1, Tab	Move keyboard focus to the editor box for the replacement text
Alt+2	Show and activate the History panel
Alt+3	Show and activate the Create panel
Alt+4	Show and activate the Convert panel
Alt+5	Show and activate the Test panel
Alt+6	Show and activate the Debug panel
Alt+7	Show and activate the Use panel
Alt+8	Show and activate the Library panel
Alt+9	Show and activate the GREP panel
Alt+0	Show and activate the Forum panel

History Panel

These keyboard shortcuts work regardless of which panel has keyboard focus in RegexBuddy. They do not show or activate the History panel. They perform their task even if the History panel is invisible.

F2	Add a regular expression (duplicates the current regex)
Ctrl+Alt+F2	Delete the current regex
Shift+F2	Move the regex up in the History
Shift+Ctrl+F2	Move the regex down in the History
Shift+Ctrl+N	Start a new instance of RegexBuddy

Create Panel

These keyboard shortcuts work regardless of which panel has keyboard focus in RegexBuddy. Using them automatically shows and activates the Create panel prior to performing their task.

Shift+F3	Toggle the regex tree between Brief and Detailed mode
Alt+I	Open the Insert Token menu (follow up with underlined letters in the menu)
Alt+P	Flavor comparison drop-down list
Ctrl+F3	Flavor comparison dialog box
Alt+X	Export the regex tree

Convert Panel

These keyboard shortcuts work regardless of which panel has keyboard focus in RegexBuddy. Using them automatically shows and activates the Convert panel prior to performing their task.

Alt+V	Flavor conversion drop-down list
Ctrl+F4	Flavor conversion dialog box

This keyboard shortcut only works if the Convert panel has keyboard focus. If any other panel has keyboard focus, the shortcut is handled by the Library panel.

Alt+A	Accept conversion
-------	-------------------

Test Panel

These keyboard shortcuts work regardless of which panel has keyboard focus in RegexBuddy. Using them automatically shows and activates the Test panel prior to performing their task.

Alt+H	Toggle match highlighting
Alt+L	Open the List All menu if RegexBuddy is in Match mode
Alt+L	Open the Replace menu if RegexBuddy is in Replace mode
Alt+L	Test the Split action if RegexBuddy is in Split mode
Shift+F5	Cycle the test scope between whole file, page by page, and line by line
Ctrl+F5	Cycle the line break mode between automatic, CRLF pairs, LF only, and CR only

These keyboard shortcuts only work if the Test panel has keyboard focus. The same shortcuts are used by other panels.

Ctrl+O	Open file
Ctrl+S	Save file under a new name

Use Panel

This keyboard shortcut works whenever the keyboard focus is not on the Library panel, where it performs a different task. Using it automatically shows and activates the Use panel prior to performing its task.

Alt+U Function drop-down list

This keyboard shortcut work regardless of which panel has keyboard focus in RegexBuddy. It does not show or activate the Use panel. It can perform its task even when the Use panel is invisible.

F7 Copy the code snippet to the clipboard

Library Panel

This keyboard shortcut works whenever the keyboard focus is not on the Convert panel, where it performs a different task. Using it automatically shows and activates the Library panel prior to performing its task.

Alt+A Add regex to library

This keyboard shortcut works regardless of which panel has keyboard focus in RegexBuddy. Using it automatically shows and activates the Library panel prior to performing its task.

Alt+A Add regex to library
Alt+E Update regex in library

These keyboard shortcuts only work if the Library panel has keyboard focus. The same shortcuts are used by other panels.

Alt+U Use regex selected in library (copy to History)
Ctrl+N New library
Ctrl+O Open library
Ctrl+S Save library under a new name

GREP Panel

These keyboard shortcuts work regardless of which panel has keyboard focus in RegexBuddy. Using them automatically shows and activates the GREP panel prior to performing their task.

F9 Preview
Ctrl+F9 Execute
Shift+F9 Quick Execute
Ctrl+Alt+F9 Undo (restore files from backups)
Ctrl+Alt+F10 Delete backup files
Shift+F10 Abort

These keyboard shortcuts only work if the Library panel has keyboard focus. The same shortcuts are used by other panels.

Ctrl+N Clear GREP panel
Ctrl+O Open GREP action
Ctrl+S Save GREP action under a new name

Shift+Ctrl+S Export GREP results

This keyboard shortcut work regardless of which panel has keyboard focus in RegexBuddy. It does not show or activate the GREP panel. It can perform its task even when the GREP panel is invisible.

Shift+Ctrl+P

Launch PowerGREP

47. Editing Text with The Keyboard

The boxes where you edit your regular expressions, replacement strings, and test subjects are full-featured edit controls. In fact, they are the same edit control that powers EditPad, one of the most flexible and convenient text editors available today (see <http://www.editpadpro.com>).

Beyond typing in text, the editor box recognizes the following keyboard shortcuts:

Cursor movement keys

Arrow key	Moves the text cursor (blinking vertical bar).
Ctrl+Left arrow	Moves the text cursor to the start of the previous word or the end of the previous line, whichever is closer.
Ctrl+Right arrow	Moves the text cursor to the start of the next line or the end of the current line, whichever is closer.
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.
Ctrl+] presently at.	Moves the text cursor to the bracket that matches the bracket the cursor is presently at.
Ctrl+[cursor.	Selects the text between the innermost pair of brackets that surround the text cursor.
	Repeat the command to select the brackets too.
	Repeat again to select the text between the next level of brackets.

Editing commands

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.
Shift+Ctrl+Delete	All the text on the current line to the right of the text cursor is deleted.

Backspace	Deletes the current selection if there. Otherwise, the character to the left of the text cursor is deleted.
Ctrl+Backspace	Deletes the current selection if there. Otherwise, the part of the current word to the left of the text cursor is deleted.
Shift+Ctrl+Backspace	Deletes the current selection if there is one. Otherwise, all the text on the current line to the left of the text cursor is deleted.
Alt+Backspace	Undo
Alt+Shift+Backspace	Redo
Ctrl+Z	Undo
Ctrl+Y	Redo
Insert	Toggles between insert and overwrite mode.
Tab	If there is a selection, the entire selection is indented. Otherwise, a tab is inserted.
Shift+Tab	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+Alt+Y	Delete Line
Shift+Ctrl+Alt+Y	Duplicate Line

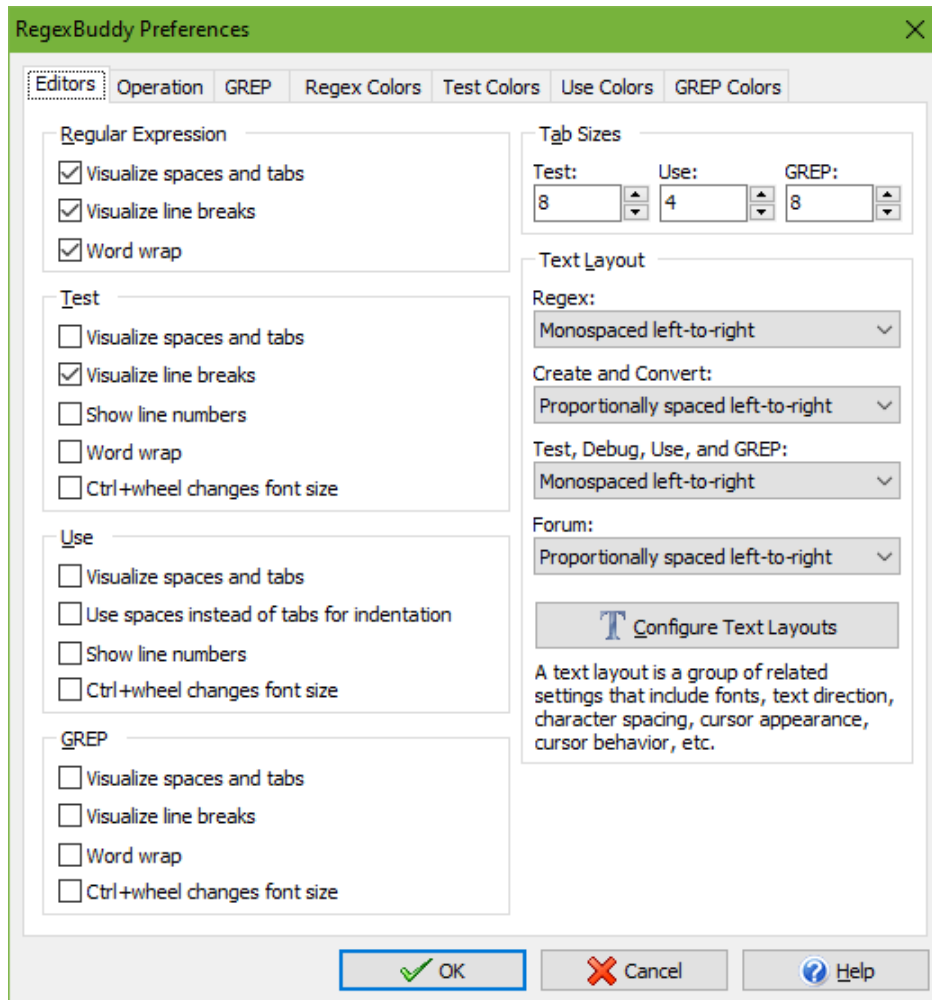
Clipboard commands

Ctrl+X	Cut
Ctrl+C	Copy
Ctrl+V	Paste
Shift+Delete	Cut
Ctrl+Insert	Copy
Shift+Insert	Paste

48. Adjust RegexBuddy to Your Preferences

Click the Preferences button to adjust RegexBuddy to your tastes and habits.

Editors



Regular Expression: Configures the edit boxes for entering the regular expression and the replacement text. You can choose whether spaces and tabs should be indicated by small dots and » signs, and whether line breaks should be indicated by line break symbols.

Test: Configures the edit boxes on the Test panel. You can choose whether spaces and tabs should be indicated by small dots and » signs, and whether line breaks should be indicated by line break symbols. You can also specify if line numbers should be shown in the left margin, and whether long lines should wrap at the edge of the edit box instead of requiring horizontal scrolling.

Use: Configures the edit boxes on the Use panel. You can choose whether spaces and tabs should be indicated by small dots and » signs, whether the tab key should insert spaces or tabs, and whether line numbers should be shown in the left margin.

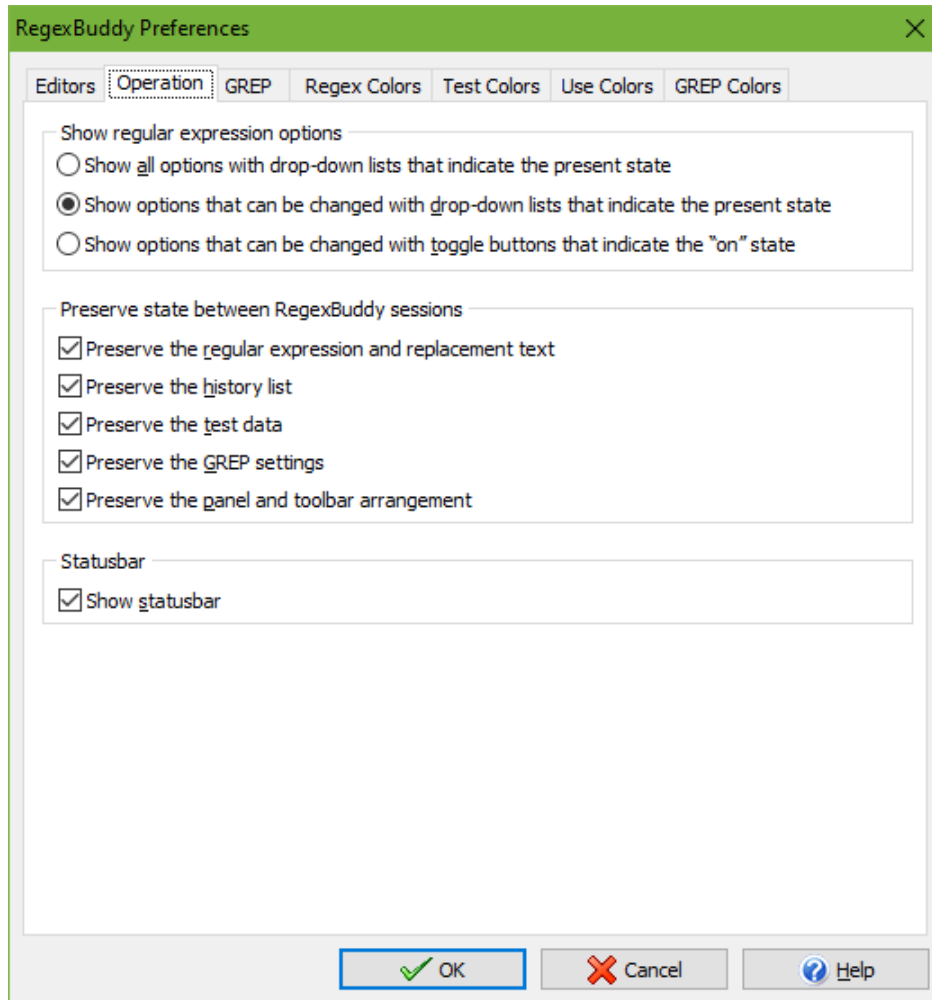
GREP: Configures the results display on the GREP panel. You can choose whether spaces and tabs should be indicated by small dots and » signs, and whether line breaks should be indicated by line break symbols. You can also specify whether long lines should wrap at the edge of the edit box instead of requiring horizontal scrolling.

Tab Sizes: Specify the width in spaces of tabs in the edit boxes on the Test, Use, and GREP panels.

Text Layout: In RegexBuddy, a “text layout” is a combination of settings that control how text is displayed and how the text cursor navigates through that text. The settings include the font, text direction, text cursor behavior, text cursor appearance, which characters are word characters, and how the text should be spaced. You can select up to four different text layouts that are used for various parts of RegexBuddy. The Regex text layout is used for edit boxes for regular expressions and replacement texts. The Convert text layout is used for the warning and error messages on the Convert panel. The font from the Convert text layout is also used for the regex tree on the Create panel. The editors on the Test, Debug, Use, and GREP panels all use the same text layout. You’ll likely want to select a monospaced text layout for this one, so that columns line up correctly. The Forum text layout is used to edit messages on the built-in user forum.

You can select predefined text layouts from the drop-down lists. Or, you can click the Configure Text Layouts button to show the Text Layout Configuration window to customize the text layouts.

Operation



Regex options: RegexBuddy’s main window has a toolbar with a series of drop-down lists or toggle buttons above the regular expression. You can use these to set various regex modes or regex options. This toolbar can be configured in three ways. The default is “show options that can be changed with drop-down lists that indicate the present state”. RegexBuddy then only shows the drop-down lists for the options that the active application allows to be changed. Depending on how many options an application supports, this can significantly reduce the amount of screen space needed for the drop-down lists. It also makes sure you don’t get confused by options that don’t apply to your application.

But if you regularly switch between applications that support different options, you may find it distracting that the drop-down lists move around each time you switch. Then you can choose “show all options with drop-down lists that indicate the present state” to always keep all the drop-down lists visible. If your application doesn’t allow a certain option to be changed, the drop-down list for that option will have the application’s default behavior as its only choice.

The drop-down lists use positive labels to indicate all states. The case sensitivity or insensitivity option, for example, is always indicated as “case sensitive” and “case insensitive”. This avoids confusing double negations like “case insensitive off”. It also avoids confusion when switching between applications where one

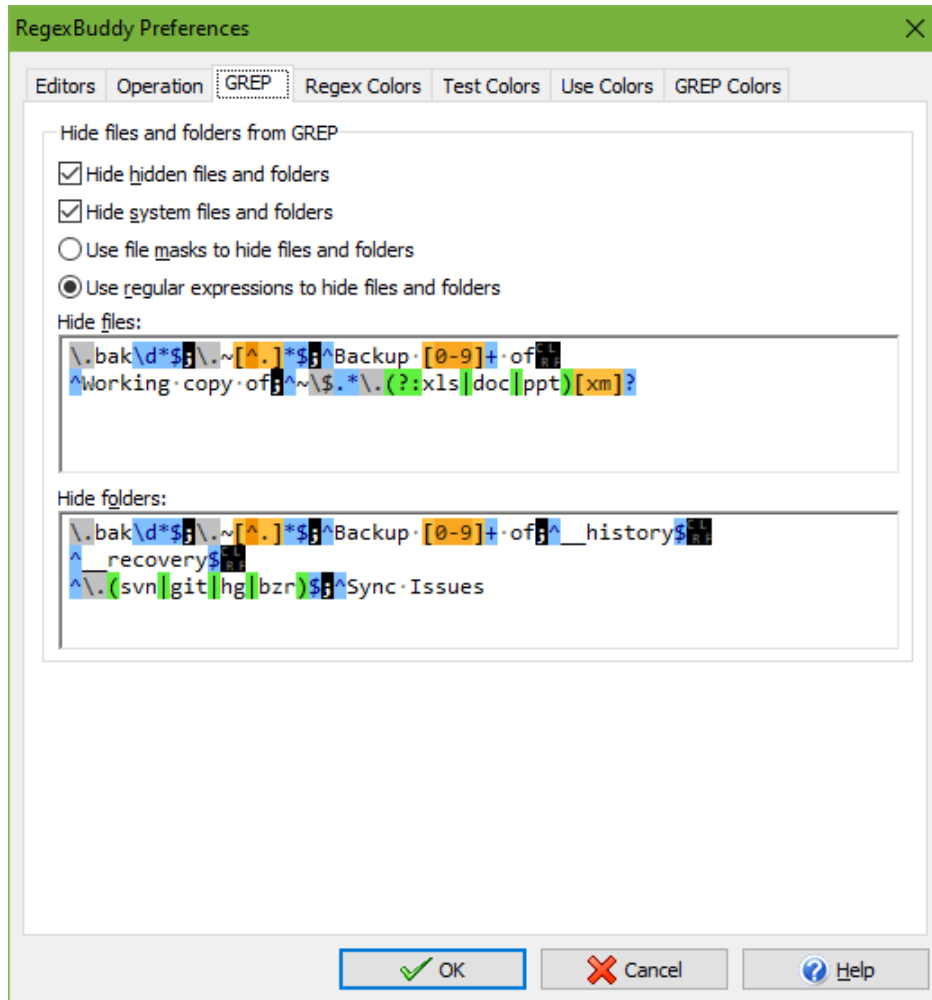
is case sensitive by default with an option to make it case insensitive, and the other is case insensitive by default and has an option to make it case sensitive.

But if the potential confusion of toggle buttons does not bother you, you can choose to “show options that can be changed with toggle buttons that indicate the “on“ state”. The benefit is that the buttons need only one click to toggle an option while drop-down lists require two. The actual application you will use the regex with likely also has toggle buttons or checkboxes rather than drop-down lists. But if you switch between applications in RegexBuddy, you will need to take into account that the meaning of the buttons may invert. If an application is case sensitive by default, the button will say “case insensitive”. If you then switch to an application that is case insensitive by default, the button will change its label to “case sensitive” and will toggle its state (depressed or not). By changing their labels the buttons ensure that the application’s default options are being used when all buttons are off (not depressed).

Preserve state between RegexBuddy sessions: With these checkboxes you can choose what RegexBuddy should remember when you close it and restart it. Turning these on allows you to continue working where you left off. Turning these off allows you to instantly start with a clean slate.

Show statusbar: The statusbar displays helpful hints as you move the mouse over different parts of RegexBuddy’s window, most importantly the toolbar buttons. Leave this option on as you familiarize yourself with RegexBuddy. Later, you can turn it off to save space on the screen.

GREP



The options “hide hidden files and folders” and “hide system files and folders” are turned on by default. These make all files and folders that have the hidden or system attribute set invisible to the GREP panel. Turn these options off if you want to grep through hidden and system files and folders.

Choose “use file masks to hide files and folders” to use traditional wildcard file masks to hide files and folders from the GREP panel. Choose “use regular expressions to hide files and folders” to use regular expressions to hide files and folders. Wildcards have to match the entire name of the file or folder to hide it. Regular expressions hide any file or folder with a name matched by the regex, even if the name is only matched partially. The regular expression “joe”, for example, hides all files that contain “joe” anywhere in the file name. The equivalent file mask using wildcards is “*joe*”. When using wildcards, “joe” hides files exactly named “joe” only.

In the “hide files” box you can enter a semicolon-delimited list of file masks or regular expressions. All files of which the names match one of these file masks are automatically skipped by the GREP panel. By default, the permanent exclusion masks are set to a regular expression that matches all backup files created by RegexBuddy, as well as temporary working copies of files created by EditPad Pro and MS Office. If you’d ever need to search through backup files or working copies, you will need to adjust the permanent exclusion file masks first.

The “hide folders” box works in the same way, except that the file masks you specify are tested against individual folder names. If the name of a folder matches one of these file masks then the GREP panel automatically skips that folder and any files or folders inside that folder. The default settings exclude `__history`, which is the folder name that RegexBuddy uses for backup files when you select the “hidden history” backup option. Also excluded by default is `.svn` which is the folder name that Subversion uses to cache data in each folder that you have added to version control.

Regex Colors

Configure the colors used to highlight different parts of regular expressions and replacement texts. 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.

Test Colors

Configure the colors used by the Highlight button on the Test panel. You can select one of the preset configurations at the top of the screen. To customize the colors, click on an item in the list of individual colors. Then click the Background Color and Text Color buttons to change the item’s colors. You can also change the font style with the bold, italic and underline checkboxes. At the bottom, a sample edit box will show what the color configuration looks like. You can edit the text in the example box to further test the colors. The regex `match(?:.*?(with capturing group))?` is used to highlight the sample text.

Use Colors

Configure the colors used by syntax coloring of the code snippets on the Use panel. You can select one of the preset configurations at the top of the screen. To customize the colors, click on an item in the list of individual colors. Then click the Background Color and Text Color buttons to change the item’s colors. You can also change the font style with the bold, italic and underline checkboxes. At the bottom, a sample edit box will show what the color configuration looks like. You can see the effect for several different programming languages. You can edit the text in the example box to further test the colors.

GREP Colors

Configure the colors used by the results display on the GREP panel. You can select one of the preset configurations at the top of the screen. To customize the colors, click on an item in the list of individual colors. Then click the Background Color and Text Color buttons to change the item’s colors. You can also change the font style with the bold, italic and underline checkboxes. At the bottom, a sample results display will show what the color configuration looks like.

49. Text Layout Configuration

In RegxBuddy, a “text layout” is a combination of settings that control how text is displayed and how the text cursor navigates through that text. The settings include the font, text direction, text cursor behavior, text cursor appearance, which characters are word characters, and how the text should be spaced.

On the Editors tab in the Preferences, you can select up to four different text layouts that are used for various parts of RegxBuddy. The Regex text layout is used for edit boxes for regular expressions and replacement texts. The Convert text layout is used for the warning and error messages on the Convert panel. The font from the Convert text layout is also used for the regex tree on the Create panel. The editors on the Test, Debug, Use, and GREP panels all use the same text layout. You’ll likely want to select a monospaced text layout for this one, so that columns line up correctly. The Forum text layout is used to edit messages on the built-in user forum.

You can select predefined text layouts from the drop-down lists. Or, you can click the Configure Text Layouts button to show the Text Layout Configuration window to customize the text layouts.

Text Layout Configuration

Select the text layout configuration that you want to use

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

+ New
X Delete
Up
Down

Selected text layout configuration

Name of this text layout configuration:
Monospaced left-to-right

Example:
Sample text to test the text layout configuration

Text layout and direction

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

Text cursor movement

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

Selection of words

- Select only the word
- Select the word plus everything up to the next word

Character sequences to treat as words

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

Text cursor appearance

Insert mode text cursor:
Insertion cursor Configure

Overwrite mode text cursor:
Overwrite cursor Configure

Main font

- Allow bitmapped fonts
- Consolas
- Bold Italic
- Size: 10

Fallback fonts (complex script only)

+ Add
X Delete
Up
Down

Line and character spacing

Increase (or decrease) the line height by 0 pixels

Add 0 pixels of extra space between lines

Increase (or decrease) the character width by 0 pixels

OK Cancel Help

Select The Text Layout Configuration That You Want to Use

The Text Layout Configuration screen shows the details of the text layout configuration that you select in the list in the top left corner. Any changes you make on the screen are automatically applied to the selected layout and persist as you choose different layouts in the list. The changes become permanent when you click OK. The layout that is selected in the list when you click OK becomes the new default layout.

Click the New and Delete buttons to add or remove layouts. You must have at least one text layout configuration. If you have more than one, you can use the Up and Down buttons to change their order. The order does not affect anything other than the order in which the text layouts configurations appear in selection lists.

RegexBuddy comes with a number of preconfigured text layouts. If you find the options on this screen bewildering, simply choose the preconfigured layout that matches your needs, and ignore all the other settings. You can fully edit and delete all the preconfigured text layouts if you don't like them.

- Left-to-right: Normal settings with best performance for editing text in European languages and ideographic languages (Chinese, Japanese, Korean). The default font is monospaced. The layout does respect individual character width if the font is not purely monospaced or if you select another font.
- Proportionally spaced left-to-right: Like left-to-right, but the default font is proportionally spaced.
- Monospaced left-to-right: Like left-to-right, but the text is forced to be monospaced. Columns are guaranteed to line up perfectly even if the font is not purely monospaced. This is the best choice for working with source code and text files with tabular data.
- Monospaced ideographic width: Like monospaced left-to-right, but ASCII characters are given the same width as ideographs. This is the best choice if you want columns of mixed ASCII and ideographic text to line up perfectly.
- Complex script left-to-right: Supports text in any language, including complex scripts (e.g. Indic scripts) and right-to-left scripts (Hebrew, Arabic). Choose this for editing text that is written from left-to-right, perhaps mixed with an occasional word or phrase written from right-to-left.
- Complex script right-to-left: For writing text in scripts such as Hebrew or Arabic that are written from right-to-left, perhaps mixed with an occasional word or phrase written from left-to-right.
- Monospaced complex left-to-right: Like “complex script left-to-right”, but using monospaced fonts for as many scripts as possible. Text is not forced to be monospaced, so columns may not line up perfectly.
- Monospaced complex right-to-left: Like “complex script right-to-left”, but using monospaced fonts for as many scripts as possible. Text is not forced to be monospaced, so columns may not line up perfectly.

Selected Text Layout Configuration

The section in the upper right corner provides a box to type in the name of the text layout configuration. This name is only used to help you identify it in selection lists when you have prepared more than one text layout configuration.

In the Example box you can type in some text to see how the selected text layout configuration causes the editor to behave.

Text Layout and Direction

- Complex script, predominantly left-to-right: Text is written from left to right and can be mixed with text written from right to left. Choose this for complex scripts such as the Indic scripts, or for text in any language that mixes in the occasional word or phrase in a right-to-left or complex script.
- Complex script, predominantly right-to-left: Text is written from right to left and can be mixed with text written from left to right. Choose this for writing text in scripts written from right to left such as Hebrew or Arabic.
- Left-to-right only: Text is always written from left to right. Complex scripts and right-to-left scripts are not supported. Choose this for best performance for editing text in European languages and ideographic languages (Chinese, Japanese, Korean) that is written from left to right without exception.
- Monospaced left-to-right only: Text is always written from left to right and is forced to be monospaced. Complex scripts and right-to-left scripts are not supported. Each character is given the same horizontal width even if the font specifies different widths for different characters. This guarantees columns to be lined up perfectly. To keep the text readable, you should choose a monospaced font.
- ASCII characters with full ideographic width: You can choose this option in combination with any of the four preceding options. In most fonts, ASCII characters (English letters, digits, and punctuation) are about half the width of ideographs. This option substitutes full-width characters for the ASCII characters so they are the same width as ideographs. If you turn this on in combination with “monospaced left-to-right only” then columns that mix English letters and digits with ideographs will line up perfectly.

Text Cursor Movement

- Monodirectional: The left arrow key on the keyboard always moves the cursor to the left on the screen and the right arrow key always moves the cursor to the right on the screen, regardless of the predominant or actual text direction.
- Bidirectional: This option is only available if you have chosen one of the complex script options in the “text layout and direction” list. The direction that the left and right arrow keys move the cursor into depends on the predominant text direction selected in the “text layout and direction” list and on the actual text direction of the word that the cursor is pointing to when you press the left or right arrow key on the keyboard.
 - Predominantly left-to-right: The left key moves to the preceding character in logical order, and the right key moves to the following character in logical order.
 - Actual left-to-right: The left key moves left, and the right key moves right.
 - Actual right-to-left: The actual direction is reversed from the predominant direction. The left key moves right, and the right key moves left.
 - Predominantly right-to-left: The left key moves to the following character in logical order, and the right key moves to the preceding character in logical order.
 - Actual left-to-right: The actual direction is reversed from the predominant direction. The left key moves right, and the right key moves left.
 - Actual right-to-left: The left key moves left, and the right key moves right.

Selection of Words

- Select only the word: Pressing Ctrl+Shift+Right moves the cursor to the end of the word that the cursor is on. The selection stops at the end of the word. This is the default behavior for all Just Great Software applications. It makes it easy to select a specific word or string of words without any extraneous spaces or characters. To include the space after the last word, press Ctrl+Shift+Right once more, and then Ctrl+Shift+Left.
- Select the word plus everything to the next word: Pressing Ctrl+Shift+Right moves the cursor to the start of the word after the one that the cursor is on. The selection includes the word that the cursor was on and the non-word characters between that word and the next word that the cursor is moved to. This is how text editors usually behave on the Windows platform.

Character Sequences to Treat as words

- Letters, digits, and underscores: Characters that are considered to be letters, digits, or underscores by the Unicode standard are selected when you double-click them. Ctrl+Left and Ctrl+Right move the cursor to the start of the preceding or following sequence of letters, digits, or underscores. If symbols or punctuation appear adjacent to the start of a word, the cursor is positioned between the symbol and the first letter of the word. Ideographs are considered to be letters.
- Letters, digits, and symbols: As above, but symbols other than punctuation are included in the selection when double-clicking. Ctrl+Left and Ctrl+Right never put the cursor between a symbol and another word character.
- Everything except whitespace: All characters except whitespace are selected when you double-click them. Ctrl+Left and Ctrl+Right move the cursor to the preceding or following position that has a whitespace character to the left of the cursor and a non-whitespace character to the right of the cursor.
- Words determined by complex script analysis: If you selected the “bidirectional” text cursor movement option, you can turn on this option to allow Ctrl+Left and Ctrl+Right to place the cursor between two letters for languages such as Thai that don’t write spaces between words.

Text Cursor Appearance

Select a predefined cursor or click the Configure button to show the text cursor configuration screen. There you can configure the looks of the blinking text cursor (and even make it stop blinking).

A text layout uses two cursors. One cursor is used for insert mode, where typing in text pushes ahead the text after the cursor. The other cursor is used for overwrite mode, where typing in text replaces the characters after the cursor. Pressing the Insert key on the keyboard toggles between insert and overwrite mode.

Main Font

Select the font that you want to use from the drop-down list. Turn on “allow bitmapped fonts” to include bitmapped fonts in the list. Otherwise, only TrueType and OpenType fonts are included. Using a TrueType or OpenType font is recommended. Bitmapped fonts may not be displayed perfectly (e.g. italics may be clipped) and only support a few specific sizes.

If you access the text layout configuration screen from the print preview, then turning on “allow bitmapped fonts” will include printer fonts rather than screen fonts in the list, in addition to the TrueType and OpenType fonts that work everywhere. A “printer font” is a font built into your printer’s hardware. If you select a printer font, set “text layout and direction” to “left to right only” for best results.

Fallback Fonts

Not all fonts are capable of displaying text in all scripts or languages. If you have selected one of the complex script options in the “text layout and direction” list, you can specify one or more “fallback” fonts. If the main font does not support a particular script, RegexBuddy will try to use one of the fallback fonts. It starts with the topmost font at the list and continues to attempt fonts lower in the list until it finds a font that supports the script you are typing with. If none of the fonts supports the script, then the text will appear as squares.

To figure out which scripts a particular font supports, first type or paste some text using those scripts into the Example box. Make sure one of the complex script options is selected. Then remove all fallback fonts. Now you can change the main font and see which characters that font can display. When you’ve come up with a list of fonts that, if used together, can display all of your characters, select your preferred font as the main font. Then add all the others as fallback fonts.

Line and Character Spacing

By default all the spacing options are set to zero. This tells RegexBuddy to use the default spacing for the font you have selected.

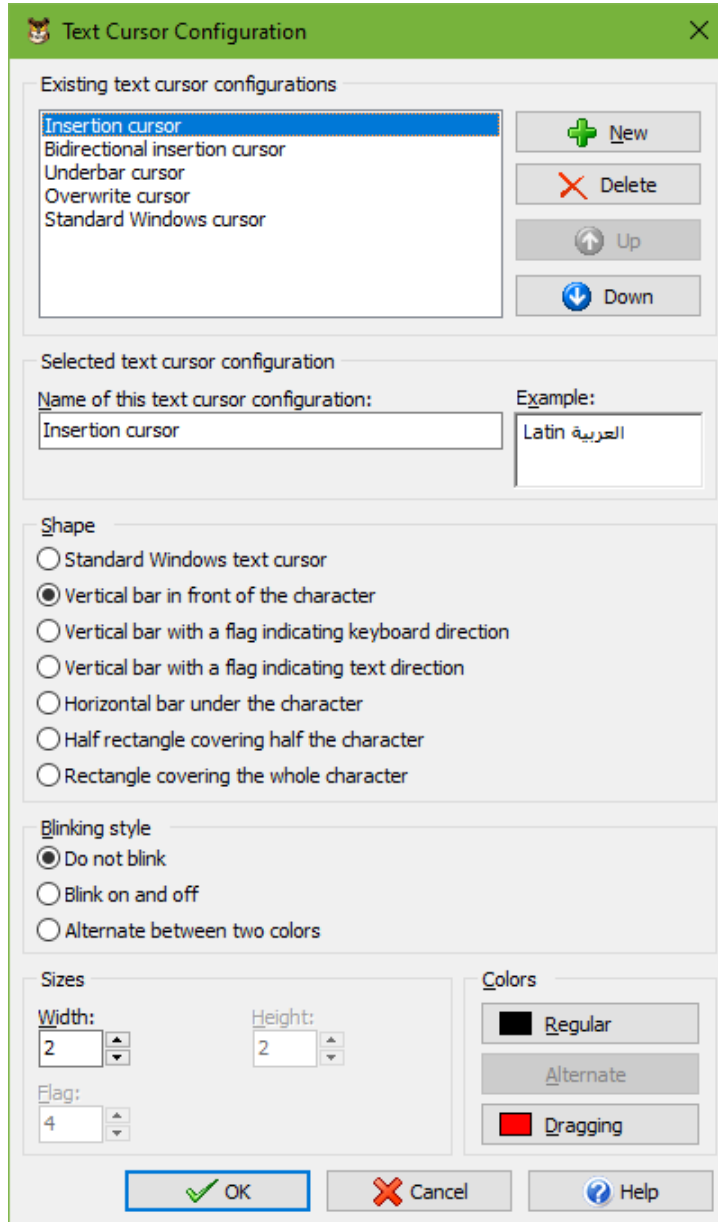
If you find that lines are spaced apart too widely, specify a negative value for “increase (or decrease) the line height”. Set to “add 0 pixels of extra space between lines”.

If you find that lines are spaced too closely together, specify a positive value for “increase (or decrease) the line height” and/or “add ... pixels of extra space between lines”. The difference between the two is that when you select a line of text, increasing the line height increases the height of the selection highlighting, while adding extra space between lines does not. If you select multiple lines of text, extra space between lines shows up as gaps between the selected lines. Adding extra space between lines may make it easier to distinguish between lines.

The “increase (or decrease) the character width by ... pixels” setting is only used when you select “monospaced left-to-right” only in the “text layout and direction” list. You can specify a positive value to increase the character or column width, or a negative value to decrease it. This can be useful if your chosen font is not perfectly monospaced and because of that characters appear spaced too widely or too closely.

50. Text Cursor Configuration

You can access the text cursor configuration screen from the text layout configuration screen by clicking one of the Configure buttons in the “text cursor appearance” section.



Existing Text Cursor Configurations

The Text Cursor Configuration screen shows the details of the text cursor configuration that you select in the list at the top. Any changes you make on the screen are automatically applied to the selected cursor and persist as you choose different cursors in the list. The changes become permanent when you click OK. The cursor that is selected in the list when you click OK becomes the new default cursor.

Click the New and Delete buttons to add or remove cursors. You must have at least one text cursor configuration. If you have more than one, you can use the Up and Down buttons to change their order. The order does not affect anything other than the order in which the text cursor configurations appear in selection lists.

RegexBuddy comes with a number of preconfigured text cursors. You can fully edit or delete all the preconfigured text cursors if you don't like them.

- Insertion cursor: Blinking vertical bar similar to the standard Windows cursor, except that it is thicker and fully black, even on a gray background.
- Bidirectional insertion cursor: Like the insertion cursor, but with a little flag that indicates whether the keyboard layout is left-to-right (e.g. you're typing in English) or right-to-left (e.g. you're typing in Hebrew). The flag is larger than what you get with the standard Windows cursor and is shown even if you don't have any right-to-left layouts installed.
- Underbar cursor: Blinking horizontal bar that lies under the character. This mimics the text cursor that was common in DOS applications.
- Overwrite cursor: Blinking rectangle that covers the bottom half of the character. In RegexBuddy this is the default cursor for overwrite mode. In this mode, which is toggled with the Insert key on the keyboard, typing text overwrites the following characters instead of pushing them ahead.
- Standard Windows cursor: The standard Windows cursor is a very thin blinking vertical bar that is XOR-ed on the screen, making it very hard to see on anything except a pure black or pure white background. If you have a right-to-left keyboard layout installed, the cursor gets a tiny flag indicating keyboard direction. You should only use this cursor if you rely on accessibility software such as a screen reader or magnification tool that fails to track any of RegexBuddy's other cursor shapes.

Selected Text Cursor Configuration

Type in the name of the text cursor configuration. This name is only used to help you identify it in selection lists when you have prepared more than one text cursor configuration.

In the Example box you can type in some text to see what the cursor looks like. The box has a word in Latin and Arabic so you can see the difference in cursor appearance, if any, based on the text direction of the word that the cursor is on.

Shape

- Standard Windows Text cursor: The standard Windows cursor is a very thin blinking vertical bar that is XOR-ed on the screen, making it very hard to see on anything except a pure black or pure white background. If you have a right-to-left keyboard layout installed, the cursor gets a tiny flag indicating keyboard direction. You should only use this cursor if you rely on accessibility software such as a screen reader or magnification tool that fails to track any of RegexBuddy's other cursor shapes. The standard Windows cursor provides no configuration options.
- Vertical bar in front of the character: On the Windows platform, the normal cursor shape is a vertical bar that is positioned in front of the character that it points to. That is to the left of the character for left-to-right text, and to the right of the character for right-to-left text.
- Vertical bar with a flag indicating keyboard direction: A vertical bar positioned in front of the character that it points to, with a little flag (triangle) at the top that indicates the direction of the active keyboard layout. When the cursor points to a character in left-to-right text, it is placed to the

left of that character. When the cursor point to a character in right-to-left text, it is placed to the right of that character. The direction of the cursor's flag is independent of the text under the cursor. The cursor's flag points to the right when the active keyboard layout is for a left-to-right language. The cursor's flag points to the left when the active keyboard layout is for a right-to-left language.

- Vertical bar with a flag indicating text direction: A vertical bar positioned in front of the character that it points to, with a little flag (triangle) at the top that points to that character. When the cursor points to a character in left-to-right text, it is placed to the left of that character with its flag pointing to the right towards that character. When the cursor point to a character in right-to-left text, it is placed to the right of that character with its flag pointing to the left towards that character.
- Horizontal bar under the character: In DOS applications, the cursor was a horizontal line under the character that the cursor points to.
- Half rectangle covering half the character: The cursor covers the bottom half of the character that it points to. This is a traditional cursor shape to indicate typing will overwrite the character rather than push it ahead.
- Rectangle covering the whole character: The cursor makes the character invisible. This can also be used to indicate overwrite mode.

Blinking Style

- Do not blink: The cursor is permanently visible in a single color. Choose this option if the blinking distracts you or if it confuses accessibility software such as screen readers or magnification tools.
- Blink on and off: The usual blinking style for text cursors on the Windows platform. The cursor is permanently visible while you type (quickly). When you stop typing for about half a second, the cursor blinks by becoming temporarily invisible. Blinking makes it easier to locate the cursor with your eyes in a large block of text.
- Alternate between two colors: Makes the cursor blink when you stop typing like “on and off”. But instead of making the cursor invisible, it is displayed with an alternate color. This option gives the cursor maximum visibility: the blinking animation attracts the eye while keeping the cursor permanently visible.

Sizes

- Width: Width in pixels for the vertical bar shape.
- Height: Height in pixels for the horizontal bar shape.
- Flag: Length in pixels of the edges of the flag that indicates text direction.

Colors

- Regular: Used for all shapes and blinking styles except the standard Windows cursor.
- Alternate: Alternate color used by the “alternate between two colors” blinking style.
- Dragging: Color of a second “ghost” cursor that appears while dragging and dropping text with the mouse. It indicates the position the text is moved or copied to when you release the mouse button.

51. Integrate RegexBuddy with Searching, Editing and Coding Tools

RegexBuddy is designed to be used as a companion to whatever software you use regular expressions with. Such applications include search tools, text editing and processing tools, programming and development tools, etc. Whenever you need to write or edit a regular expression while working with those applications, RegexBuddy pops up to provide assistance, and disappears when you finish editing the regex.

Without integrating RegexBuddy with your tools, this workflow takes many steps:

1. In your tool, copy the regular expression to the clipboard.
2. Start RegexBuddy via a desktop or start menu shortcut.
3. Paste the regular expression into RegexBuddy, manually selecting the correct format.
4. Edit the regular expression.
5. Copy the regular expression from RegexBuddy to the clipboard, manually selecting the correct format.
6. Close RegexBuddy.
7. Paste the regular expression into your tool.

Basic Integration

Integrating RegexBuddy with your favorite tools can significantly speed up this workflow by automating most of the steps. The simplest form of integration is by starting RegexBuddy with certain command line parameters. This method is appropriate when you want to integrate RegexBuddy with a 3rd party tool. If the tool has the ability to run external applications, you can easily launch RegexBuddy from within the tool.

E.g. in Microsoft Visual Studio, you could select Tools|External Tools from the menu, and specify the following arguments: `-getfromclipboard -putonclipboard -app vbnet20 -appname "Visual Studio"`. Use `-app csharp20` instead of `-app vbnet20` if you develop in C# rather than Visual Basic.

When adding RegexBuddy to Borland Delphi's Tools menu, you can specify `-getfromclipboard -putonclipboard -app delphixe4 -appname "Delphi XE4"` for the parameters. This selects the regex flavor used by TRegEx.

This style of integration, which only takes a few minutes to implement, reduces the workflow to:

1. In your tool, copy the regular expression to the clipboard.
2. Start RegexBuddy from a menu item, button or keyboard shortcut in your tool.
3. Edit the regular expression, which RegexBuddy automatically grabs from the clipboard, in the right format.
4. Click the Send To button in RegexBuddy, which closes RegexBuddy and automatically puts the modified regex on the clipboard, in the right format.
5. Paste the regular expression into your tool.

Though you still need to copy and paste the regular expression from and into your tool, the reduced workflow is a lot more convenient. You can launch RegexBuddy from within your tool, you no longer need

to copy and paste the regex in RegexBuddy, and RegexBuddy automatically uses the correct regex flavor and string style for your tool.

Advanced Integration

Tighter integration is possible with tools that have a programming interface (API), and with software that you develop yourself. RegexBuddy provides a COM Automation interface, which you can easily import and call from any development tool or language that supports COM. The interface enables you to launch RegexBuddy, and send and receive regex actions in response to the user's actions.

Except for actually editing the regular expression, the workflow is automated entirely:

1. Start RegexBuddy from a menu item, button or keyboard shortcut in your tool. This automatically sends the regex to RegexBuddy.
2. Edit the regular expression.
3. Click the Send To button in RegexBuddy, which automatically updates the regex in your tool, and closes RegexBuddy.

If you are a software developer, and some of your products support regular expressions, make your customers happy and provide tight integration with RegexBuddy. Take a look at PowerGREP and EditPad Pro to see how convenient they make it to edit a regular expression with RegexBuddy.

52. Basic Integration with RegexBuddy

Integrating RegexBuddy with your favorite tools can significantly speed up this workflow by automating most of the steps. The simplest form of integration is by starting RegexBuddy with certain command line parameters. This method is appropriate when you want to integrate RegexBuddy with a 3rd party tool. If the tool has the ability to run external applications, you can easily launch RegexBuddy from within the tool.

RegexBuddy's command line parameters are case insensitive. The order in which they appear is irrelevant, except for parameters that must be followed by a second "data" parameter. The second parameter must immediately follow the first parameter, separated with a space. Values with spaces in them must be delimited by a pair of double quotes. Parameters can start with a hyphen or a forward slash.

`-getfromclipboard`

RegexBuddy will use the text on the clipboard as the regular expression. The text is used "as is", unless you use the `-app` or `-string` parameters to specify a string style.

`-putonclipboard`

RegexBuddy will show the Send button. When clicked, the regex is copied to the clipboard. The regex is copied "as is", unless you use the `-app` or `-string` parameters to specify a string style.

`-delimitclipboard delimiter`

Specify a string to use as a delimiter so that more text than just the regular expression can be passed via the clipboard. Your chosen delimiter must not be used in the regular expression or replacement text. Something like `***?***?` is a good choice because it's not something people can or would use in a regex.

If you do not use the `-delimitclipboard` parameter, then the all the text on the clipboard is used as the regular expression. When clicking the Send button, RegexBuddy only puts the regex on the clipboard.

If you use `-delimitclipboard` then the clipboard is then interpreted as "regexdelimiteroptionsdelimiterreplacement". The regex and replacement are interpreted using the string style specified by the `-app` or `-string` parameter. The options string is checked for the presence or absence of the following substrings. If a substring is present, that matching mode is turned on. If a substring is absent, that matching mode is turned off. For the line break modes, specifying one of them selects that line break mode. Specifying none of them selects "default" as the line break mode. If your application or regex flavor does not allow a particular mode to be toggled, you don't need to specify it, even if it is always on.

<code>caseless</code>	Case insensitive
<code>freespacing</code>	Free-spacing regex
<code>dotall</code>	Dot matches line breaks
<code>multiline</code>	<code>^</code> and <code>\$</code> match at line breaks
<code>lbfonly</code>	LF only line break mode
<code>lbcronly</code>	CR only line break mode

<code>lbcrlfonly</code>	CRLF pairs only line break mode
<code>lbcrlf</code>	CR, LF, or CRLF line break mode
<code>lbunicode</code>	Unicode line break mode
<code>explicitcapture</code>	Parentheses do not create numbered groups
<code>namedduplicate</code>	Named capturing groups can have the same name
<code>ungreedy</code>	Quantifiers are lazy and appending <code>?</code> makes them greedy
<code>skipzerolength</code>	Skip zero-length matches
<code>stringsyntax</code>	Support literal string escapes on top of the regex or replacement text syntax
<code>splitlimit1234</code>	Split action with 1234 as the limit; 1234 can be any positive or negative integer; omit 1234 to split without a limit
<code>splitcapture</code>	Add capturing groups to the resulting array when splitting a string
<code>splitempty</code>	Add empty strings to the resulting array when splitting a string

Putting “`regexdelimiteroptionsdelimiterreplacement`” on the clipboard will put `RegexBuddy` in Replace mode even if the replacement string is blank. Include `splitlimit` in the options to put `RegexBuddy` in Split mode. Omit the second delimiter, the replacement text, and omit `splitlimit` from the options to put `RegexBuddy` in Match mode. `RegexBuddy` will do the same when putting the modified action on the clipboard.

`RegexBuddy` does not care how or whether you delimit the options, whether you format them as a string, or in which order you place them. `RegexBuddy` will put them on the clipboard as a comma-delimited list in the order shown in the table. `RegexBuddy` lists all options that are turned on, even if the application or regex flavor doesn’t allow them to be turned off.

`-app app`

Specify the application that your regular expression should work with. The *app* value must be an application identifier. `RegexBuddy` will use this application’s string style, regex flavor, replacement flavor, split flavor, and source code template. If there is no application identifier for the combination of string style and flavors that you want, then you cannot use the `-app` parameter. You can use the separate `-string` and `-flavor` parameters to specify custom combinations.

`-string stringstyle`

Tell `RegexBuddy` to use a certain string format for both `-getfromclipboard` and `-putonclipboard`. The *stringstyle* value must be a string style identifier. If you use both `-app` and `-string`, then `-string` takes precedence.

`-flavor flavor`

Set the regular expression flavor you’re working with. The *flavor* value must be a valid regex flavor identifier. If you use both `-app` and `-flavor`, then `-flavor` takes precedence.

`-flavorreplace flavor`

Set the replacement text flavor you’re working with. The *flavor* value must be a valid replacement flavor identifier. If you use both `-app` and `-flavorreplace`, then `-flavorreplace` takes

precedence. `-flavorreplace` is ignored unless you also use `-app` or `-flavor` to set the regex flavor.

`-flavorsplit flavor`

Set the split flavor you're working with. The *flavor* value must be a valid split flavor identifier. If you use both `-app` and `-flavorsplit`, then `-flavorsplit` takes precedence. `-flavorsplit` is ignored unless you also use `-app` or `-flavor` to set the regex flavor.

`-helpful`

Set the flavor emulation mode to Helpful.

`-strict`

Set the flavor emulation mode to Strict.

`-template filename.rbsct`

Select the RegexBuddy source code template for the Use panel. You can specify the file name of the template without the path and without the extension. If you use both `-app` and `-template`, then `-template` takes precedence.

`-convertapp app`

Specify the application that your regular expression should be converted to. The *app* value must be an application identifier. RegexBuddy will select this application on the Convert panel. If there is no application identifier for the combination of string style and flavors that you want, then you cannot use the `-convertapp` parameter. You can use the `-convertflavor` parameters to specify custom combinations.

`-convertflavor flavor`

Regular expression flavor that RegexBuddy should select on the Convert panel. The *flavor* value must be a valid regex flavor identifier. If you use both `-convertapp` and `-convertflavor`, then `-convertflavor` takes precedence.

`-convertflavorreplace flavor`

Replacement text flavor that RegexBuddy should select on the Convert panel. The *flavor* value must be a valid replacement flavor identifier. If you use both `-convertapp` and `-convertflavorreplace`, then `-convertflavorreplace` takes precedence. `-convertflavorreplace` is ignored unless you also use `-convertapp` or `-convertflavor` to set the regex flavor.

`-convertflavorsplit flavor`

Split flavor that RegexBuddy should select on the Convert panel. The *flavor* value must be a valid split flavor identifier. If you use both `-convertapp` and `-convertflavorsplit`, then `-convertflavorsplit` takes precedence. `-convertflavorsplit` is ignored unless you also use `-convertapp` or `-convertflavor` to set the regex flavor.

-convert

Tell RegexBuddy that the regular expression that it puts on the clipboard must be converted to the required flavor. If you're using the `-convertapp` or `-convertflavor` parameters, then the regex must be converted to the regex flavor specified by those parameters. If not, then the regex must be converted to the flavor specified by `-app` or `-appflavor`. If you don't use any of these 4 parameters to specify a regex flavor or you don't use `-putonclipboard` then the `-convert` parameter is ignored.

If you use this parameter and then click the Send button while a different flavor is selected in the toolbar at the top, RegexBuddy attempts to automatically convert the regex. If the conversion succeeds without warnings, the converted regex is put on the clipboard and RegexBuddy closes. If the conversion fails or has warnings, those are shown on the Convert panel with an additional message that the Send button requires the regex to be converted.

-convertfreespacing

Use in combination with `-convertapp` or `-convertflavor` to select "free-spacing" on the Convert panel. Omit to select "exact spacing".

-convertstripcomments

Use in combination with `-convertapp` or `-convertflavor` to select "strip comments" on the Convert panel. Omit to select "keep comments".

-appname "Application Name"

The `-appname` parameter must be immediately followed with a parameter that indicates the name of the application that is invoking RegexBuddy. RegexBuddy will show this name in its caption bar, to make it clear which application the Regex will be sent to. You should always specify `-appname` when using `-putonclipboard`.

-action filename.rba

Loads the regular expression from the specified RegexBuddy action file. Use this parameter if you want to associate RegexBuddy with rba files. This file format is used to share individual regular expressions on the RegexBuddy user forum. If you use `-action` then the `-app` through `-strict` parameters (as listed above) are ignored, because the action file already specifies those.

-testfile document.txt

Loads the file as the test subject on the Test panel.

-testclipboard

Uses the text on the clipboard as the test subject on the Test panel. The text is always used "as is". Should not be used in combination with `-getfromclipboard`.

-library filename.rbl

Opens the specified library on the Library panel. Use this parameter if you want to associate RegexBuddy with rbl files.

`-grep filename.rbg`

Opens the specified GREP action on the GREP panel. Use this parameter if you want to associate RegexBuddy with rbg files.

`-activate 1234`

Use this parameter in combination with `-putonclipboard`. `-putonclipboard` must be followed by a decimal unsigned integer that indicates the window handle of the active top-level form in your application. When the Send To button is clicked, RegexBuddy will use the `SetForegroundWindow()` Win32 API call to bring your application to the front. That way, the Sent To button enables you to continue working with that application right away.

53. Integrating RegexBuddy Using COM Automation

If you are a software developer, and some of your products support regular expressions, make your customers happy and provide tight integration with RegexBuddy. That way, they can edit the regexes they use in your software with RegexBuddy at the click of a button, without having to manually copy and paste regexes between your software and RegexBuddy.

On the Microsoft Windows platform, RegexBuddy provides a COM Automation interface, which you can easily import and call from any development tool or language that supports COM. The interface enables you to launch RegexBuddy, and send and receive regex actions in response to the user's actions. It is a single instance interface, which means that each application has its own private instance of RegexBuddy.

Take a look at PowerGREP and EditPad Pro to see how convenient they make it to edit a regular expression with RegexBuddy.

Demo Applications Implementing RegexBuddy's COM Automation

At <http://download.jgsoft.com/buddy/RegexBuddy4Clients.zip> you can download two sample applications that communicate with RegexBuddy through its COM Automation interface. One is written in Delphi 2009 (Win32) and the other one is written in Visual Studio 2010 using C#. The examples show how to use `InitAction` and `FinishAction` to communicate with RegexBuddy and how to fall back on any older version of RegexBuddy if the user does not have RegexBuddy 4.

Importing RegexBuddy's Type Library

RegexBuddy's installer automatically registers RegexBuddy's automation interface with Windows. To automate RegexBuddy via COM, you need to import its type library. It is stored in `RegexBuddy4.exe`, which is installed under `C:\Program Files\Just Great Software\RegexBuddy 4\` by default.

In Delphi, select `Component | Import Component` from the menu. Select "Import a Type Library" and click Next. Select "RegexBuddy API" version "4.0" from `RegexBuddy4.exe` and click Next. Choose a palette page and a directory, make sure `Generate Component Wrappers` is ticked, and click Next. Install into a new or existing package as you prefer. Three new component called `TRegexBuddyIntf`, `TRegexBuddyIntf3`, and `TRegexBuddyIntf4` appear on the component palette. These components implement the methods and events you can use to communicate with RegexBuddy. Drop `TRegexBuddyIntf` on a form or data module. Set `ConnectKind` to `ckNewInstance`, and make sure `AutoConnect` is `False`. This ensures your application has its own instance of RegexBuddy, and that RegexBuddy only appears when the user actually wants to edit a regular expression. Call the component's `Connect()` method the first time the user wants to edit a regular expression with RegexBuddy. For efficiency, do not call the `Disconnect()` method until your application terminates (at which point it is called automatically). Assign an event handler either to `OnFinishRegex` or `OnFinishAction`, depending on which way of communication you prefer (see below). You can use `TRegexBuddyIntf3` or `TRegexBuddyIntf4` instead if you want to use the new methods in RegexBuddy 3 or 4. Calling `Connect()` on these components will raise an exception if the user has an older version of RegexBuddy. You can fall back to `TRegexBuddyIntf` when that happens. Once a call to `Connect()` succeeds, your application should not make any other calls to `Connect()` so that only one instance of the `TRegexBuddyIntf` component is live. Otherwise, it will launch multiple instances of RegexBuddy. Doing so would not cause any errors, but does waste resources.

In Visual Studio, right-click on “References” in the Solution Explorer, and pick “Add Reference”. Switch to the COM tab, and choose “RegexBuddy API” version “4.0” from `RegexBuddy4.exe`. After adding the reference, import the `RegexBuddy` namespace with `using RegexBuddy`. Then you can easily access the `RegexBuddyIntf4` class. Create a new object from this class the first time the user wants to edit a regular expression with `RegexBuddy`. Use the same object for all subsequent times the user wants to edit the same or another regex. Do not delete the object until your application terminates. Each time you create an object of `RegexBuddyIntf4`, a new `RegexBuddy` instance is launched. You also need to create an instance of either `IRegexBuddyIntfEvents_FinishRegexEventHandler` or `IRegexBuddyIntfEvents_FinishActionEventHandler` and assign it to the `FinishRegex` or `FinishAction` event of your `RegexBuddyIntf4` object. Do not assign both. Only assign the one you will actually use.

Note that to successfully communicate with `RegexBuddy`, two-way communication is required. Your application not only needs to call the methods of the `RegexBuddyIntf` interface to send the regular expression to `RegexBuddy`. It also needs to implement an event sink for either the `FinishRegex()` or `FinishAction()` method defined in the `IRegexBuddyIntfEvents` interface. Not all development and scripting tools that can call COM automation objects can also implement event sinks. The tool must support “early binding”.

RegexBuddyIntf Interface

The `RegexBuddyIntf` COM automation interface provides the following methods:

```
void IndicateApp(BSTR AppName, uint Wnd)
```

Call `IndicateApp` right after connecting to the COM interface. `AppName` will be displayed in `RegexBuddy`’s caption bar, to make it clear that this instance is connected to your application. `Wnd` is the handle of your application’s top-level form where the regex is used. `RegexBuddy` will call `SetForegroundWindow(Wnd)` to activate your application when the user is done with `RegexBuddy` (i.e. after a call to `FinishAction` or `FinishRegex`). You can call `IndicateApp` as often as you want, as long as you also call it right after connecting to the COM interface.

```
void InitAction(VARIANT Action)
```

Call `InitAction` to make `RegexBuddy` show up with the given `RegexBuddy` Action. `RegexBuddy` will follow up with a call to `FinishAction` when the user closes `RegexBuddy` by clicking the Send button.

```
BOOL CheckActionVersion(uint Version)
```

Returns `TRUE` if `RegexBuddy` supports this version of the `RegexBuddy` Action COM Variant Structure. `RegexBuddy` 1.x.x and 2.x.x support only version 1. `RegexBuddy` 3.0.0 and later support versions 1 and 3. `RegexBuddy` 4.0.0 and later support versions 1, 3, and 4. There is no version 2. These versions of the COM structure are supported regardless of whether you’re using `RegexBuddyIntf`, `RegexBuddyIntf3`, or `RegexBuddyIntf4`.

You should call this function with the highest version that your application supports. Newly developed applications should support versions 4, 3, and 1. If the function returns `FALSE`, call it again with a lower version, until you reach version 1. Version 1 is always supported. There’s no need to call `CheckActionVersion` for version 1. `RegexBuddy` will always use version 1 of the variant structure unless a call to `CheckActionVersion` returned `TRUE`.

If you will be calling `Convert()` with `ConvertBeforeSend` set to `TRUE` then it is sufficient to call `CheckActionVersion(4)` because the conversion ensures that `RegexBuddy` will send back a flavor that your application support. If you're not calling `Convert()` with `ConvertBeforeSend`, then calling only `CheckActionVersion(4)` allows `RegexBuddy` to send back an action with any application or regex flavor supported by `RegexBuddy`, including flavors added in future versions.

`RegexBuddy 4.1.0` and later allow you to tell `RegexBuddy` which application and regular expression flavor identifiers your application supports. Pass 410 for `Version` to request and indicate support for flavors supported by `RegexBuddy 4.1.0` (but not flavors added in post-4.1.0 releases) and to use version 4 of the `COM` structure. If `CheckActionVersion(410)` returns `FALSE`, call `CheckActionVersion(4)` to check whether the user has `RegexBuddy 4.0.x` (which by definition can't send back flavors added in 4.1.0 or later).

`RegexBuddy 4.1.0` also return `TRUE` if you call `CheckActionVersion(400)` to restrict the flavors to those available in `RegexBuddy 4.0.0`. But all `RegexBuddy 4.0.x` releases return `FALSE` when you call `CheckActionVersion(400)` even though they support all 4.0.0 flavors because the ability to check for flavor versions was added in 4.1.0.

```
uint GetWindowHandle()
```

Returns the window handle of `RegexBuddy`'s top window, which is either the main window, or the topmost modal dialog box in `RegexBuddy`. Pass this to `SetForegroundWindow()` after calling `InitRegex` or `InitAction`. `SetForegroundWindow()` only works when called by the thread that has input focus, so `RegexBuddy` cannot bring itself to front. That is your application's duty. Likewise, `RegexBuddy` will bring your application to front after calling `FinishAction`, using the window handle you passed on in `IndicateApp()`.

```
void InitRegex(BSTR Regex, uint StringType)
```

Deprecated. This method does not allow the regex flavor or matching modes to be passed, which have a significant impact on how the regular expression is interpreted. Use `InitAction` instead. The only reason to call `InitRegex` would be to have `RegexBuddy 1.x.x` or `2.x.x` interpret the regex using a string type, as version 1 of the `RegexBuddy Action Variant Structure` does not have an element to set the string style.

`InitRegex` makes `RegexBuddy` show up with the given regular expression. If you call `InitRegex`, then `RegexBuddy` will call `FinishRegex` instead of `FinishAction` when the user closes `RegexBuddy` by clicking the `Send` button. This means you'll need to assign event handler to `FinishRegex` instead of `FinishAction` if you want to use `InitRegex`. `Regex` is the regular expression that the user will edit in `RegexBuddy`. Can be an empty string. `StringType` tells `RegexBuddy` how to interpret the string you provided. Pass one of the integers listed in this table. By using a `StringType` other than zero, you can let `RegexBuddy` take care of adding and removing quote characters, escaping characters, etc. If you pass an unsupported string type, "as is" will be used instead.

StringType	Equivalent Identifier	String	Style Equivalent Copy/Paste menu item	Minimum Version	RegexBuddy
0	asis		as is	1.0.0 (2004)	
1	c		C string	1.0.0	
2	pascal		Delphi string	1.0.0	

3	perl	Perl string	1.0.0
4	perlop	Perl m// or s/// operator	1.0.0
5	basic	Basic string	1.0.0
6	javascript	JavaScript // operator	1.0.0
7	phppreg	PHP "/" string	1.0.0
8	phpereg	PHP string	3.0.0 (2007)
9	c#	C# string	3.0.0
10	python	Python string	3.0.0
11	ruby	Ruby // operator	3.0.0
12	sql	SQL string	3.0.0
13	tcl	Tcl word	3.0.0
14	xml	XML	3.0.0
15	java	Java string	3.1.0 (2007)
16	javascriptstring	JavaScript string	3.2.0 (2008)
17	powershell	PowerShell string	3.2.0
18	prism	Delphi Prism string	3.3.0 (2009)
19	groovy	Groovy string	3.3.0
20	scala	Scala string	3.4.0 (2009)
21	postgres	PostgreSQL string	4.0.0 (2013)
22	r	R string	4.0.0
23	cL	C wide string	4.2.0 (2014)
24	c11	C++11 raw string	4.5.0 (2015)
25	c11L	C++11 wide raw string	4.5.0

RegexBuddyIntf3 Interface

RegexBuddy 3.0.0 and later provide an additional interface called `RegexBuddyIntf3`. This interface descends from `RegexBuddyIntf`, so all the above methods are also supported by `RegexBuddyIntf3`. You can use this interface to make use of the new functionality in RegexBuddy 3. If your application is unable to instantiate this interface, it should fall back on `RegexBuddyIntf`. If you don't want to use the additional functionality, your application can simply use `RegexBuddyIntf` which works with all versions of RegexBuddy.

```
void SetTestString(BSTR Subject)
```

Call `SetTestString` to put (a sample of) the data your application will use the regular expression on into the Test panel in RegexBuddy. That way the user can instantly test the regex on the data it'll actually be used on. You should only use `SetTestString` if the data is not stored as a file on disk. Otherwise, `SetTestFile` is more efficient.

```
void SetTestFile(BSTR Filename)
```

Call `SetTestFile` to make `RegexBuddy` load one of the files your application will use the regular expression on into the Test panel. That way the user can instantly test the regex on the data it'll actually be used on.

```
void DebugAtPos(uint CharPos)
```

Show `RegexBuddy`'s debugger output for the match attempt starting at character position `CharPos` in the file. The first character in the file has position zero. Note that if matches are highlighted on the Test panel, and `CharPos` is in the middle of a highlighted match, the debugger will begin at the start of that match rather than at `CharPos`. The Debug button on the Test panel does the same.

```
void DebugAfterPos(uint CharPos)
```

Like `DebugAtPos`, but continues showing debugging output for all match attempts following the one at `CharPos`.

```
void DebugAll()
```

Show `RegexBuddy`'s debugger output for all match attempts in the whole test subject.

```
void AllowOptions(VARIANT Options)
```

Deprecated. `RegexBuddy 4` does not allow options to be disabled, because that creates problems with loading regexes from libraries or from the history. In `RegexBuddy 4`, calling `AllowOptions()` with the 5th element in the array (regular expression flavor) set to `FALSE` has the same effect as calling `Convert()` with the `ConvertBeforeSend` argument set to `TRUE` and the flavor set to the same flavor as passed via `InitAction()`. If both the 3rd and 5th element are `FALSE`, then the regex is also converted the same free-spacing setting as passed to `InitAction()`. Passing `TRUE` for the 3rd or 5th element has no effect other than to cancel the effects of a previous call with `FALSE` values. All other elements are always ignored.

`RegexBuddy 3` allows you to call `AllowOptions` to disable options that cannot be changed in your application and thus have to be always on or off. This method takes a variant that should be an array of 6 Boolean values.

Index Type Option

- 0 BOOL Dot matches all
- 1 BOOL Case insensitive
- 2 BOOL ^ and \$ match at line breaks
- 3 BOOL Free-spacing syntax
- 4 BOOL Split limit
- 5 BOOL Regular expression flavor

If you set any of these to `FALSE`, the user will be unable to change that option in `RegexBuddy 3`. Your application can still change the options via the `InitAction` method. In fact, it should do so, to make sure the options in `RegexBuddy` correspond with those actually used in your application.

RegexBuddyIntf4 Interface

RegexBuddy 4.0.0 and later provide yet an additional interface called `RegexBuddyIntf4`. This interface descends from `RegexBuddyIntf3`, so all the above methods, including the deprecated ones, are also supported by `RegexBuddyIntf4`. You can use this interface to make use of the new functionality in RegexBuddy 4. If your application is unable to instantiate this interface, it should fall back on `RegexBuddyIntf3` or `RegexBuddyIntf`. If you don't want to use the additional functionality, your application can simply use `RegexBuddyIntf3` or `RegexBuddyIntf`.

```
void Convert(VARIANT Options, BOOL Show, BOOL ConvertBeforeSend)
```

Set the drop-down lists on the Convert panel according to the `Options` argument. This argument should be an array with three elements.

Index	Type	Description
0	VARIANT	Target application. You can specify a built-in application as a string with one of the application identifiers. You can specify a custom application by passing a nested array as explained in the section “Custom Application COM Variant Structure”.
1	int	0 = preserve free-spacing or exact spacing; 1 = convert to free-spacing; 2 = convert to exact spacing
2	BOOL	Strip comments (TRUE) or keep comments (FALSE).

If you pass TRUE as the `Show` argument, then RegexBuddy immediately activates the Convert panel.

You should pass TRUE as the `ConvertBeforeSend` argument if your application can only work with one specific regex flavor. RegexBuddy then forces the regular expression to be converted to this flavor when the user clicks the Send button. If the conversion succeeds without warning, the converted regex is passed to your application via `FinishAction()`. If it doesn't, the Send button activates the Convert panel (regardless of the `Show` argument), changes the combo boxes to the values you passed the most recent call to `Convert()`, and informs the user that the conversion must be accepted before the regex can be sent back to your application.

The convert-before-send mode remains in effect until you call `Convert()` with `ConvertBeforeSend` set to FALSE. Doing so does change the drop-down lists on the Convert panel unless you pass NULL as the `Options` argument.

```
void SetTestStringEx(BSTR Subject, VARIANT Options)
```

Call `SetTestStringEx` to put (a sample of) the data your application will use the regular expression on into the Test panel in RegexBuddy. That way the user can instantly test the regex on the data it'll actually be used on. You should only use `SetTestStringEx` if the data is not stored as a file on disk. Otherwise, `SetTestFileEx` is more efficient. The `Meta` argument should be an array with 3 elements.

Index Type Description

0	BSTR	String style identifier that indicates how RegexBuddy should interpret the Subject string. Defaults to <code>as is</code> if you do not specify a valid string style identifier.
1	BSTR	If your application converted the string from a particular encoding to the Unicode BSTR

needed for the Subject argument, then you can specify one of the text file encoding identifiers to have RegexBuddy convert the string back to that encoding so the user can work with the same encoding on the Test panel in RegexBuddy as in your application. The default is `utf16le` which is the native encoding of the COM interface.

- 2 BSTR Specify `file`, `page`, or `line` to set the scope to “whole file”, “page by page”, or “line by line”. Pass `NULL` or an empty string to leave the scope unchanged.
- 3 BSTR You should set this to `mixed` to load the string without changing its line breaks. Other supported line break options are `auto`, `crlf`, `lf`, and `cr`.

```
void SetTestFileEx(BSTR Filename, VARIANT Options)
```

Call `SetTestFileEx` to make RegexBuddy load one of the files your application will use the regular expression on into the Test panel. That way the user can instantly test the regex on the data it'll actually be used on. The Meta argument should be an array with 3 elements.

Index Type Description

- 0 BSTR Specify one of the text file encoding identifiers to interpret the file as a text file with the given encoding. Specify `bytes` to load the file as a binary file in hexadecimal mode. Pass `NULL` or an empty string to have RegexBuddy auto-detect the encoding.
- 1 BSTR Specify `file`, `page`, or `line` to set the scope to “whole file”, “page by page”, or “line by line”. Pass `NULL` or an empty string to leave the scope unchanged.
- 2 BSTR Specify `auto` convert the line breaks in the file to the style expected by the regex flavor. Specify `mixed` to load the file without converting its line breaks. Specify `crlf`, `lf`, or `cr` to convert the line breaks to the specified style. The conversion is only applied to the copy of the file in memory. The conversion is not saved unless the user explicitly saves the file. Pass `NULL` or an empty string to leave the line break mode unchanged.

RegexBuddyIntfEvents Interface

```
FinishRegex(BSTR Regex, uint StringType)
```

You must provide an event handler for this event if you call `InitRegex`. After calling `InitRegex`, RegexBuddy will call `FinishRegex` when the user closes RegexBuddy by clicking the Send button. RegexBuddy will provide the final regular expression, using the same string type as you used in the last call to `InitRegex`.

```
FinishAction(VARIANT Action)
```

You must provide an event handler for this event if you call `InitAction`. After calling `InitAction`, RegexBuddy will call `FinishAction` when the user closes RegexBuddy by clicking the Send button. RegexBuddy will pass the same version of the variant structure to `FinishAction` as you passed to `InitAction`.

If the user closes RegexBuddy without clicking the Send button, your application will not receive a call to `FinishRegex` or `FinishAction`.

Which Methods to Call and Events to Handle

In summary, you must call `IndicateApp` immediately after connecting to `RegexBuddy`. Then, call either `InitRegex` or `InitAction` each time the user wants to edit a regular expression with `RegexBuddy`. `RegexBuddy` follows up a call to `InitRegex` or `InitAction` with a corresponding call to `FinishRegex` or `FinishAction`. If you call `InitRegex` or `InitAction` again before receiving a call to `FinishRegex` or `FinishAction`, the effects of the previous call to `InitRegex` or `InitAction` are canceled.

Note that in `RegexBuddy 3.0.0` and later, both `RegexBuddyIntf` and `RegexBuddyIntf3` support version 3 of the COM variant structure. You only need to instantiate `RegexBuddyIntf3` if you want to call the additional methods it offers.

RegexBuddy Action COM Variant Structure

The `InitAction` method and the `FinishAction` event handler of `RegexBuddy`'s COM interface pass the `RegexBuddy Action` as a Variant array. All elements must be present in the array, even those that are not applicable for the given kind of action or regular expression flavor. The values that do not apply are used as the defaults if the user changes the kind of action in `RegexBuddy`. `RegexBuddy` also fills elements that do not apply when calling `FinishAction`. You can ignore those. If the specified regular expression flavor does not allow a matching mode to be toggled, then it doesn't matter whether you specify `TRUE` or `FALSE` for that mode. `RegexBuddy` will only use modes that the flavor actually supports.

If you called `CheckActionVersion(1)` or if all your calls to `CheckActionVersion()` returned `FALSE`, then you should use version 1 of the variant structure.

Index Type Description

0	int	Version number. Must be set to 1.
1	int	Kind of action; 0 = match; 1 = replace; 2 = split.
2	BSTR	The regular expression, to be used "as is".
3	BSTR	The replacement text, to be used "as is".
4	BOOL	Dot matches all.
5	BOOL	Case insensitive.
6	BOOL	^ and \$ match at line breaks.
7	int	Split limit (if ≥ 2 , the split array contains at most <i>limit</i> items).
8	BSTR	Description of the action. Used as the default description if the user adds the action to a library.

There is no version 2 of the variant structure.

If you called `CheckActionVersion(3)` and `RegexBuddy` returned `TRUE`, then you should use version 3 of the variant structure:

Index Type Description

0	int	Version number. Must be set to 3.
1	int	Kind of action; 0 = match; 1 = replace; 2 = split.

- 2 BSTR The regular expression. Can be formatted as a string as indicated by element 11.
- 3 BSTR The replacement text. Can be formatted as a string as indicated by element 11.
- 4 BOOL Dot matches all.
- 5 BOOL Case insensitive.
- 6 BOOL `^` and `$` match at line breaks.
- 7 BOOL Free-spacing syntax.
- 8 int Split limit (if ≥ 2 , the split array contains at most *limit* items).
Regular expression flavor. One of: `igsoft`, `dotnet`, `java`, `perl`, `pcre`, `javascript`, `python`, `ruby`, `tcl`, `bre`, `ere`, `gnubre`, `gnuere`, `xml`, `xpath`. Defaults to `igsoft` if you specify an invalid flavor. These identifiers are a subset of the regex flavor identifiers supported by RegxBuddy 4. They are the complete list of flavors supported by RegxBuddy 3. If the user selected a different flavor in RegxBuddy 4, then it will specify a RegxBuddy 3 flavor that is “close enough”, as the flavor definitions in RegxBuddy 3 were not nearly as exact as those in RegxBuddy 4.
- 9 BSTR Replacement text flavor. One of: `igsoft`, `dotnet`, `java`, `perl`, `javascript`, `python`, `ruby`, `tcl`, `phpereg`, `phppreg`, `realbasic`, `oracle`, `xpath`. RegxBuddy 3 will use the default replacement flavor for the chosen regex flavor if the replacement text flavor you specified is incompatible with the regular expression flavor. RegxBuddy 4 allows any combination of regex and replacement flavor. These identifiers are a subset of the replacement flavor identifiers supported by RegxBuddy 4. They are the complete list of flavors supported by RegxBuddy 3.
- 10 BSTR StringType. One of the values for the StringType argument of the InitRegex function. If this value is nonzero, RegxBuddy will interpret elements 2 and 3 in this array using this string type. RegxBuddy will use the same value in FinishAction as you used in InitAction and format the regex and replacement accordingly.
- 11 int Description of the action. Used as the default description if the user adds the action to a library.
- 12 BSTR

If you called `CheckActionVersion(4)` and RegxBuddy returned `TRUE`, then you should use version 4 of the variant structure:

Index	Type	Description
0	int	Version number. Must be set to 4.
1	int	Action purpose: 0 = match; 1 = replace; 2 = split.
2	BSTR	The regular expression. Can be formatted as a string as indicated by element 5.
3	BSTR	The replacement text. Can be formatted as a string as indicated by element 5.
4	VARIANT	Application. You can specify a built-in application as a string with one of the application identifiers. You can specify a custom application by passing a nested array as explained in the next section.
5	BSTR	String style identifier that indicates how RegxBuddy should interpret elements 2 and 3 in this array. RegxBuddy will use the same value in FinishAction as you used in InitAction and format the regex and replacement accordingly. Defaults to <code>as is</code> if you do not specify a valid string style identifier. This string style is used only for the COM variant structure and is independent of the string style normally used by the application as specified in element 4.
6	BSTR	Description of the action. Used as the default description if the user adds the action to a library.

7	BOOL	Strict emulation (TRUE) or Helpful emulation (FALSE).
8	BOOL	Case insensitive (TRUE) or case sensitive (FALSE).
9	BOOL	Free-spacing (TRUE) or exact spacing (FALSE).
10	BOOL	Dot matches line breaks (TRUE) or dot doesn't match line breaks (FALSE).
11	BOOL	^ and \$ match at line breaks (TRUE) or ^ and \$ don't match at line breaks (FALSE).
12	BSTR	Line break style. One of: default, lf, cr, crlfonly, crlf, unicode
13	BOOL	Named capture only (TRUE) or numbered capture (FALSE).
14	BOOL	Allow duplicate names (TRUE) or require names to be unique (FALSE).
15	BOOL	Lazy quantifiers (TRUE) or greedy quantifiers (FALSE).
16	BOOL	Skip zero-length matches (TRUE) or allow zero-length matches (FALSE).
17	BOOL	Support string syntax (TRUE) or regex syntax only (FALSE).
18	int NULL	or Limit for "split" actions. Set to a NULL variant to indicate that no limit is set at all (which is not the same as setting the limit to zero).
19	BOOL	Split: add groups (TRUE) or don't add groups (FALSE).
20	BOOL	Split: add empty strings (TRUE) or don't add empty strings (FALSE).

Custom Application COM Variant Structure

Version 4 of the RegexBuddy Action COM variant structure allows you to specify a custom application in element 4. To do so, pass an array with these elements:

Index	Type	Description
0	BSTR	Name the flavor is indicated with in flavor selection lists. This is only used if the custom application is unknown to RegexBuddy. If the user previously added the same application under a different name, then the user's chosen name will be preserved.
1	BSTR	Regular expression flavor identifier
2	BSTR or NULL	Replacement flavor identifier. You can set this to the empty string or NULL to disable Replace mode if the application cannot search-and-replace using a regex.
3	BSTR or NULL	Split flavor identifier. You can set this to the empty string or NULL to disable Split mode if the application cannot split strings using a regex.
4	BSTR or NULL	String style identifier. Used only for the Copy and Paste menus in RegexBuddy. Not used to pass regexes via the COM interface. Defaults to "asis" if omitted.
5	BSTR or NULL	File name without extension of the RegexBuddy source code template for generating source code snippets on the Use panel. This can be a built-in or a custom template. Use panel is disabled if you don't specify a template file name or if the file cannot be found.

Working with a List of Actions

RegexBuddy 3.0.0 and later can work with a list of regular expressions via `InitAction()` and `FinishAction()`. If you invoke RegexBuddy this way, your list of regular expressions will replace the History feature in RegexBuddy.

RegexBuddy will also pass all regular expressions in the History list to `FinishAction()`, using the Variant array as described in this section. If the History list holds only one regular expression, RegexBuddy will still pass an array `[0x103, 1, [action]]`. RegexBuddy will always return at least one regular expression, since the user cannot delete the last regular expression in the History list. It can be blank though.

If you called `CheckActionVersion(3)` and RegexBuddy returned `TRUE`, then you should pass the list of actions this way:

Index	Type	Description
0	int	Version number. Must be set to 0x103.
1	int	Number of regular expressions in the list. If you set this to zero, RegexBuddy starts with one blank regex in the History list.
2..n+1	VARIANT	As many version 3 variant structures (described above) as specified in element 1.

If you called `CheckActionVersion(4)` and RegexBuddy returned `TRUE`, then you should pass the list of actions this way:

Index	Type	Description
0	int	Version number. Must be set to 0x104.
1	int	Number of regular expressions in the list. If you set this to zero, RegexBuddy starts with one blank regex in the History list.
2	int	Selected regular expression in the list. RegexBuddy starts with this action selected in the History list.
3..n+2	VARIANT	As many version 4 variant structures (described above) as specified in element 1.

54. Application Identifiers

Identifiers for the `-app` command line parameter and for the RegexBuddy action variant structure in the COM interface. Each application defines a regular expression flavor and a string style. Most applications also define a replacement text flavor, a split flavor, and a source code template.

Identifier	Age	Application
act2	4.0.0	AceText 2 & 3
ace4	4.14.0	AceText 4
aspnet11	4.1.0	ASP.NET 1.1
aspnet20	4.1.0	ASP.NET 2.0–4.8
boostreansi1038	4.7.0	boost::regex 1.38–1.39
boostreansi1042	4.7.0	boost::regex 1.42–1.43
boostreansi1044	4.7.0	boost::regex 1.44–1.46
boostreansi1047	4.7.0	boost::regex 1.47
boostreansi1048	4.13.0	boost::regex 1.48–1.49
boostreansi1050	4.13.0	boost::regex 1.50–1.53
boostreansi1054	4.7.0	boost::regex 1.54–1.57
boostreansi1058	4.7.0	boost::regex 1.58–1.59
boostreansi1060	4.7.0	boost::regex 1.60–1.61
boostreansi1062	4.7.0	boost::regex 1.62–1.63
boostreansi1064	4.8.0	boost::regex 1.64–1.65
boostreansi1066	4.9.0	boost::regex 1.66–1.77
boostreansi1078	4.14.0	boost::regex 1.78–1.83
boostwide1038	4.7.0	boost::wregex 1.38–1.39
boostwide1042	4.7.0	boost::wregex 1.42–1.43
boostwide1044	4.7.0	boost::wregex 1.44–1.46
boostwide1047	4.7.0	boost::wregex 1.47–1.53
boostwide1054	4.7.0	boost::wregex 1.54–1.57
boostwide1058	4.7.0	boost::wregex 1.58–1.59
boostwide1060	4.7.0	boost::wregex 1.60–1.63
boostwide1064	4.8.0	boost::wregex 1.64–1.65
boostwide1066	4.9.0	boost::wregex 1.66–1.77
boostwide1078	4.14.0	boost::wregex 1.78–1.83
csharp11	4.0.0	C# (.NET 1.1)
csharp20	4.0.0	C# (.NET 2.0–7.0)
cppb102core	4.8.0	C++Builder 10.2 (TPerlRegEx)
cppb102	4.8.0	C++Builder 10.2 (TRegEx)
cppb103core	4.10.0	C++Builder 10.3–11 (TPerlRegEx)
cppb103	4.10.0	C++Builder 10.3–11 (TRegEx)
cppbxecore	4.0.0	C++Builder XE (TPerlRegEx)

cppbxe	4.0.0	C++Builder XE (TRegExp)
cppbxe2core	4.0.0	C++Builder XE2–XE3 (TPerlRegExp)
cppbxe2	4.0.0	C++Builder XE2–XE3 (TRegExp)
cppbxe4core	4.0.0	C++Builder XE4–XE5 (TPerlRegExp)
cppbxe4	4.0.0	C++Builder XE4–XE5 (TRegExp)
cppbxe6core	4.1.0	C++Builder XE6 (TPerlRegExp)
cppbxe6	4.1.0	C++Builder XE6 (TRegExp)
cppbxe7core	4.2.0	C++Builder XE7–XE8 & 10–10.1 (TPerlRegExp)
cppbxe7	4.2.0	C++Builder XE7–XE8 & 10–10.1 (TRegExp)
delphi102core	4.8.0	Delphi 10.2 (TPerlRegExp)
delphi102	4.8.0	Delphi 10.2 (TRegExp)
delphi103core	4.10.0	Delphi 10.3–11 (TPerlRegExp)
delphi103	4.10.0	Delphi 10.3–11 (TRegExp)
delphi2007	4.0.0	Delphi 2007 and prior (TPerlRegExp)
delphi2009	4.0.0	Delphi 2009–2010 (TPerlRegExp)
delphinet	4.0.0	Delphi for .NET
prism11	4.0.0	Delphi Prism (.NET 1.1)
prism20	4.0.0	Delphi Prism (.NET 2.0–4.8)
delphixecore	4.0.0	Delphi XE (TPerlRegExp)
delphixe	4.0.0	Delphi XE (TRegExp)
delphixe2core	4.0.0	Delphi XE2–XE3 (TPerlRegExp)
delphixe2	4.0.0	Delphi XE2–XE3 (TRegExp)
delphixe4core	4.0.0	Delphi XE4–XE5 (TPerlRegExp)
delphixe4	4.0.0	Delphi XE4–XE5 (TRegExp)
delphixe6core	4.1.0	Delphi XE6 (TPerlRegExp)
delphixe6	4.1.0	Delphi XE6 (TRegExp)
delphixe7core	4.2.0	Delphi XE7–XE8 & 10–10.1 (TPerlRegExp)
delphixe7	4.2.0	Delphi XE7–XE8 & 10–10.1 (TRegExp)
epp5	4.0.0	EditPad 5
epp6	4.0.0	EditPad 6 & 7
epp8	4.10.0	EditPad 8
gnubre	4.0.0	GNU BRE
gnuere	4.0.0	GNU ERE
groovy4	4.0.0	Groovy (JDK 1.4)
groovy5	4.0.0	Groovy (JDK 1.5)
groovy6	4.0.0	Groovy (JDK 1.6)
groovy7	4.0.0	Groovy (JDK 1.7)
groovy8	4.1.0	Groovy (JDK 1.8)
groovy9	4.9.0	Groovy (JDK 9–12)
groovy13	4.10.0	Groovy (JDK 13–14)
groovy15	4.12.0	Groovy (JDK 15)

groovy16	4.12.0	Groovy (JDK 16)
groovy17	4.13.0	Groovy (JDK 17–18)
groovy19	4.14.0	Groovy (JDK 19–21)
html5chrome	4.1.0	HTML5 Pattern (Chrome)
html5edge	4.5.0	HTML5 Pattern (Edge)
html5firefox	4.1.0	HTML5 Pattern (Firefox)
html5msie	4.1.0	HTML5 Pattern (MSIE standard)
html5opera	4.1.0	HTML5 Pattern (Opera)
java4	4.0.0	Java 4
java5	4.0.0	Java 5
java6	4.0.0	Java 6
java7	4.0.0	Java 7
java8	4.1.0	Java 8
java9	4.9.0	Java 9–12
java13	4.10.0	Java 13–14
java15	4.12.0	Java 15
java16	4.12.0	Java 16
java17	4.13.0	Java 17–18
java19	4.14.0	Java 19–21
chrome	4.0.0	JavaScript (Chrome)
edge	4.5.0	JavaScript (Edge)
firefox	4.0.0	JavaScript (Firefox)
quirks	4.0.0	JavaScript (MSIE quirks)
msie	4.0.0	JavaScript (MSIE standard)
opera	4.0.0	JavaScript (Opera)
mysql	4.0.0	MySQL
oracle10gR1	4.0.0	Oracle 10gR1
oracle10gR2	4.0.0	Oracle 10gR2
oracle11g	4.0.0	Oracle 11gR1, 11gR2 & 12c
pcre40	4.0.0	PCRE 4.0–4.4
pcre45	4.8.0	PCRE 4.5
pcre50	4.0.0	PCRE 5.0–6.4
pcre65	4.0.0	PCRE 6.5–6.6
pcre67	4.0.0	PCRE 6.7
pcre70	4.0.0	PCRE 7.0
pcre71	4.0.0	PCRE 7.1
pcre72	4.0.0	PCRE 7.2–7.3
pcre74	4.0.0	PCRE 7.4
pcre75	4.0.0	PCRE 7.5–7.6
pcre77	4.0.0	PCRE 7.7–7.9
pcre800	4.0.0	PCRE 8.00

pcr801	4.0.0	PCRE 8.01–8.02
pcr810	4.0.0	PCRE 8.10
pcr811	4.0.0	PCRE 8.11–8.12
pcr813	4.0.0	PCRE 8.13
pcr820	4.0.0	PCRE 8.20
pcr821	4.0.0	PCRE 8.21
pcr830	4.0.0	PCRE 8.30–8.33 UTF-8
pcr830_16	4.0.0	PCRE 8.30–8.33 UTF-16
pcr832_32	4.0.0	PCRE 8.32–8.33 UTF-32
pcr834	4.1.0	PCRE 8.34–8.35 UTF-8
pcr834_16	4.1.0	PCRE 8.34–8.35 UTF-16
pcr834_32	4.1.0	PCRE 8.34–8.35 UTF-32
pcr836	4.3.0	PCRE 8.36–8.38 UTF-8
pcr836_16	4.3.0	PCRE 8.36–8.38 UTF-16
pcr836_32	4.3.0	PCRE 8.36–8.38 UTF-32
pcr839	4.8.0	PCRE 8.39 UTF-8
pcr839_16	4.8.0	PCRE 8.39 UTF-16
pcr839_32	4.8.0	PCRE 8.39 UTF-32
pcr840	4.8.0	PCRE 8.40–8.45 UTF-8
pcr840_16	4.8.0	PCRE 8.40–8.45 UTF-16
pcr840_32	4.8.0	PCRE 8.40–8.45 UTF-32
pcr1010	4.3.0	PCRE2 10.10
pcr1020	4.5.0	PCRE2 10.20
pcr1021	4.6.0	PCRE2 10.21–10.22
pcr1023	4.8.0	PCRE2 10.23
pcr1030	4.9.0	PCRE2 10.30–10.31
pcr1032	4.10.0	PCRE2 10.32–10.34
pcr1035	4.12.0	PCRE2 10.35–10.39
perl508	4.0.0	Perl 5.8
perl510	4.0.0	Perl 5.10
perl512	4.0.0	Perl 5.12
perl514	4.0.0	Perl 5.14–5.16
perl518	4.0.0	Perl 5.18
perl520	4.2.0	Perl 5.20
perl522	4.4.0	Perl 5.22
perl524	4.8.0	Perl 5.24
perl526	4.8.0	Perl 5.26–5.28
perl530	4.10.0	Perl 5.30–5.32
phpereg	4.0.0	PHP ereg 4.3.3–5.2.17
phppreg4303	4.0.0	PHP preg 4.3.3–4.3.4
phppreg4305	4.8.0	PHP preg 4.3.5–4.3.11

phppreg4402	4.0.0	PHP preg 4.4.0–4.4.2
phppreg4403	4.0.0	PHP preg 4.4.3–4.4.4
phppreg4405	4.0.0	PHP preg 4.4.5
phppreg4406	4.0.0	PHP preg 4.4.6–4.4.8
phppreg5000	4.0.0	PHP preg 5.0.0–5.0.4
phppreg5005	4.0.0	PHP preg 5.0.5–5.1.2
phppreg5103	4.0.0	PHP preg 5.1.3–5.1.6
phppreg5200	4.0.0	PHP preg 5.2.0–5.2.1
phppreg5202	4.0.0	PHP preg 5.2.2–5.2.3
phppreg5204	4.0.0	PHP preg 5.2.4
phppreg5205	4.0.0	PHP preg 5.2.5
phppreg5206	4.0.0	PHP preg 5.2.6
phppreg5207	4.0.0	PHP preg 5.2.7–5.2.13
phppreg5214	4.0.0	PHP preg 5.2.14–5.2.17
phppreg5300	4.0.0	PHP preg 5.3.0–5.3.1
phppreg5302	4.0.0	PHP preg 5.3.2
phppreg5303	4.0.0	PHP preg 5.3.3
phppreg5304	4.0.0	PHP preg 5.3.4–5.3.5
phppreg5306	4.0.0	PHP preg 5.3.6–5.3.18
phppreg5319	4.0.0	PHP preg 5.3.19–5.3.29
phppreg5400	4.0.0	PHP preg 5.4.0–5.4.8
phppreg5409	4.0.0	PHP preg 5.4.9–5.4.40
phppreg5441	4.4.0	PHP preg 5.4.41–5.4.45
phppreg5500	4.1.0	PHP preg 5.5.0–5.5.9
phppreg5510	4.1.0	PHP preg 5.5.10–5.5.24
phppreg5525	4.4.0	PHP preg 5.5.25–5.5.26
phppreg5527	4.5.0	PHP preg 5.5.27–5.5.31
phppreg5532	4.6.0	PHP preg 5.5.32–5.5.38
phppreg5600	4.2.0	PHP preg 5.6.0–5.6.8
phppreg5609	4.4.0	PHP preg 5.6.9–5.6.10
phppreg5611	4.5.0	PHP preg 5.6.11–5.6.17
phppreg5618	4.6.0	PHP preg 5.6.18–5.6.40
phppreg7000	4.6.0	PHP preg 7.0.0–7.0.2
phppreg7003	4.6.0	PHP preg 7.0.3
phppreg7004	4.6.0	PHP preg 7.0.4–7.1.33
phppreg7200	4.9.0	PHP preg 7.2.0–7.2.34
phppreg7300	4.10.0	PHP preg 7.3.0–7.3.33
phppreg7400	4.11.0	PHP preg 7.4.0–7.4.11
phppreg7412	4.12.0	PHP preg 7.4.12–7.4.33
phppreg8000	4.12.0	PHP preg 8.0.0–8.1.24
bre	4.0.0	POSIX BRE

ere	4.0.0	POSIX ERE
postgresql	4.0.0	PostgreSQL
pgr2	4.0.0	PowerGREP 2
pgr4	4.0.0	PowerGREP 3 & 4
pgr5	4.6.0	PowerGREP 5
powershell	4.0.0	PowerShell
powershellops	4.5.0	PowerShell operators
python24	4.0.0	Python 2.4–2.6
python27	4.0.0	Python 2.7
python30	4.0.0	Python 3.0
python31	4.0.0	Python 3.1
python32	4.0.0	Python 3.2
python33	4.0.0	Python 3.3
python34	4.1.0	Python 3.4
python35	4.5.0	Python 3.5
python36	4.8.0	Python 3.6
python37	4.9.0	Python 3.7–3.10
python311	4.14.0	Python 3.11–3.12
r2140	4.0.0	R 2.14.0–2.14.1
r2142	4.0.0	R 2.14.2
r2150	4.0.0	R 2.15.0–3.0.2
r3003	4.1.0	R 3.0.3–3.1.2
r3013	4.3.0	R 3.1.3–3.4.4
r3050	4.9.0	R 3.5.0–3.6.3
r4000	4.11.0	R 4.0.0–4.1.3
r4020	4.14.0	R 4.2.0–4.2.1
ruby18	4.0.0	Ruby 1.8
ruby19	4.0.0	Ruby 1.9
ruby20	4.0.0	Ruby 2.0–2.1
ruby22	4.3.0	Ruby 2.2–2.3
ruby24	4.8.0	Ruby 2.4–3.2
scala4	4.0.0	Scala (JDK 1.4)
scala5	4.0.0	Scala (JDK 1.5)
scala6	4.0.0	Scala (JDK 1.6)
scala7	4.0.0	Scala (JDK 1.7)
scala8	4.1.0	Scala (JDK 1.8)
scala9	4.9.0	Scala (JDK 9–12)
scala13	4.10.0	Scala (JDK 13–14)
scala15	4.12.0	Scala (JDK 15)
scala16	4.12.0	Scala (JDK 16)
scala17	4.13.0	Scala (JDK 17–18)

scala19	4.14.0	Scala (JDK 19–21)
stdreansicx10	4.5.0	std::regex (C++Builder 10–10.4)
stdreansicb11	4.13.0	std::regex (C++Builder 11)
stdreansicbxe3	4.2.0	std::regex (C++Builder Win64 XE3–XE6)
stdreansicbxe7	4.2.0	std::regex (C++Builder Win64 XE7–XE8)
stdreansivc2008	4.2.0	std::regex (Visual C++ 2008)
stdreansivc2010	4.2.0	std::regex (Visual C++ 2010)
stdreansivc2012	4.2.0	std::regex (Visual C++ 2012)
stdreansivc2013	4.2.0	std::regex (Visual C++ 2013)
stdreansivc2015	4.5.0	std::regex (Visual C++ 2015)
stdreansivc2017	4.8.0	std::regex (Visual C++ 2017–2022)
stdrewidecx10	4.5.0	std:wregex (C++Builder 10–10.4)
stdrewidecb11	4.13.0	std:wregex (C++Builder 11)
stdrewidecbxe3	4.2.0	std:wregex (C++Builder Win64 XE3–XE6)
stdrewidecbxe7	4.2.0	std:wregex (C++Builder Win64 XE7–XE8)
stdrewidevc2008	4.2.0	std:wregex (Visual C++ 2008)
stdrewidevc2010	4.2.0	std:wregex (Visual C++ 2010)
stdrewidevc2012	4.2.0	std:wregex (Visual C++ 2012)
stdrewidevc2013	4.5.0	std:wregex (Visual C++ 2013)
stdrewidevc2015	4.5.0	std:wregex (Visual C++ 2015)
stdrewidevc2017	4.8.0	std:wregex (Visual C++ 2017–2022)
tcl84	4.0.0	Tcl 8.4
tcl85	4.3.0	Tcl 8.5
tcl86	4.0.0	Tcl 8.6
vbscript	4.0.0	VBScript
vbnet11	4.0.0	Visual Basic (.NET 1.1)
vbnet20	4.0.0	Visual Basic (.NET 2.0–7.0)
vb6	4.0.0	Visual Basic 6
vs2012	4.1.0	Visual Studio 2012–2022 IDE
wxWidgets	4.0.0	wxWidgets
xml	4.0.0	XML Schema
xpath	4.0.0	XPath
xregexp2chrome	4.0.0	XRegExp 2 (Chrome)
xregexp2edge	4.5.0	XRegExp 2 (Edge)
xregexp2firefox	4.0.0	XRegExp 2 (Firefox)
xregexp2quirks	4.0.0	XRegExp 2 (MSIE quirks)
xregexp2msie	4.0.0	XRegExp 2 (MSIE standard)
xregexp2opera	4.0.0	XRegExp 2 (Opera)
xregexp3chrome	4.5.0	XRegExp 3 (Chrome)
xregexp3edge	4.5.0	XRegExp 3 (Edge)
xregexp3firefox	4.5.0	XRegExp 3 (Firefox)

xregexp3quirks	4.5.0	XRegExp 3 (MSIE quirks)
xregexp3msie	4.5.0	XRegExp 3 (MSIE standard)
xregexp3opera	4.5.0	XRegExp 3 (Opera)
xregexp4chrome	4.9.0	XRegExp 4 (Chrome)
xregexp4edge	4.9.0	XRegExp 4 (Edge)
xregexp4firefox	4.9.0	XRegExp 4 (Firefox)
xregexp4msie	4.9.0	XRegExp 4 (MSIE standard)
xregexp4opera	4.9.0	XRegExp 4 (Opera)
xregexp5chrome	4.12.0	XRegExp 5 (Chrome)
xregexp5edge	4.12.0	XRegExp 5 (Edge)
xregexp5firefox	4.12.0	XRegExp 5 (Firefox)
xregexp5msie	4.12.0	XRegExp 5 (MSIE standard)

Regex Flavor Identifiers

Identifiers for the `-flavor` command line parameter and for the custom application variant structure in the COM interface.

Identifier	Age	Regex Flavor
jgsoft5	4.0.0	JGsoft V2 (EditPad 8; PowerGREP 5)
jgsoft	3.0.0	JGsoft (PowerGREP 3 & 4)
jgsoft3	4.0.0	JGsoft (EditPad 6 & 7)
jgsoftpcr	4.0.0	JGsoft PCRE (EditPad 5; PowerGREP 2)
dotnet11	4.0.0	.NET 1.1
dotnet	3.0.0	.NET 2.0–7.0
vs2012	4.1.0	Visual Studio 2012–2022 IDE
dotnetecma11	4.5.0	.NET 1.1 (ECMAScript)
dotnetecma20	4.5.0	.NET 2.0–7.0 (ECMAScript)
java4	4.0.0	Java 4
java	3.0.0	Java 5
java6	4.0.0	Java 6
java7	4.0.0	Java 7
java8	4.1.0	Java 8
java9	4.9.0	Java 9–12
java13	4.10.0	Java 13–14
java15	4.12.0	Java 15
java16	4.12.0	Java 16
java17	4.13.0	Java 17–18
java19	4.14.0	Java 19–21
perl	3.0.0	Perl 5.8
perl510	4.0.0	Perl 5.10

perl512	4.0.0	Perl 5.12
perl514	4.0.0	Perl 5.14–5.16
perl518	4.0.0	Perl 5.18
perl520	4.2.0	Perl 5.20
perl522	4.4.0	Perl 5.22
perl524	4.8.0	Perl 5.24
perl526	4.8.0	Perl 5.26–5.28
perl530	4.10.0	Perl 5.30–5.32
pcre40	4.0.0	PCRE 4.0–4.4
pcre45	4.8.0	PCRE 4.5
pcre50	4.0.0	PCRE 5.0–6.4
pcre65	4.0.0	PCRE 6.5–6.6
pcre	3.0.0	PCRE 6.7
pcre70	4.0.0	PCRE 7.0
pcre71	4.0.0	PCRE 7.1
pcre72	4.0.0	PCRE 7.2–7.3
pcre74	4.0.0	PCRE 7.4
pcre75	4.0.0	PCRE 7.5–7.6
pcre77	4.0.0	PCRE 7.7–7.9
pcre800	4.0.0	PCRE 8.00
pcre801	4.0.0	PCRE 8.01–8.02
pcre810	4.0.0	PCRE 8.10
pcre811	4.0.0	PCRE 8.11–8.12
pcre813	4.0.0	PCRE 8.13
pcre820	4.0.0	PCRE 8.20
pcre821	4.0.0	PCRE 8.21
pcre830	4.0.0	PCRE 8.30–8.33
pcre834	4.1.0	PCRE 8.34–8.35
pcre836	4.3.0	PCRE 8.36–8.38
pcre839	4.8.0	PCRE 8.39
pcre840	4.8.0	PCRE 8.40–8.45
PCRE1010	4.3.0	PCRE2 10.10
pcre1020	4.5.0	PCRE2 10.20
pcre1021	4.6.0	PCRE2 10.21–10.22
pcre1023	4.8.0	PCRE2 10.23
pcre1030	4.9.0	PCRE2 10.30–10.31
pcre1032	4.10.0	PCRE2 10.32–10.34
pcre1035	4.12.0	PCRE2 10.35–10.39
delphixe	4.0.0	Delphi XE (TRegEx)
delphixe2	4.0.0	Delphi XE2–XE5 (TRegEx)
delphixe6	4.1.0	Delphi XE6 (TRegEx)

delphixe7	4.2.0	Delphi XE7–XE8 & 10–10.1 (TRegEx)
delphi102	4.8.0	Delphi 10.2 (TRegEx)
delphi103	4.10.0	Delphi 10.3–11 (TRegEx)
delphi2007	4.0.0	Delphi 2007 (TPerlRegEx)
delphixecore	4.0.0	Delphi XE (TPerlRegEx)
delphixe2core	4.0.0	Delphi XE2–XE3, 2009 (TPerlRegEx)
delphixe4core	4.0.0	Delphi XE4–XE5 (TPerlRegEx)
delphixe6core	4.1.0	Delphi XE6 (TPerlRegEx)
delphixe7core	4.2.0	Delphi XE7–XE8 & 10–10.1 (TPerlRegEx)
delphi102core	4.8.0	Delphi 10.2 (TPerlRegEx)
delphi103core	4.10.0	Delphi 10.3–11 (TPerlRegEx)
phppreg4303	4.0.0	PHP preg 4.3.3–4.3.4
phppreg4305	4.8.0	PHP preg 4.3.5–4.3.11, 5.0.0–5.0.4
phppreg4402	4.0.0	PHP preg 4.4.0–4.4.2, 5.0.5–5.1.2
phppreg4403	4.0.0	PHP preg 4.4.3–4.4.4
phppreg4405	4.0.0	PHP preg 4.4.5
phppreg4406	4.0.0	PHP preg 4.4.6–4.4.8
phppreg5103	4.0.0	PHP preg 5.1.3–5.1.6
phppreg5200	4.0.0	PHP preg 5.2.0–5.2.1
phppreg5202	4.0.0	PHP preg 5.2.2–5.2.3
phppreg5204	4.0.0	PHP preg 5.2.4
phppreg5205	4.0.0	PHP preg 5.2.5
phppreg5206	4.0.0	PHP preg 5.2.6
phppreg5207	4.0.0	PHP preg 5.2.7–5.2.13, 5.3.0–5.3.1
phppreg5214	4.0.0	PHP preg 5.2.14–5.2.17, 5.3.3
phppreg5303	4.0.0	PHP preg 5.3.2
phppreg5304	4.0.0	PHP preg 5.3.4–5.3.5
phppreg5306	4.0.0	PHP preg 5.3.6–5.3.18, 5.4.0–5.4.8
phppreg5319	4.0.0	PHP preg 5.3.19–5.3.29, 5.4.9–5.4.40, 5.5.0–5.5.9
phppreg5510	4.1.0	PHP preg 5.5.10–5.5.24, 5.6.0–5.6.8
phppreg5609	4.4.0	PHP preg 5.4.41–5.4.45, 5.5.25–5.5.26, 5.6.9–5.6.10
phppreg5611	4.5.0	PHP preg 5.5.27–5.5.31, 5.6.11–5.6.17
phppreg5618	4.6.0	PHP preg 5.5.32–5.5.38, 5.6.18–5.6.40
phppreg7000	4.6.0	PHP preg 7.0.0–7.0.2
phppreg7003	4.6.0	PHP preg 7.0.3–7.1.33
phppreg7200	4.9.0	PHP preg 7.2.0–7.2.34
phppreg7300	4.10.0	PHP preg 7.3.0–7.3.33
phppreg7400	4.11.0	PHP preg 7.4.0–7.4.11
phppreg7412	4.12.0	PHP preg 7.4.12–7.4.33
phppreg8000	4.12.0	PHP preg 8.0.0–8.1.24
r2140	4.0.0	R 2.14.0–2.14.1

r2142	4.0.0	R 2.14.2
r2150	4.0.0	R 2.15.0–3.0.2
r3003	4.1.0	R 3.0.3–3.1.2
r3013	4.3.0	R 3.1.3–3.4.4
r3050	4.9.0	R 3.5.0–3.6.3
r4000	4.11.0	R 4.0.0–4.1.3
r4020	4.14.0	R 4.2.0–4.2.1
chrome	4.0.0	JavaScript (Chrome & Firefox)
javascript	3.0.0	JavaScript (MSIE standard)
quirks	4.0.0	JavaScript (MSIE quirks)
html5chrome	4.1.0	HTML5 Pattern (Chrome & Firefox)
html5	4.1.0	HTML5 Pattern (MSIE standard)
xregexp2firefox	4.0.0	XRegExp 2 (Chrome & Firefox)
xregexp2	4.0.0	XRegExp 2 (MSIE standard)
xregexp2quirks	4.0.0	XRegExp 2 (MSIE quirks)
xregexp3firefox	4.5.0	XRegExp 3 (Chrome & Firefox)
xregexp3	4.5.0	XRegExp 3 (MSIE standard)
xregexp3quirks	4.5.0	XRegExp 3 (MSIE quirks)
xregexp4firefox	4.9.0	XRegExp 4 (Chrome & Firefox)
xregexp4msie	4.9.0	XRegExp 4 (MSIE standard)
xregexp5firefox	4.12.0	XRegExp 5 (Chrome & Firefox)
xregexp5msie	4.12.0	XRegExp 5 (MSIE standard)
python	3.0.0	Python 2.4–2.6
python27	4.0.0	Python 2.7
python30	4.0.0	Python 3.0–3.1
python32	4.0.0	Python 3.2
python33	4.0.0	Python 3.3–3.4
python35	4.5.0	Python 3.5
python36	4.8.0	Python 3.6
python37	4.9.0	Python 3.7–3.10
python311	4.14.0	Python 3.11–3.12
ruby	3.0.0	Ruby 1.8
ruby19	4.0.0	Ruby 1.9
ruby20	4.0.0	Ruby 2.0–2.1
ruby22	4.3.0	Ruby 2.2–2.3
ruby24	4.8.0	Ruby 2.4–3.2
tcl	3.0.0	Tcl ARE 8.4
tcl85	4.3.0	Tcl ARE 8.5
tcl86	4.0.0	Tcl ARE 8.6
postgres	4.0.0	PostgreSQL ARE
tclbre84	4.0.0	Tcl BRE 8.4

tclbre85	4.3.0	Tcl BRE 8.5
tclbre86	4.0.0	Tcl BRE 8.6
postgresbre	4.0.0	PostgreSQL BRE
tclere84	4.0.0	Tcl ERE 8.4
tclere85	4.3.0	Tcl ERE 8.5
tclere86	4.0.0	Tcl ERE 8.6
postgresere	4.0.0	PostgreSQL ERE
phpereg	4.0.0	PHP ereg 4.3.3–5.2.17
bre	3.0.0	POSIX BRE
ere	3.0.0	POSIX ERE
gnubre	3.1.0	GNU BRE
gnuere	3.1.0	GNU ERE
oracle10gR1	4.0.0	Oracle 10gR1
oracle	3.0.0	Oracle 10gR2–12cR1
xml	3.0.0	XML Schema
xpath	3.1.0	XPath
stdreecmaansivc2008	4.2.0	std::regex ECMAScript (Visual C++ 2008)
stdreecmaansivc2010	4.2.0	std::regex ECMAScript (Visual C++ 2010)
stdreecmaansivc2012	4.2.0	std::regex ECMAScript (Visual C++ 2012)
stdreecmaansivc2013	4.2.0	std::regex ECMAScript (Visual C++ 2013)
stdreecmaansivc2015	4.5.0	std::regex ECMAScript (Visual C++ 2015 & C++Builder 10–10.4)
stdreecmaansivc2017	4.8.0	std::regex ECMAScript (Visual C++ 2017–2022)
stdreecmaansicbx3	4.2.0	std::regex ECMAScript (C++Builder Win64 XE3–XE6)
stdreecmaansicbx7	4.2.0	std::regex ECMAScript (C++Builder Win64 XE7–XE8)
stdreecmaansicb11	4.13.0	std::regex ECMAScript (C++Builder 11)
stdreecmawidevc2008	4.2.0	std::wregex ECMAScript (Visual C++ 2008)
stdreecmawidevc2010	4.2.0	std::wregex ECMAScript (Visual C++ 2010)
stdreecmawidevc2012	4.2.0	std::wregex ECMAScript (Visual C++ 2012–2013)
stdreecmawidevc2015	4.5.0	std::wregex ECMAScript (Visual C++ 2015)
stdreecmawidevc2017	4.8.0	std::wregex ECMAScript (Visual C++ 2017–2022)
stdreecmawidecb3	4.2.0	std::wregex ECMAScript (C++Builder Win64 XE3–XE6)
stdreecmawidecb7	4.2.0	std::wregex ECMAScript (C++Builder Win64 XE7–XE8)
stdreecmawidecx10	4.5.0	std::wregex ECMAScript (C++Builder 10–10.4)
stdreecmawidecb11	4.13.0	std::wregex ECMAScript (C++Builder 11)
stdrebreansivc2008	4.2.0	std::regex basic (Visual C++ 2008)
stdrebreansivc2010	4.2.0	std::regex basic (Visual C++ 2010)
stdrebreansivc2012	4.7.0	std::regex basic (Visual C++ 2012–2013)
stdrebreansivc2015	4.5.0	std::regex basic (Visual C++ 2015 & C++Builder 10–10.4)
stdrebreansicbx3	4.2.0	std::regex basic (C++Builder Win64 XE3–XE6)
stdrebreansicbx7	4.7.0	std::regex basic (C++Builder Win64 XE7–XE8)
stdrebreansicb11	4.13.0	std::regex basic (C++Builder 11)

<code>stdrebrewidevc2008</code>	4.2.0	<code>std::wregex basic</code> (Visual C++ 2008)
<code>stdrebrewidevc2010</code>	4.2.0	<code>std::wregex basic</code> (Visual C++ 2010)
<code>stdrebrewidevc2012</code>	4.7.0	<code>std::wregex basic</code> (Visual C++ 2012–2015)
<code>stdrebrewidecbxe3</code>	4.2.0	<code>std::wregex basic</code> (C++Builder Win64 XE3–XE6)
<code>stdrebrewidecbxe7</code>	4.7.0	<code>std::wregex basic</code> (C++Builder Win64 XE7–XE8 & C++Builder 10–10.4)
<code>stdrebrewidecb11</code>	4.13.0	<code>std::wregex basic</code> (C++Builder 11)
<code>stdregrepansivc2008</code>	4.2.0	<code>std::regex grep</code> (Visual C++ 2008)
<code>stdregrepansivc2010</code>	4.2.0	<code>std::regex grep</code> (Visual C++ 2010)
<code>stdregrepansivc2012</code>	4.7.0	<code>std::regex grep</code> (Visual C++ 2012–2013)
<code>stdregrepansivc2015</code>	4.5.0	<code>std::regex grep</code> (Visual C++ 2015 & C++Builder 10–10.4)
<code>stdregrepansicbxe3</code>	4.2.0	<code>std::regex grep</code> (C++Builder Win64 XE3–XE6)
<code>stdregrepansicbxe7</code>	4.7.0	<code>std::regex grep</code> (C++Builder Win64 XE7–XE8)
<code>stdregrepansicb11</code>	4.13.0	<code>std::regex grep</code> (C++Builder 11)
<code>stdregrepwidevc2008</code>	4.2.0	<code>std::wregex grep</code> (Visual C++ 2008)
<code>stdregrepwidevc2010</code>	4.2.0	<code>std::wregex grep</code> (Visual C++ 2010)
<code>stdregrepwidevc2012</code>	4.7.0	<code>std::wregex grep</code> (Visual C++ 2012–2015)
<code>stdregrepwidecbxe3</code>	4.2.0	<code>std::wregex grep</code> (C++Builder Win64 XE3–XE6)
<code>stdregrepwidecbxe7</code>	4.7.0	<code>std::wregex grep</code> (C++Builder Win64 XE7–XE8 & C++Builder 10–10.4)
<code>stdregrepwidecb11</code>	4.13.0	<code>std::wregex grep</code> (C++Builder 11)
<code>stdreereansivc2008</code>	4.2.0	<code>std::regex extended</code> (Visual C++ 2008)
<code>stdreereansivc2010</code>	4.2.0	<code>std::regex extended</code> (Visual C++ 2010–2013)
<code>stdreereansivc2015</code>	4.5.0	<code>std::regex extended</code> (Visual C++ 2015 & C++Builder 10–10.4)
<code>stdreereansicbxe3</code>	4.2.0	<code>std::regex extended</code> (C++Builder Win64 XE3–XE8)
<code>stdreereansicb11</code>	4.13.0	<code>std::regex extended</code> (C++Builder 11)
<code>stdreerewidevc2008</code>	4.2.0	<code>std::wregex extended</code> (Visual C++ 2008)
<code>stdreerewidevc2010</code>	4.2.0	<code>std::wregex extended</code> (Visual C++ 2010–2015)
<code>stdreerewidecbxe3</code>	4.2.0	<code>std::wregex extended</code> (C++Builder Win64 XE3–XE8 & C++Builder 10–10.4)
<code>stdreerewidecb11</code>	4.13.0	<code>std::wregex extended</code> (C++Builder 11)
<code>stdreeregrepansivc2008</code>	4.2.0	<code>std::regex egrep</code> (Visual C++ 2008)
<code>stdreeregrepansivc2010</code>	4.2.0	<code>std::regex egrep</code> (Visual C++ 2010–2013)
<code>stdreeregrepansivc2015</code>	4.5.0	<code>std::regex egrep</code> (Visual C++ 2015 & C++Builder 10–10.4)
<code>stdreeregrepansicbxe3</code>	4.2.0	<code>std::regex egrep</code> (C++Builder Win64 XE3–XE8)
<code>stdreeregrepansicb11</code>	4.13.0	<code>std::regex egrep</code> (C++Builder 11)
<code>stdreeregrepwidevc2008</code>	4.2.0	<code>std::wregex egrep</code> (Visual C++ 2008)
<code>stdreeregrepwidevc2010</code>	4.2.0	<code>std::wregex egrep</code> (Visual C++ 2010–2015)
<code>stdreeregrepwidecbxe3</code>	4.2.0	<code>std::wregex egrep</code> (C++Builder Win64 XE3–XE8 & C++Builder 10–10.4)
<code>stdreeregrepwidecb11</code>	4.13.0	<code>std::wregex egrep</code> (C++Builder 11)
<code>stdreawkansivc2008</code>	4.2.0	<code>std::regex awk</code> (Visual C++ 2008)
<code>stdreawkansivc2010</code>	4.2.0	<code>std::regex awk</code> (Visual C++ 2010–2012)
<code>stdreawkansivc2013</code>	4.2.0	<code>std::regex awk</code> (Visual C++ 2013)
<code>stdreawkansivc2015</code>	4.5.0	<code>std::regex awk</code> (Visual C++ 2015 & C++Builder 10–10.4)

stdreawkansi1038	4.2.0	std::regex awk (C++Builder Win64 XE3–XE6)
stdreawkansi1042	4.2.0	std::regex awk (C++Builder Win64 XE7–XE8)
stdreawkansi1044	4.13.0	std::regex awk (C++Builder 11)
stdreawkwidevc2008	4.2.0	std::wregex awk (Visual C++ 2008)
stdreawkwidevc2010	4.2.0	std::wregex awk (Visual C++ 2010–2015)
stdreawkwidecbxe3	4.2.0	std::wregex awk (C++Builder Win64 XE3–XE8 & C++Builder 10–10.4)
stdreawkwidecb11	4.13.0	std::wregex awk (C++Builder 11)
boostecmaansi1038	4.7.0	boost::regex ECMAScript 1.38–1.39
boostecmaansi1042	4.7.0	boost::regex ECMAScript 1.42–1.43
boostecmaansi1044	4.7.0	boost::regex ECMAScript 1.44–1.46
boostecmaansi1047	4.7.0	boost::regex ECMAScript 1.47
boostecmaansi1048	4.8.0	boost::regex ECMAScript 1.48–1.49
boostecmaansi1050	4.13.0	boost::regex ECMAScript 1.50–1.53
boostecmaansi1054	4.7.0	boost::regex ECMAScript 1.54–1.57
boostecmaansi1058	4.7.0	boost::regex ECMAScript 1.58–1.59
boostecmaansi1060	4.7.0	boost::regex ECMAScript 1.60–1.61
boostecmaansi1062	4.7.0	boost::regex ECMAScript 1.62–1.63
boostecmaansi1064	4.8.0	boost::regex ECMAScript 1.64–1.65
boostecmaansi1066	4.9.0	boost::regex ECMAScript 1.66–1.77
boostecmaansi1078	4.14.0	boost::regex ECMAScript 1.78–1.83
boostecmawide1038	4.7.0	boost::wregex ECMAScript 1.38–1.39
boostecmawide1042	4.7.0	boost::wregex ECMAScript 1.42–1.43
boostecmawide1044	4.7.0	boost::wregex ECMAScript 1.44–1.46
boostecmawide1047	4.7.0	boost::wregex ECMAScript 1.47
boostecmawide1048	4.8.0	boost::wregex ECMAScript 1.48–1.49
boostecmawide1050	4.13.0	boost::wregex ECMAScript 1.50–1.53
boostecmawide1054	4.7.0	boost::wregex ECMAScript 1.54–1.57
boostecmawide1058	4.7.0	boost::wregex ECMAScript 1.58–1.59
boostecmawide1060	4.7.0	boost::wregex ECMAScript 1.60–1.63
boostecmawide1064	4.8.0	boost::wregex ECMAScript 1.64–1.65
boostecmawide1066	4.9.0	boost::wregex ECMAScript 1.66–1.77
boostecmawide1078	4.14.0	boost::wregex ECMAScript 1.78–1.83
boostbreansi1038	4.7.0	boost::regex basic 1.38–1.39
boostbreansi1042	4.7.0	boost::regex basic 1.42–1.61
boostbreansi1062	4.7.0	boost::regex basic 1.62–1.83
boostbrewide1038	4.7.0	boost::wregex basic 1.38–1.39
boostbrewide1042	4.7.0	boost::wregex basic 1.42–1.59
boostbrewide1060	4.7.0	boost::wregex basic 1.60–1.83
boostgrepansi1038	4.7.0	boost::regex grep 1.38–1.39
boostgrepansi1042	4.7.0	boost::regex grep 1.42–1.61
boostgrepansi1062	4.7.0	boost::regex grep 1.62–1.83

boostgrepwide1038	4.7.0	boost::wregex grep 1.38–1.39
boostgrepwide1042	4.7.0	boost::wregex grep 1.42–1.59
boostgrepwide1060	4.7.0	boost::wregex grep 1.60–1.83
boostereansi1038	4.7.0	boost::regex extended 1.38–1.39
boostereansi1042	4.7.0	boost::regex extended 1.42–1.61
boostereansi1062	4.7.0	boost::regex extended 1.62–1.77
boostereansi1078	4.14.0	boost::regex extended 1.78–1.83
boosterewide1038	4.7.0	boost::wregex extended 1.38–1.39
boosterewide1042	4.7.0	boost::wregex extended 1.42–1.59
boosterewide1060	4.7.0	boost::wregex extended 1.60–1.77
boosterewide1078	4.14.0	boost::wregex extended 1.78–1.83
boostegrepansi1038	4.7.0	boost::regex egrep 1.38–1.39
boostegrepansi1042	4.7.0	boost::regex egrep 1.42–1.61
boostegrepansi1062	4.7.0	boost::regex egrep 1.62–1.77
boostegrepansi1078	4.14.0	boost::regex egrep 1.78–1.83
boostegrepwide1038	4.7.0	boost::wregex egrep 1.38–1.39
boostegrepwide1042	4.7.0	boost::wregex egrep 1.42–1.59
boostegrepwide1060	4.7.0	boost::wregex egrep 1.60–1.77
boostegrepwide1078	4.14.0	boost::wregex egrep 1.78–1.83
boostawkansi1038	4.7.0	boost::regex awk 1.38–1.39
boostawkansi1042	4.7.0	boost::regex awk 1.42–1.61
boostawkansi1062	4.7.0	boost::regex awk 1.62–1.77
boostawkansi1078	4.14.0	boost::regex awk 1.78–1.83
boostawkwide1038	4.7.0	boost::wregex awk 1.38–1.39
boostawkwide1042	4.7.0	boost::wregex awk 1.42–1.59
boostawkwide1060	4.7.0	boost::wregex awk 1.60–1.77
boostawkwide1078	4.14.0	boost::wregex awk 1.78–1.83

Replacement Flavor Identifiers

Identifiers for the `-flavorreplace` command line parameter and for the custom application variant structure in the COM interface.

Identifier	Age	Replace Flavor
jpgsoft5	4.0.0	JGsoft V2 (EditPad 8; PowerGREP 5)
jpgsoft	3.0.0	JGsoft (EditPad 6 & 7; PowerGREP 3 & 4)
jpgsoftpcr	4.0.0	JGsoft (EditPad 5; PowerGREP 2)
delphixe	4.2.0	Delphi XE–XE3 (TRegEx)
delphixe4	4.0.0	Delphi XE4–XE5 (TRegEx)
delphi	3.0.0	Delphi 10.2–11 (TRegEx & TPerlRegEx) & Delphi 2007–2009 (TPerlRegEx)

delphixecore	4.2.0	Delphi XE–XE3 (TPerlRegEx)
delphixe4core	4.2.0	Delphi XE6–10.1 (TRegEx & TPerlRegEx) & Delphi XE4–XE5 (TPerlRegEx)
dotnet	3.0.0	.NET 1.1–7.0
dotnetecma	4.5.0	.NET 1.1–7.0 (ECMAScript)
java	3.0.0	Java 4–6
java7	4.0.0	Java 7–21
perl	3.0.0	Perl 5.8
perl510	4.0.0	Perl 5.10–5.12
perl514	4.0.0	Perl 5.14–5.16
perl518	4.0.0	Perl 5.18–5.20
perl522	4.4.0	Perl 5.22–5.32
pcre1010	4.3.0	PCRE2 10.10–10.20
pcre1021	4.6.0	PCRE2 10.21–10.39
pcrex1021	4.6.0	PCRE2 extended 10.21–10.39
javascript	3.0.0	JavaScript (Chrome)
firefox	4.0.0	JavaScript (Firefox)
msie	4.0.0	JavaScript (MSIE standard)
quirks	4.0.0	JavaScript (MSIE quirks)
xregexp2firefox	4.5.0	XRegExp 2–3 (Chrome & Firefox)
xregexp2	4.0.0	XRegExp 2-3 (MSIE standard & quirks)
xregexp4firefox	4.13.0	XRegExp 4 (Chrome & Firefox)
xregexp4msie	4.13.0	XRegExp 4 (MSIE standard)
xregexp5firefox	4.12.0	XRegExp 5 (Chrome & Firefox)
xregexp5msie	4.12.0	XRegExp 5 (MSIE standard)
vbscript	3.1.0	VBScript
python	3.0.0	Python 2.4–3.4
python35	4.5.0	Python 3.5–3.6
python37	4.9.0	Python 3.7–3.12
ruby	3.0.0	Ruby 1.8
ruby19	4.0.0	Ruby 1.9–3.2
tcl	3.0.0	Tcl 8.4–8.5
tcl86	4.0.0	Tcl 8.6
phpereg	3.0.0	PHP ereg 4.3.3–5.2.17
phppreg	3.0.0	PHP preg 4.3.3–5.6.40
phppreg7	4.6.0	PHP preg 7.0.0–8.1.24
r	3.2.0	R 2.14.0–3.6.3
r4000	4.11.0	R 4.0.0–4.2.1
realbasic	3.0.0	REALbasic/Xojo
oracle	3.0.0	Oracle
postgres	3.0.0	PostgreSQL
xpath	3.0.0	XPath

<code>stdreansivc</code>	4.2.0	<code>std::regex</code> ECMAScript (Visual C++ 2008–2013)
<code>stdreansivc2015</code>	4.5.0	<code>std::regex</code> ECMAScript (Visual C++ 2015–2022 & C++Builder 10–10.4)
<code>stdreansicb</code>	4.2.0	<code>std::regex</code> ECMAScript (C++Builder Win64 XE3–XE8)
<code>stdrewide</code>	4.2.0	<code>std::wregex</code> ECMAScript (Visual C++ 2008–2013 & C++Builder Win64 XE3–XE8)
<code>stdrewidevc2015</code>	4.5.0	<code>std::wregex</code> ECMAScript (Visual C++ 2015–2022 & C++Builder 10–11)
<code>stdresedansivc2008</code>	4.6.0	<code>std::regex</code> sed (Visual C++ 2008–2013)
<code>stdresedansivc2015</code>	4.6.0	<code>std::regex</code> sed (Visual C++ 2015–2022 & C++Builder 10–10.4)
<code>stdresedansicbxe3</code>	4.6.0	<code>std::regex</code> sed (C++Builder Win64 XE3–XE8)
<code>stdresedwidevc2008</code>	4.6.0	<code>std::wregex</code> sed (Visual C++ 2008–2013 & C++Builder Win64 XE3–XE8)
<code>stdresedwidevc2015</code>	4.6.0	<code>std::wregex</code> sed (Visual C++ 2015–2022 & C++Builder 10–11)
<code>boostallansi1038</code>	4.7.0	<code>boost::regex</code> all 1.38–1.39
<code>boostallansi1042</code>	4.7.0	<code>boost::regex</code> all 1.42–1.83
<code>boostallwide1038</code>	4.7.0	<code>boost::wregex</code> all 1.38–1.39
<code>boostallwide1042</code>	4.7.0	<code>boost::wregex</code> all 1.42–1.83
<code>boostecmaansi1038</code>	4.7.0	<code>boost::regex</code> ECMAScript 1.38–1.39
<code>boostecmaansi1042</code>	4.7.0	<code>boost::regex</code> ECMAScript 1.42–1.83
<code>boostecmawide1038</code>	4.7.0	<code>boost::wregex</code> ECMAScript 1.38–1.39
<code>boostecmawide1042</code>	4.7.0	<code>boost::wregex</code> ECMAScript 1.42–1.83
<code>boostsedansi1038</code>	4.7.0	<code>boost::regex</code> sed 1.38–1.39 & 1.42–1.83
<code>boostsedwide1038</code>	4.7.0	<code>boost::wregex</code> sed 1.38–1.39 & 1.42–1.83

Split Flavor Identifiers

Identifiers for the `-flavorsplit` command line parameter and for the custom application variant structure in the COM interface.

Identifier	Age	Split Flavor
<code>delphi</code>	4.0.0	Delphi XE–XE5 (TRegEx)
<code>delphixe6</code>	4.1.0	Delphi XE6–XE8 & 10–11 (TRegEx)
<code>tperlregex</code>	4.0.0	Delphi 2007–XE8 & 10–11 (TPerlRegEx)
<code>dotnet11</code>	4.0.0	.NET 1.1
<code>dotnet</code>	4.0.0	.NET 2.0–7.0
<code>java</code>	4.0.0	Java 4–7
<code>java8</code>	4.1.0	Java 8–21
<code>perl</code>	4.0.0	Perl 5.8–5.32
<code>javascript</code>	4.0.0	JavaScript (Chrome & Firefox)
<code>msie</code>	4.0.0	JavaScript (MSIE standard)
<code>quirks</code>	4.0.0	JavaScript (MSIE quirks)
<code>python</code>	4.0.0	Python 2.4–3.4
<code>python35</code>	4.5.0	Python 3.5–3.6

python37	4.9.0	Python 3.7-3.12
ruby	4.0.0	Ruby 1.8–3.2
phpereg	4.0.0	PHP ereg 4.3.3–5.2.17
phppreg	4.0.0	PHP preg 4.3.3–5.6.40, 7.0.5–8.1.24
phppreg7000	4.6.0	PHP preg 7.0.0–7.0.4
r	4.0.0	R 2.14.0–4.2.1
postgres	4.0.0	PostgreSQL
xpath	4.0.0	XPath

String Style Identifiers

Identifiers for the `-string` command line parameter and for the `StringType` parameter in the COM interface.

Identifier	Age	String Style
asis	1.0.0	as is
basic	1.0.0	Basic string
c	1.0.0	C string
cL	4.2.0	C wide string
c11	4.5.0	C++11 raw string
c11L	4.5.0	C++11 wide raw string
c#	3.0.0	C# string
pascal	1.0.0	Delphi string
prism	3.3.0	Delphi Prism string
groovy	3.3.0	Groovy string
java	3.1.0	Java string
javascriptstring	3.2.0	JavaScript string
javascript	1.0.0	JavaScript <code>//</code> operator
perl	1.0.0	Perl string
perlop	1.0.0	Perl <code>m//</code> or <code>s///</code> operator
phpereg	3.0.0	PHP string
phppreg	1.0.0	PHP <code>"//"</code> string
postgres	3.0.0	PostgreSQL string
powershell	3.2.0	PowerShell string
python	3.0.0	Python string
r	4.0.0	R string
ruby	3.0.0	Ruby <code>//</code> operator
scala	3.3.0	Scala string
sql	3.0.0	SQL string
tcl	3.0.0	Tcl word
xml	3.0.0	XML

Text File Encoding Identifiers

Identifier	Encoding
bytes	Binary file to be opened in hexadecimal mode
utf8	Unicode, UTF-8
utf32le	Unicode, UTF-32 little endian
utf32be	Unicode, UTF-32 big endian
utf16le	Unicode, UTF-16 little endian
utf16be	Unicode, UTF-16 big endian
win1250	Windows 1250: Central European
win1251	Windows 1251: Cyrillic
win1252	Windows 1252: Western European
win1253	Windows 1253: Greek
win1254	Windows 1254: Turkish
win1255	Windows 1255: Hebrew
win1256	Windows 1256: Arabic
win1257	Windows 1257: Baltic
win1258	Windows 1258: Vietnam
win874	Windows 874: Thai
8859-1	ISO-8859-1 Latin-1 Western European
8859-2	ISO-8859-2 Latin-2 Central European
8859-3	ISO-8859-3 Latin-3 South European
8859-4	ISO-8859-4 Latin-4 North European
8859-5	ISO-8859-5 Cyrillic
8859-6	ISO-8859-6 Arabic
8859-7	ISO-8859-7 Greek
8859-8	ISO-8859-8 Hebrew
8859-9	ISO-8859-9 Latin-5 Turkish
8859-10	ISO-8859-10 Latin-6 Nordic
8859-11	ISO-8859-11 Thai (TIS-620)
8859-13	ISO-8859-13 Latin-7 Baltic Rim
8859-14	ISO-8859-14 Latin-8 Celtic
8859-15	ISO-8859-15 Latin-9
8859-16	ISO-8859-16 Latin-10 South-Eastern European
dos437	DOS 437: United States
dos737	DOS 737: Greek
dos775	DOS 775: Baltic Rim
dos850	DOS 850: Western European
dos852	DOS 852: Central European
dos855	DOS 855: Cyrillic
dos857	DOS 857: Turkish

dos860	DOS 860: Portuguese
dos861	DOS 861: Icelandic
dos862	DOS 862: Hebrew
dos863	DOS 863: Canadian French
dos864	DOS 864: Arabic
dos865	DOS 865: Nordic
dos866	DOS 866: Cyrillic Russian
dos869	DOS 869: Greek 2
koi8r	KOI8-R: Russian
koi8u	KOI8-U: Ukrainian
ebcdic037	EBCDIC 037: US & Canada
ebcdic424	EBCDIC 424: Hebrew
ebcdic500	EBCDIC 500: International
ebcdic875	EBCDIC 875: Greek
ebcdic1026	EBCDIC 1026: Turkish
10585	ISO-10585: Armenian
armscii7	ArmSCII-7: Armenian
armscii8	ArmSCII-8: Armenian
armscii8a	ArmSCII-8A: Armenian
geostd8	GEOSTD8: Georgian
isiri-3342	ISIRI 3342: Farsi
kamenicky	Kamenický: Czech & Slovak
kz-1048	KZ-1048: Kazach
mazovia	Mazovia: Polish
mik	MIK: Bulgarian
ptcp154	PTCP154: Cyrillic Asian
tcvn	TCVN: Vietnamese
viscii	VISCII: Vietnamese
vps	VPS: Vietnamese
yuscii-cyr	YUSCII Cyrillic: Yugoslavia
yuscii-lat	YUSCII Latin: Yugoslavia
ascii	US-ASCII (7-bit)
win932	Windows 932: Japanese (Shift-JIS)
win949	Windows 949: Korean
win936	Windows 936: Simplified Chinese (GBK)
win950	Windows 950: Traditional Chinese (Big5)
euc-jp	EUC-JP: Japanese (JIS 201+208)
euc-jp-0212	EUC-JP-212: Japanese (JIS 201+208+212)
euc-kr	EUC-KR: Korean (KS 1001)
euc-cn	EUC-CN: Simplified Chinese (GB 2312)
euc-tw	EUC-TW: Traditional Chinese (CNS 11643)

uffff	ASCII + \uFFFF Unicode Escapes
ncrhex	ASCII + NCR
ncrdec	ASCII + NCR
htmlentities	ASCII + &htmlchar;
iscii-dev	ISCII Devanagari
iscii-bng	ISCII Bengali & Assamese
iscii-pnj	ISCII Punjabi (Gurmukhi)
iscii-gjr	ISCII Gujarati
iscii-ori	ISCII Oriya
iscii-tml	ISCII Tamil
iscii-tlg	ISCII Telugu
iscii-knd	ISCII Kannada
iscii-mlm	ISCII Malayalam
vni	VNI: Vietnamese
viqr	VIQR: Vietnamese
mac-arabic	Mac Arabic
mac-celtic	Mac Celtic
mac-ce	Mac Central European
mac-croatian	Mac Croatian
mac-cyrillic	Mac Cyrillic
mac-dingbats	Mac Dingbats
mac-farsi	Mac Farsi
mac-gaelic	Mac Gaelic
mac-greek	Mac Greek
mac-hebrew	Mac Hebrew
mac-icelandic	Mac Icelandic
mac-inuit	Mac Inuit
mac-roman	Mac Roman (Western European)
mac-romanian	Mac Romanian
mac-symbol	Mac Symbol
mac-thai	Mac Thai
mac-turkish	Mac Turkish
mac-chinesesimp	Mac Chinese Simplified
mac-chinesetrad	Mac Chinese Traditional
mac-devanagari	Mac Devanagari
mac-gujarati	Mac Gujarati
mac-gurmukhi	Mac Gurmukhi
mac-japanese	Mac Japanese
mac-korean	Mac Korean
utf7	Unicode, UTF-7
2022-jp	ISO-2022-JP: Japanese (JIS 201+208)

2022-jp-1	ISO-2022-JP-1: Japanese (JIS 201+208+212)
2022-jp-2	ISO-2022-JP-2: Japanese multilingual (JIS 201+208+212)
2022-kr	ISO-2022-KR: Korean (KS 1001)
2022-cn	ISO-2022-CN: Chinese (GB 2312 + CNS 11643)
hz	HZ: Simplified Chinese
tscii	TSCII: Tamil

55. Contact RegexBuddy's Developer and Publisher

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

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

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

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

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

To request technical support, please use the Support and Feedback command in the Help menu. This command will show some basic information about your computer and your copy of RegexBuddy. 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 RegexBuddy, and thus cannot access the Support and Feedback command, you can email support@regexbuddy.com. You can expect to receive a reply by the next business day. For instant gratification, try the forums.

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

Where to Buy (More Copies of) RegexBuddy

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

Part 2

Regular Expressions Tutorial

1. Regular Expressions Tutorial

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

The tutorial doesn't stop there. It also explains how a regular expression engine works on the inside and alerts you to the consequences. This helps you to quickly understand why a particular regex does not do what you initially expected. It will save you lots of guesswork and head scratching when you need to write more complex regexes.

What Regular Expressions Are Exactly - Terminology

Basically, a regular expression is a pattern describing a certain amount of text. Their name comes from the mathematical theory on which they are based. But we will not dig into that. You will usually find the name abbreviated to “regex” or “regexp”. This tutorial uses “regex”, because it is easy to pronounce the plural “regexes”. In this book, regular expressions are shaded gray as `regex`.

This first example is actually a perfectly valid regex. It is the most basic pattern, simply matching the literal text `regex`. A “match” is the piece of text, or sequence of bytes or characters that pattern was found to correspond to by the regex processing software. Matches are highlighted in blue in this book.

`\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}\b` is a more complex pattern. It describes a series of letters, digits, dots, underscores, percentage signs and hyphens, followed by an at sign, followed by another series of letters, digits and hyphens, finally followed by a single dot and two or more letters. In other words: this pattern describes an email address. This also shows the syntax highlighting applied to regular expressions in this book. Word boundaries and quantifiers are blue, character classes are orange, and escaped literals are gray. You'll see additional colors like green for grouping and purple for meta tokens later in the tutorial.

With the above regular expression pattern, you can search through a text file to find email addresses, or verify if a given string looks like an email address. This tutorial uses the term “string” to indicate the text that the regular expression is applied to. This book highlights them in `green`. The term “string” or “character string” is used by programmers to indicate a sequence of characters. In practice, you can use regular expressions with whatever data you can access using the application or programming language you are working with.

Different Regular Expression Engines

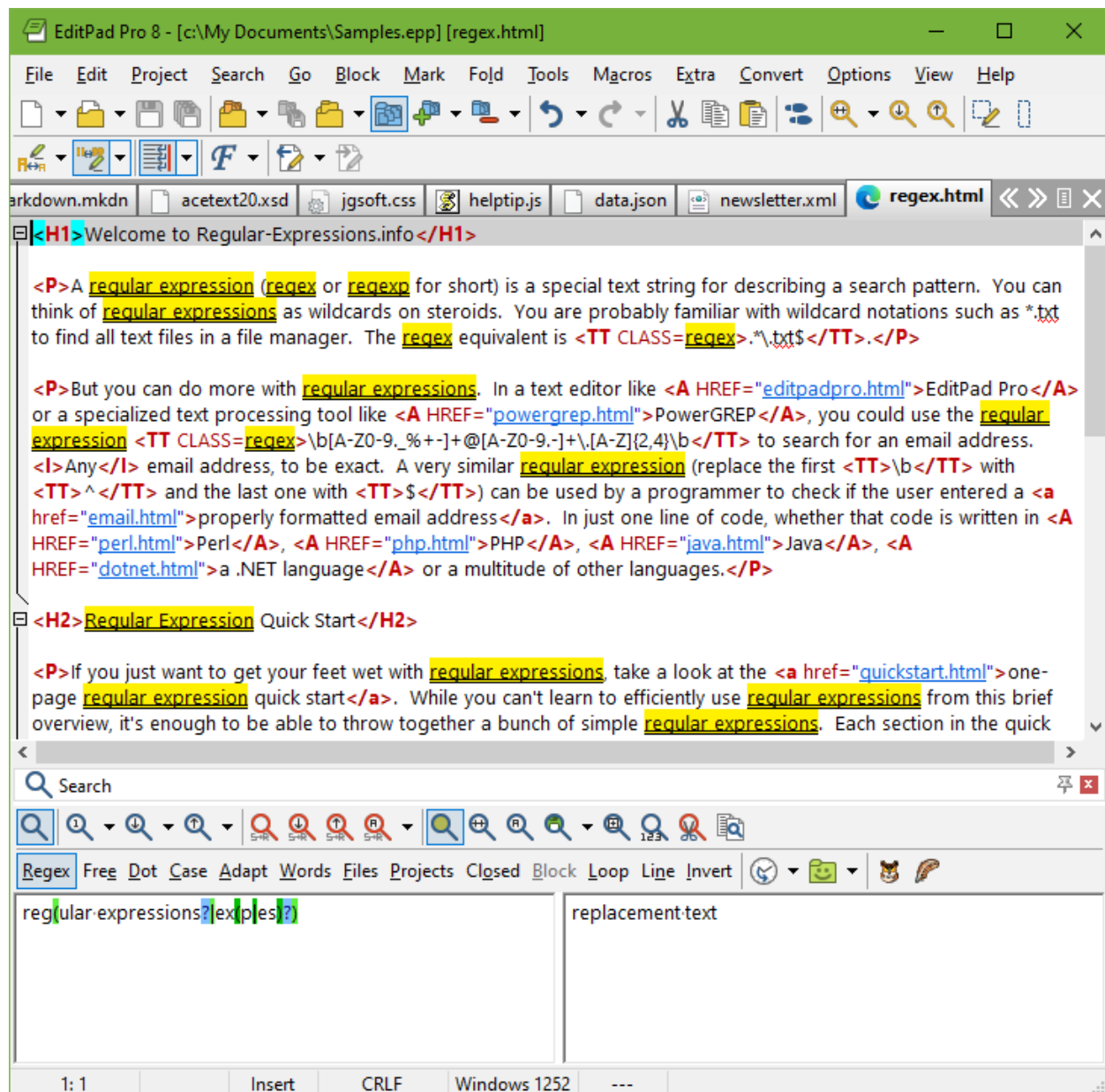
A regular expression “engine” is a piece of software that can process regular expressions, trying to match the pattern to the given string. Usually, the engine is part of a larger application and you do not access the engine directly. Rather, the application invokes it for you when needed, making sure 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. The syntax and behavior of a particular engine is called a regular expression flavor. This tutorial covers all the popular regular expression flavors, including Perl, PCRE, PHP, .NET, Java, JavaScript, XRegExp, VBScript, Python, Ruby, Delphi, R, Tcl, POSIX, and many others. The tutorial alerts you when these flavors require

different syntax or show different behavior. Even if your application is not explicitly covered by the tutorial, it likely uses a regex flavor that is covered, as most applications are developed using one of the programming environments or regex libraries just mentioned.

Give Regexes a First Try

You can easily try the following yourself in a text editor that supports regular expressions, such as EditPad Pro. If you do not have such an editor, you can download the free evaluation version of EditPad Pro to try this out. EditPad Pro's regex engine is fully functional in the demo version.



As a quick test, copy and paste the text of this page into EditPad Pro. Then select Search | Multiline Search Panel in the menu. In the search panel that appears near the bottom, type in `regex` in the box labeled “Search Text”. Mark the “Regular expression” checkbox, and click the Find First button. This is the leftmost button on the search panel. See how EditPad Pro’s regex engine finds the first match. Click the Find Next button, which sits next to the Find First button, to find further matches. When there are no further matches, the Find Next button’s icon flashes briefly.

Now try to search using the regex `reg(ular expressions?|ex(ples)?)`. This regex finds all names, singular and plural, I have used on this page to say “regex”. If we only had plain text search, we would have needed 5 searches. With regexes, we need just one search. Regexes save you time when using a tool like EditPad Pro. Select Count Matches in the Search menu to see how many times this regular expression can match the file you have open in EditPad Pro.

If you are a programmer, your software will run faster since even a simple regex engine applying the above regex once will outperform a state of the art plain text search algorithm searching through the data five times. Regular expressions also reduce development time. With a regex engine, it takes only one line (e.g. in Perl, PHP, Python, Ruby, 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, such as `a`. It matches the first occurrence of that character in the string. If the string is `Jack is a boy`, it matches the `a` after the `J`. The fact that this `a` is in the middle of the word does not matter to the regex engine. If it matters to you, you will need to tell that to the regex engine by using word boundaries. We will get to that later.

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

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

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

Special Characters

Because we want to do more than simply search for literal pieces of text, we need to reserve certain characters for special use. In the regex flavors discussed in this tutorial, there are 12 characters with special meanings: the backslash `\`, the caret `^`, the dollar sign `$`, the period or dot `.`, the vertical bar or pipe symbol `|`, the question mark `?`, the asterisk or star `*`, the plus sign `+`, the opening parenthesis `(`, the closing parenthesis `)`, the opening square bracket `[`, and the opening curly brace `{`. These special characters are often called “metacharacters”. Most of them are errors when used alone.

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

Note that `1+1=2`, with the backslash omitted, is a valid regex. So you won’t get an error message. But it doesn’t match `1+1=2`. It would match `111=2` in `123+111=234`, due to the special meaning of the plus character.

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

Most regular expression flavors treat the brace `{` as a literal character, unless it is part of a repetition operator like `a{1,3}`. So you generally do not need to escape it with a backslash, though you can do so if you want. But there are a few exceptions. Java requires literal opening braces to be escaped. Boost and `std::regex` require all literal braces to be escaped.

`]` is a literal outside character classes. Different rules apply inside character classes. Those are discussed in the topic about character classes. Again, there are exceptions. `std::regex` and Ruby require closing square brackets to be escaped even outside character classes.

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

Escaping a single metacharacter with a backslash works in all regular expression flavors. Some flavors also support the `\Q...VE` escape sequence. All the characters between the `\Q` and the `VE` are interpreted as literal characters. E.g. `\Q*\d+*VE` matches the literal text `*\d+*`. The `VE` may be omitted at the end of the regex, so `\Q*\d+*` is the same as `\Q*\d+*VE`. This syntax is supported by the JGsoft engine, Perl, PCRE, PHP, Delphi, Java, both inside and outside character classes. Java 4 and 5 have bugs that cause `\Q...VE` to misbehave, however, so you shouldn't use this syntax with Java. Boost supports it outside character classes, but not inside.

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 are 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 turns 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.

3. Non-Printable Characters

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

In some flavors, `\v` matches the vertical tab (ASCII 0x0B). In other flavors, `\v` is a shorthand that matches any vertical whitespace character. That includes the vertical tab, form feed, and all line break characters. Perl 5.10, PCRE 7.2, PHP 5.2.4, R, Delphi XE, and later versions treat it as a shorthand. Earlier versions treated it as a needlessly escaped literal `v`. The JGsoft flavor originally matched only the vertical tab with `\v`. JGsoft V2 matches any vertical whitespace with `\v`.

Many regex flavors also support the tokens `\cA` through `\cZ` to insert ASCII control characters. The letter after the backslash is always a lowercase `c`. The second letter is an uppercase letter `A` through `Z`, to indicate Control+A through Control+Z. These are equivalent to `\x01` through `\x1A` (26 decimal). E.g. `\cM` matches a carriage return, just like `\r`, `\x0D`, and `\u000D`. Most flavors allow the second letter to be lowercase, with no difference in meaning. Only Java requires the `A` to `Z` to be uppercase.

Using characters other than letters after `\c` is not recommended because the behavior is inconsistent between applications. Some allow any character after `\c` while other allow ASCII characters. The application may take the last 5 bits that character index in the code page or its Unicode code point to form an ASCII control character. Or the application may just flip bit 0x40. Either way `\c@` through `\c_` would match control characters 0x00 through 0x1F. But `\c*` might match a line feed or the letter `j`. The asterisk is character 0x2A in the ASCII table, so the lower 5 bits are 0x0A while flipping bit 0x40 gives 0x6A. Metacharacters indeed lose their meaning immediately after `\c` in applications that support `\cA` through `\cZ` for matching control characters. The original JGsoft flavor, .NET, and XRegExp are more sensible. They treat anything other than a letter after `\c` as an error.

In XML Schema regular expressions and XPath, `\c` is a shorthand character class that matches any character allowed in an XML name.

The JGsoft flavor originally treated `\cA` through `\cZ` as control characters. But JGsoft V2 treats `\c` as an XML shorthand.

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

If your regex engine works with 8-bit code pages instead of Unicode, then you can include any character in your regular expression if you know its position in the character set that you are working with. In the Latin-1 character set, the copyright symbol is character 0xA9. So to search for the copyright symbol, you can use `\xA9`. Another way to search for a tab is to use `\x09`. Note that the leading zero is required. In Tcl 8.5 and prior you have to be careful with this syntax, because Tcl used to eat up all hexadecimal characters after `\x` and treat the last 4 as a Unicode code point. So `\xA9ABC20AC` would match the euro symbol. Tcl 8.6 only takes the first two hexadecimal digits as part of the `\x`, as all other regex flavors do, so `\xA9ABC20AC` matches `©ABC20AC`.

Line Breaks

`\R` is a special escape that matches any line break, including Unicode line breaks. What makes it special is that it treats CRLF pairs as indivisible. If the match attempt of `\R` begins before a CRLF pair in the string, then a single `\R` matches the whole CRLF pair. `\R` will not backtrack to match only the CR in a CRLF pair. So while `\R` can match a lone CR or a lone LF, `\R{2}` or `\R\R` cannot match a single CRLF pair. The first `\R` matches the whole CRLF pair, leaving nothing for the second one to match.

Or at least, that is how `\R` should work. It works like that in JGsoft V2, Ruby 2.0 and later, Java 8, and PCRE 8.13 and later. Java 9 introduced a bug that allows `\R\R` to match a single CRLF pair. PCRE 7.0 through 8.12 had a bug that allows `\R{2}` to match a single CRLF pair. Perl has a different bug with the same result.

Note that `\R` only looks forward to match CRLF pairs. The regex `\r\R` can match a single CRLF pair. After `\r` has consumed the CR, the remaining lone LF is a valid line break for `\R` to match. This behavior is consistent across all flavors.

Octal Escapes

Many applications also support octal escapes in the form of `\0377` or `\377`, where 377 is the octal representation of the character's position in the character set (255 decimal in this case). There is a lot of variation between regex flavors as to the number of octal digits allowed or required after the backslash, whether the leading zero is required or not allowed, and whether `\0` without additional digits matches a NULL byte. In some flavors this causes complications as `\1` to `\77` can be octal escapes 1 to 63 (decimal) or backreferences 1 to 77 (decimal), depending on how many capturing groups there are in the regex. Therefore, using these octal escapes in regexes is strongly discouraged. Use hexadecimal escapes instead.

Perl 5.14, PCRE 8.34, PHP 5.5.10, and R 3.0.3 support a new syntax `\o{377}` for octal escapes. You can have any number of octal digits between the curly braces, with or without leading zero. There is no confusion with backreferences and literal digits that follow are cleanly separated by the closing curly brace. Do be careful to only put octal digits between the curly braces. In Perl, `\o{whatever}` is not an error but matches a NULL byte.

The JGsoft flavor originally supported octal escapes in the form of `\0377`. JGsoft V2 supports `\o{377}` and treats `\0377` as an error.

Regex Syntax versus String Syntax

Many programming languages support similar escapes for non-printable characters in their syntax for literal strings in source code. Then such escapes are translated by the compiler into their actual characters before the string is passed to the regex engine. If the regex engine does not support the same escapes, this can cause an apparent difference in behavior when a regex is specified as a literal string in source code compared with a regex that is read from a file or received from user input. For example, POSIX regular expressions do not support any of these escapes. But the C programming language does support escapes like `\n` and `\x0A` in string literals. So when developing an application in C using the POSIX library, `\n` is only interpreted as a newline when you add the regex as a string literal to your source code. Then the compiler interprets `\n` and the regex engine sees an actual newline character. If your code reads the same regex from a file, then the regex engine sees `\n`. Depending on the implementation, the POSIX library interprets this as a literal `n` or as

an error. The actual POSIX standard states that the behavior of an “ordinary” character preceded by a backslash is “undefined”.

A similar issue exists in Python 3.2 and prior with the Unicode escape `\uFFFF`. Python has supported this syntax as part of (Unicode) string literals ever since Unicode support was added to Python. But Python’s `re` module only supports `\uFFFF` starting with Python 3.3. In Python 3.2 and earlier, `\uFFFF` works when you add your regex as a literal (Unicode) string to your Python code. But when your Python 3.2 script reads the regex from a file or user input, `\uFFFF` matches `uFFFF` literally as the regex engine sees `\u` as an escaped literal `u`.

4. First Look at How a Regex Engine Works Internally

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

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

While there are many implementations of regular expressions that differ sometimes slightly and sometimes significantly in syntax and behavior, there are basically only two kinds of regular expression engines: text-directed engines, and regex-directed engines. Nearly all modern regex flavors are based on regex-directed engines. This is because certain very useful features, such as lazy quantifiers and backreferences, can only be implemented in regex-directed engines.

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

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

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

The Regex Engine Always Returns the Leftmost Match

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

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

At the 15th character in the string, **c** again matches **c**. The engine then proceeds to attempt to match the remainder of the regex at character 15 and finds that **a** matches **a** and **t** matches **t**.

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

In this first example of the engine’s internals, our regex engine simply appears to work like a regular text search routine. However, it is important that you can follow the steps the engine takes in your mind. In following examples, the way the engine works has a profound impact on the matches it finds. Some of the results may be surprising. But they are always logical and predetermined, once you know how the engine works.

5. Character Classes or Character Sets

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

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

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

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

Negated Character Classes

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

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

Metacharacters Inside Character Classes

In most regex flavors, 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. The POSIX and GNU flavors are an exception. They treat backslashes in character classes as literal characters. So with these flavors, you can't escape anything in character classes.

To include an unescaped caret as a literal, place it anywhere except right after the opening bracket. `[x^]` matches an x or a caret. This works with all flavors discussed in this tutorial.

You can include an unescaped closing bracket by placing it right after the opening bracket, or right after the negating caret. `[]x]` matches a closing bracket or an x. `[^]x]` matches any character that is not a closing bracket or an x. This does not work in JavaScript, which treats `[]` as an empty character class that always fails to match, and `[^]` as a negated empty character class that matches any single character. Ruby treats empty character classes as an error. So both JavaScript and Ruby require closing brackets to be escaped with a backslash to include them as literals in a character class.

The hyphen can be included right after the opening bracket, or right before the closing bracket, or right after the negating caret. Both `[-x]` and `[x-]` match an x or a hyphen. `[^x-]` and `[^x-]` match any character that is not an x or a hyphen. This works in all flavors discussed in this tutorial. Hyphens at other positions in character classes where they can't form a range may be interpreted as literals or as errors. Regex flavors are quite inconsistent about this.

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

Repeating Character Classes

If you repeat a character class by using the `?`, `*`, or `+` operators, you're repeating the entire character class. You're not repeating just the character that it matched. The regex `[0-9]+` can match `837` as well as `222`.

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

Looking Inside The Regex Engine

As was mentioned earlier: the order of the characters inside a character class does not matter. `gr[ae]y` matches `grey` in `Is his hair grey or gray?`, because that is the *leftmost match*. We already saw how the engine applies a regex consisting only of literal characters. Now we'll see how it applies a regex that has more than one permutation. That is: `gr[ae]y` can match both `gray` and `grey`.

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

But because we are using a regex-directed engine, it must continue trying to match all the other permutations of the regex pattern before deciding that the regex cannot be matched with the text starting at character 13. So it continues with the other option, and finds that `e` matches `e`. The last regex token is `y`, which can be matched with the following character as well. The engine has found a complete match with the text starting at character 13. It returns `grey` as the match result, and looks no further. Again, the *leftmost match* is returned, even though we put the `a` first in the character class, and `gray` could have been matched in the string. But the engine simply did not get that far, because another equally valid match was found to the left of it. `gray` is only matched if you tell the regex engine to continue looking for a second match in the remainder of the subject string after the first match.

6. Character Class Subtraction

Character class subtraction is supported by the XML Schema, XPath, .NET (version 2.0 and later), and JGsoft regex flavors. It makes it easy to match any single character present in one list (the character class), but not present in another list (the subtracted class). The syntax for this is `[class-[subtract]]`. If the character after a hyphen is an opening bracket, these flavors interpret the hyphen as the subtraction operator rather than the range operator. You can use the full character class syntax within the subtracted character class.

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

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

Nested Character Class Subtraction

Since you can use the full character class syntax within the subtracted character class, you can subtract a class from the class being subtracted. `[0-9-[0-6-[0-3]]]` first subtracts 0-3 from 0-6, yielding `[0-9-[4-6]]`, or `[0-37-9]`, which matches any character in the string `0123789`.

The class subtraction must always be the last element in the character class. `[0-9-[4-6]a-f]` is not a valid regular expression. It should be rewritten as `[0-9a-f-[4-6]]`. The subtraction works on the whole class. E.g. `[\p{Ll}\p{Lu}-[\p{IsBasicLatin}]]` matches all uppercase and lowercase Unicode letters, except any ASCII letters. The `\p{IsBasicLatin}` is subtracted from the combination of `\p{Ll}\p{Lu}` rather than from `\p{Lu}` alone. This regex will not match `abc`.

While you can use nested character class subtraction, you cannot subtract two classes sequentially. To subtract ASCII characters and Greek characters from a class with all Unicode letters, combine the ASCII and Greek characters into one class, and subtract that, as in `[\p{L}-[\p{IsBasicLatin}\p{IsGreek}]]`.

Negation Takes Precedence over Subtraction

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

Notational Compatibility with Other Regex Flavors

Note that a regex like `[a-z-[aeiou]]` does not cause any errors in most regex flavors that do not support character class subtraction. But it won't match what you intended either. In most flavors, this regex consists

of a character class followed by a literal `]`. The character class matches a character that is either in the range a-z, or a hyphen, or an opening bracket, or a vowel. Since the a-z range and the vowels are redundant, you could write this character class as `[a-z-[]` or `[-[a-z]` in Perl. A hyphen after a range is treated as a literal character, just like a hyphen immediately after the opening bracket. This is true in the XML, .NET and JGsoft flavors too. `[a-z- _]` matches a lowercase letter, a hyphen or an underscore in these flavors.

Strictly speaking, this means that the character class subtraction syntax is incompatible with Perl and the majority of other regex flavors. But in practice there's no difference. Using non-alphanumeric characters in character class ranges is very bad practice because it relies on the order of characters in the ASCII character table. That makes the regular expression hard to understand for the programmer who inherits your work. While `[A-[]` would match any uppercase letter or an opening square bracket in Perl, this regex is much clearer when written as `[A-Z[]`. The former regex would cause an error with the XML, .NET and JGsoft flavors, because they interpret `-[]` as an empty subtracted class, leaving an unbalanced `[`.

7. Character Class Intersection

Character class intersection is supported by Java, JGsoft V2, and by Ruby 1.9 and later. It makes it easy to match any single character that must be present in two sets of characters. The syntax for this is `[class&&[intersect]]`. You can use the full character class syntax within the intersected character class.

If the intersected class does not need a negating caret, then Java and Ruby allow you to omit the nested square brackets: `[class&&intersect]`.

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

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

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

Intersection of Multiple Classes

You can intersect the same class more than once. `[0-9&&[0-6&&[4-9]]]` is the same as `[4-6]` as those are the only digits present in all three parts of the intersection. In Java and Ruby you can write the same regex as `[0-9&&[0-6]&&[4-9]]`, `[0-9&&[0-6&&4-9]]`, `[0-9&&0-6&&[4-9]]`, or just `[0-9&&0-6&&4-9]`. The nested square brackets are only needed if one of the parts of the intersection is negated.

If you do not use square brackets around the right hand part of the intersection, then there is no confusion that the entire remainder of the character class is the right hand part of the intersection. If you do use the square brackets, you could write something like `[0-9&&[12]56]`. In Ruby, this is the same as `[0-9&&1256]`. But Java has bugs that cause it to treat this as `[0-9&&56]`, completely ignoring the nested brackets.

PowerGREP does not allow anything after the nested `]`. The characters `56` in `[0-9&&[12]56]` are an error. This way there is no ambiguity about their meaning.

You also shouldn't put `&&` at the very start or very end of the regex. Ruby treats `[0-9&&]` and `[&&0-9]` as intersections with an empty class, which matches no characters at all. Java ignores leading and trailing `&&` operators. PowerGREP treats them as literal ampersands.

Intersection in Negated Classes

The character class `[^1234&&[3456]]` is both negated and intersected. In Java and PowerGREP, negation takes precedence over intersection. Java and PowerGREP read this regex as “(not 1234) and 3456”. Thus in Java and PowerGREP this class is the same as `[56]` and matches the digits 5 and 6. In Ruby, intersection takes precedence over negation. Ruby reads `[^1234&&3456]` as “not (1234 and 3456)”. Thus in Ruby this class is the same as `[^34]` which matches anything except the digits 3 and 4.

If you want to negate the right hand side of the intersection, then you must use square brackets. Those automatically control precedence. So Java, PowerGREP, and Ruby all read `[1234&&[^3456]]` as “1234 and (not 3456)”. Thus this regex is the same as `[12]`.

Notational Compatibility with Other Regex Flavors

The ampersand has no special meaning in character classes in any other regular expression flavors discussed in this tutorial. The ampersand is simply a literal, and repeating it just adds needless duplicates. All these flavors treat `[1234&&3456]` as identical to `[&123456]`.

Strictly speaking, this means that the character class intersection syntax is incompatible with the majority of other regex flavors. But in practice there’s no difference, because there is no point in using two ampersands in a character class when you just want to add a literal ampersand. A single ampersand is still treated as a literal by Java, Ruby, and PowerGREP.

8. Shorthand Character Classes

Since certain character classes are used often, a series of shorthand character classes are available. `\d` is short for `[0-9]`. In most flavors that support Unicode, `\d` includes all digits from all scripts. Notable exceptions are Java, JavaScript, and PCRE. These Unicode flavors match only ASCII digits with `\d`.

`\w` stands for “word character”. It always matches the ASCII characters `[A-Za-z0-9_]`. Notice the inclusion of the underscore and digits. In most flavors that support Unicode, `\w` includes many characters from other scripts. There is a lot of inconsistency about which characters are actually included. Letters and digits from alphabetic scripts and ideographs are generally included. Connector punctuation other than the underscore and numeric symbols that aren’t digits may or may not be included. XML Schema and XPath even include all symbols in `\w`. Again, Java, JavaScript, and PCRE match only ASCII characters with `\w`.

`\s` stands for “whitespace character”. Again, which characters this actually includes, depends on the regex flavor. In all flavors discussed in this tutorial, it includes `[\t\r\n\f]`. That is: `\s` matches a space, a tab, a carriage return, a line feed, or a form feed. Most flavors also include the vertical tab, with Perl (prior to version 5.18) and PCRE (prior to version 8.34) being notable exceptions. In flavors that support Unicode, `\s` normally includes all characters from the Unicode “separator” category. Java and PCRE are exceptions once again. But JavaScript does match all Unicode whitespace with `\s`.

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 matches `2` (space two), while the latter matches `1` (one). `[\da-fA-F]` matches a hexadecimal digit, and is equivalent to `[0-9a-fA-F]` if your flavor only matches ASCII characters with `\d`.

Negated Shorthand Character Classes

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

Be careful when using the negated shorthands inside square brackets. `[\D\S]` is *not* the same as `[^\d\s]`. The latter matches any character that is neither a digit nor whitespace. It matches `x`, but not `8`. The former, however, matches any character that is either not a digit, or is not whitespace. Because all digits are not whitespace, and all whitespace characters are not digits, `[\D\S]` matches any character; digit, whitespace, or otherwise.

More Shorthand Character Classes

While support for `\d`, `\s`, and `\w` is quite universal, there are some regex flavors that support additional shorthand character classes. Perl 5.10 introduced `\h` and `\v`. `\h` matches horizontal whitespace, which includes the tab and all characters in the “space separator” Unicode category. It is the same as `[\t\p{Zs}]`. `\v` matches “vertical whitespace”, which includes all characters treated as line breaks in the Unicode standard. It is the same as `[\n\cK\f\r\x85\x{2028}\x{2029}]`.

PCRE also supports `\h` and `\v` starting with version 7.2. PHP does as of version 5.2.2, Java as of version 8, and the JGsoft engine as of version 2.

If your flavor supports `\h` and `\v` then you should definitely use them instead of `\s` whenever you want to match only one type of whitespace. Using `\h` instead of `\s` to match spaces and tabs makes sure your regex match doesn't accidentally spill into the next line.

In many other regex flavors, `\v` matches only the vertical tab character. Perl, PCRE, and PHP never supported this, so they were free to give `\v` a different meaning. Java 4 to 7 and JGsoft V1 did use `\v` to match only the vertical tab. Java 8 and JGsoft V2 changed the meaning of this token anyway. The vertical tab is also a vertical whitespace character. To avoid confusion, the above paragraph uses `\ck` to represent the vertical tab.

Boost supports `\h` starting with version 1.42. Boost 1.42 and later support `\v` as a shorthand only outside character classes. `[\v]` matches only the vertical tab in Boost.

Ruby 1.9 and later have their own version of `\h`. It matches a single hexadecimal digit just like `[0-9a-fA-F]`. `\v` is a vertical tab in Ruby.

XML Character Classes

XML Schema, XPath, and JGsoft V2 regular expressions support four more shorthands that aren't supported by any other regular expression flavors. `\i` matches any character that may be the first character of an XML name. `\c` matches any character that may occur after the first character in an XML name. `\I` and `\C` are the respective negated shorthands. Note that the `\c` shorthand syntax conflicts with the control character syntax used in many other regex flavors.

You can use these four shorthands both inside and outside character classes using the bracket notation. They're very useful for validating XML references and values in your XML schemas. The regular expression `\i\c*` matches an XML name like `xml:schema`.

The regex `<\i\c*\s*>` matches an opening XML tag without any attributes. `</\i\c*\s*>` matches any closing tag. `<\i\c*(\s+\i\c*\s*=\s*(("[^"]*"|'['']*'))*\s*>` matches an opening tag with any number of attributes. Putting it all together, `<(\i\c*(\s+\i\c*\s*=\s*(("[^"]*"|'['']*'))*)|/\i\c*)\s*>` matches either an opening tag with attributes or a closing tag.

No other regex flavors discussed in this tutorial support XML character classes. If your XML files are plain ASCII, you can use `[_:A-Za-z]` for `\i` and `[-._:A-Za-z0-9]` for `\c`. If you want to allow all Unicode characters that the XML standard allows, then you will end up with some pretty long regexes. Instead of `\i` you would use:

```
[[:A-Z_a-z\u00C0-\u00D6\u00D8-\u00F6\u00F8-\u02FF\u0370-\u037D\u037F-\u1FFF\u200C-\u200D\u2070-\u218F\u2C00-\u2FEF\u3001-\uD7FF\uF900-\uFDCF\uFDF0-\uFFFD]]
```

Instead of `\c` you would use:

```
[[-.0-9:A-Z_a-z\u00B7\u00C0-\u00D6\u00D8-\u00F6\u00F8-\u037D\u037F-\u1FFF\u200C-\u200D\u203F\u2040\u2070-\u218F\u2C00-\u2FEF\u3001-\uD7FF\uF900-\uFDCF\uFDF0-\uFFFD]]
```

9. The Dot Matches (Almost) Any Character

In regular expressions, the dot or period is one of the most commonly used metacharacters. Unfortunately, it is also the most commonly misused metacharacter.

The dot matches a single character, without caring what that character is. The only exception are line break characters. In all regex flavors discussed in this tutorial, the dot does *not* match line breaks by default.

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

Modern tools and languages can apply regular expressions to very large strings or even entire files. Except for VBScript, all regex flavors discussed here have an option to make the dot match all characters, including line breaks. Older implementations of JavaScript don't have the option either. It was formally added in the ECMAScript 2018 specification.

In PowerGREP, tick the checkbox labeled “dot matches line breaks” to make the dot match all characters. In EditPad Pro, turn on the “Dot” or “Dot matches newline” search option.

In Perl, the mode where the dot also matches line breaks 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 .NET's `Regex` class you activate this mode by specifying `RegexOptions.Singleline`, such as in `Regex.Match("string", "regex", RegexOptions.Singleline)`.

In JavaScript (for compatibility with older browsers) and VBScript you can use a character class such as `[\s\S]` to match any character. This character matches a character that is either a whitespace character (including line break characters), or a character that is not a whitespace character. Since all characters are either whitespace or non-whitespace, this character class matches any character. Do not use alternation like `(\s|\S)` which is slow. And certainly don't use `(.|\s)` which can lead to catastrophic backtracking as spaces and tabs can be matched by both `.` and `\s`.

In all of Boost's regex grammars the dot matches line breaks by default. Boost's ECMAScript grammar allows you to turn this off with `regex_constants::no_mod_m`.

Line Break Characters

While support for the dot is universal among regex flavors, there are significant differences in which characters they treat as line break characters. All flavors treat the newline `\n` as a line break. UNIX text files terminate lines with a single newline. All the scripting languages discussed in this tutorial do not treat any other characters as line breaks. This isn't a problem even on Windows where text files normally break lines with a `\r\n` pair. That's because these scripting languages read and write files in *text mode* by default. When running on Windows, `\r\n` pairs are automatically converted into `\n` when a file is read, and `\n` is automatically written to file as `\r\n`.

std::regex, XML Schema and XPath also treat the carriage return `\r` as a line break character. JavaScript adds the Unicode line separator `\u2028` and paragraph separator `\u2029` on top of that. Java includes these plus the Latin-1 next line control character `\u0085`. Boost adds the form feed `\f` to the list. Only Delphi and the JGsoft flavor supports all Unicode line breaks, completing the mix with the vertical tab.

.NET is notably absent from the list of flavors that treat characters other than `\n` as line breaks. Unlike scripting languages that have their roots in the UNIX world, .NET is a Windows development framework that does not automatically strip carriage return characters from text files that it reads. If you read a Windows text file as a whole into a string, it will contain carriage returns. If you use the regex `abc.*` on that string, without setting `RegexOptions.SingleLine`, then it will match `abc` plus all characters that follow on the same line, plus the carriage return at the end of the line, but without the newline after that.

Some flavors allow you to control which characters should be treated as line breaks. Java has the `UNIX_LINES` option which makes it treat only `\n` as a line break. PCRE has options that allow you to choose between `\n` only, `\r` only, `\r\n`, or all Unicode line breaks.

On POSIX systems, the POSIX locale determines which characters are line breaks. The C locale treats only the newline `\n` as a line break. Unicode locales support all Unicode line breaks.

`\N` Never Matches Line Breaks

Perl 5.12 and PCRE 8.10 introduced `\N` which matches any single character that is not a line break, just like the dot does. Unlike the dot, `\N` is not affected by “single-line mode”. `(?s)\N.` turns on single-line mode and then matches any character that is not a line break followed by any character regardless of whether it is a line break.

PCRE’s options that control which characters are treated as line breaks affect `\N` in exactly the same way as they affect the dot.

PHP 5.3.4 and R 2.14.0 also support `\N` as their regex support is based on PCRE 8.10 or later. JGsoft V2 also supports `\N`.

Use The Dot Sparingly

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

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

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

This regex is still far from perfect. It matches `99/99/99` as a valid date. `[01]\d[- /.][0-3]\d[- /.]\d\d` is a step ahead, though it still matches `19/39/99`. How perfect you want your regex to be depends on what you want to do with it. If you are validating user input, it has to be perfect. If you are parsing data files from a known source that generates its files in the same way every time, our last attempt is probably more than sufficient to parse the data without errors. You can find a better regex to match dates in the example section.

Use Negated Character Classes Instead of the Dot

A negated character class is often more appropriate than the dot. The tutorial section that explains the repeat operators star and plus covers this in more detail. But the warning is important enough to mention it here as well. Again let's illustrate with an example.

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

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

In the date-matching example, we improved our regex by replacing the dot with a character class. Here, we do the same with a negated character class. Our original definition of a double-quoted string was faulty. We do not want any number of *any character* between the quotes. We want any number of characters that are not double quotes or newlines between the quotes. So the proper regex is `"[^\r\n]*"`. If your flavor supports the shorthand `\v` to match any line break character, then `"[^\v]*"` is an even better solution.

10. Start of String and End of String Anchors

Thus far, we have learned about literal characters, character classes, and the dot. Putting one of these in a regex tells the regex engine to try to match a single character.

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

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

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

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 is 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, most 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, `$` still matches 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 Ruby and `std::regex` the caret and dollar also always match at the start and end of each line. In Boost they match at the start and end of each line by default. Boost allows you to turn this off with `regex_constants::no_mod_m` when using the ECMAScript grammar.

In all other programming languages and libraries discussed in this book, 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)`.

Line Break Characters

The tutorial page about the dot already discussed which characters are seen as line break characters by the various regex flavors. This affects the anchors just as much when in multi-line mode, and when the dollar matches before the end of the final break. The anchors handle line breaks that consist of a single character the same way as the dot in each regex flavor.

For anchors there’s an additional consideration when CR and LF occur as a pair and the regex flavor treats both these characters as line breaks. Delphi, Java, and the JGsoft flavor treat CRLF as an indivisible pair. `^` matches after CRLF and `$` matches before CRLF, but neither match in the middle of a CRLF pair. JavaScript and XPath treat CRLF pairs as two line breaks. `^` matches in the middle of and after CRLF, while `$` matches before and in the middle of CRLF.

Permanent Start of String and End of String Anchors

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

JavaScript, POSIX, XML, and XPath do not support `\A` and `\Z`. You’re stuck with using the caret and dollar for this purpose.

The GNU extensions to POSIX regular expressions use `\`` (backtick) to match the start of the string, and `\'` (single quote) to match the end of the string.

Strings Ending with a Line Break

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

Most modern regex flavors have copied this behavior. That includes .NET, Java, PCRE, Delphi, PHP, and Python. This behavior is independent of any settings such as “multi-line mode”.

In all these flavors except Python, `\z` also matches before the final line break. If you only want a match at the absolute very end of the string, use `\z` (lowercase z instead of uppercase Z). `\A\d+\z` does not match `123\n`. `\z` matches after the line break, which is not matched by the shorthand character class.

In Python, `\Z` matches only at the very end of the string. Python does not support `\z`.

Strings Ending with Multiple Line Breaks

If a string ends with multiple line breaks and multi-line mode is off then `$` only matches before the last of those line breaks in all flavors where it can match before the final break. The same is true for `\Z` regardless of multi-line mode.

Boost is the only exception. In Boost, `\Z` can match before any number of trailing line breaks as well as at the very end of the string. So if the subject string ends with three line breaks, Boost's `\Z` has four positions that it can match at. Like in all other flavors, Boost's `\Z` is independent of multi-line mode. Boost's `$` only matches at the very end of the string when you turn off multi-line mode (which is on by default in Boost).

Looking Inside The Regex Engine

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

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

Yet again, the engine must try to match the first token again. Previously, it was successfully matched at the second `4`, so the engine continues at the next character, `8`, where the caret does not match. Same at the `6` and the newline.

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

Since `$` was the last token in the regex, the engine has found a successful match: the last `4` in the string.

11. Word Boundaries

The metacharacter `\b` is an anchor like the caret and the dollar sign. It matches at a position that is called a “word boundary”. This match is zero-length.

There are three different positions that qualify as word boundaries:

- Before the first character in the string, if the first character is a word character.
- After the last character in the string, if the last character is a word character.
- Between two characters in the string, where one is a word character and the other is not a word character.

Simply put: `\b` allows you to perform a “whole words only” search using a regular expression in the form of `\bword\b`. A “word character” is a character that can be used to form words. All characters that are not “word characters” are “non-word characters”.

Exactly which characters are word characters depends on the regex flavor you’re working with. In most flavors, characters that are matched by the short-hand character class `\w` are the characters that are treated as word characters by word boundaries. Java is an exception. Java supports Unicode for `\b` but not for `\w`.

Most flavors, except the ones discussed below, have only one metacharacter that matches both before a word and after a word. This is because any position between characters can never be both at the start and at the end of a word. Using only one operator makes things easier for you.

Since digits are considered to be word characters, `\b4\b` can be used to match a 4 that is not part of a larger number. This regex does not match `44 sheets of a4`. So saying “`\b` matches before and after an alphanumeric sequence” is more exact than saying “before and after a word”.

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

Looking Inside The Regex Engine

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

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

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

Advancing a character and restarting with the first regex token, `\b` matches between the space and the second `i` in the string. Continuing, the regex engine finds that `i` matches `i` and `s` matches `s`. Now, the engine tries to

match the second `\b` at the position before the `l`. This fails because this position is between two word characters. The engine reverts to the start of the regex and advances one character to the `s` in `island`. Again, the `\b` fails to match and continues to do so until the second space is reached. It matches there, but matching the `i` fails.

But `\b` matches at the position before the third `i` in the string. The engine continues, and finds that `i` matches `i` and `s` matches `s`. The last token in the regex, `\b`, also matches at the position before the third space in the string because the space is not a word character, and the character before it is.

The engine has successfully matched the word `is` in our string, skipping the two earlier occurrences of the characters `i` and `s`. If we had used the regular expression `is`, it would have matched the `is` in `This`.

Tcl Word Boundaries

Word boundaries, as described above, are supported by most regular expression flavors. Notable exceptions are the POSIX and XML Schema flavors, which don't support word boundaries at all. Tcl uses a different syntax.

In Tcl, `\b` matches a backspace character, just like `\x08` in most regex flavors (including Tcl's). `\B` matches a single backslash character in Tcl, just like `\\` in all other regex flavors (and Tcl too).

Tcl uses the letter “y” instead of the letter “b” to match word boundaries. `\y` matches at any word boundary position, while `\Y` matches at any position that is not a word boundary. These Tcl regex tokens match exactly the same as `\b` and `\B` in Perl-style regex flavors. They don't discriminate between the start and the end of a word.

Tcl has two more word boundary tokens that do discriminate between the start and end of a word. `\m` matches only at the start of a word. That is, it matches at any position that has a non-word character to the left of it, and a word character to the right of it. It also matches at the start of the string if the first character in the string is a word character. `\M` matches only at the end of a word. It matches at any position that has a word character to the left of it, and a non-word character to the right of it. It also matches at the end of the string if the last character in the string is a word character.

The only regex engine that supports Tcl-style word boundaries (besides Tcl itself) is the JGsoft engine. In PowerGREP and EditPad Pro, `\b` and `\B` are Perl-style word boundaries, while `\y`, `\Y`, `\m` and `\M` are Tcl-style word boundaries.

In most situations, the lack of `\m` and `\M` tokens is not a problem. `\yword\y` finds “whole words only” occurrences of “word” just like `\mword\M` would. `\Mword\m` could never match anywhere, since `\M` never matches at a position followed by a word character, and `\m` never at a position preceded by one. If your regular expression needs to match characters before or after `\y`, you can easily specify in the regex whether these characters should be word characters or non-word characters. If you want to match any word, `\y\w+\y` gives the same result as `\m.+ \M`. Using `\w` instead of the dot automatically restricts the first `\y` to the start of a word, and the second `\y` to the end of a word. Note that `\y.+ \y` would not work. This regex matches each word, and also each sequence of non-word characters between the words in your subject string. That said, if your flavor supports `\m` and `\M`, the regex engine could apply `\m\w+ \M` slightly faster than `\y\w+\y`, depending on its internal optimizations.

If your regex flavor supports lookahead and lookbehind, you can use `(?<!\w)(?=\w)` to emulate Tcl's `\m` and `(?<=\w)(?! \w)` to emulate `\M`. Though quite a bit more verbose, these lookaround constructs match exactly the same as Tcl's word boundaries.

If your flavor has lookahead but not lookbehind, and also has Perl-style word boundaries, you can use `\b(?=\w)` to emulate Tcl's `\m` and `\b(?! \w)` to emulate `\M`. `\b` matches at the start or end of a word, and the lookahead checks if the next character is part of a word or not. If it is we're at the start of a word. Otherwise, we're at the end of a word.

GNU Word Boundaries

The GNU extensions to POSIX regular expressions add support for the `\b` and `\B` word boundaries, as described above. GNU also uses its own syntax for start-of-word and end-of-word boundaries. `\<` matches at the start of a word, like Tcl's `\m`. `\>` matches at the end of a word, like Tcl's `\M`.

Boost also treats `\<` and `\>` as word boundaries when using the ECMAScript, extended, egrep, or awk grammar.

POSIX Word Boundaries

The POSIX standard defines `[[[:<:]]` as a start-of-word boundary, and `[[[:>:]]` as an end-of-word boundary. Though the syntax is borrowed from POSIX bracket expressions, these tokens are word boundaries that have nothing to do with and cannot be used inside character classes. Tcl and GNU also support POSIX word boundaries. PCRE supports POSIX word boundaries starting with version 8.34. Boost supports them in all its grammars.

12. Alternation with The Vertical Bar or Pipe Symbol

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

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

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

Remember That The Regex Engine Is Eager

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

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

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

Contrary to what we intended, the regex did not match the entire string. There are several solutions. One option is to take into account that the regex engine is eager, and change the order of the options. If we use `GetValue|Get|SetValue|Set`, `SetValue` is attempted before `Set`, and the engine matches the entire string. We could also combine the four options into two and use the question mark to make part of them optional: `Get(Value)?|Set(Value)?`. Because the question mark is greedy, `SetValue` is 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`.

Text-Directed Engine Returns the Longest Match

Alternation is where regex-directed and text-directed engines differ. When a text-directed engine attempts `Get|GetValue|Set|SetValue` on `SetValue`, it tries all permutations of the regex at the start of the string. It does so efficiently, without any backtracking. It sees that the regex can find a match at the start of the string, and that the matched text can be either `Set` or `SetValue`. Because the text-directed engine evaluates the regex as a whole, it has no concept of one alternative being listed before another. But it has to make a choice as to which match to return. It always returns the longest match, in this case `SetValue`.

POSIX Requires The Longest Match

The POSIX standard leaves it up to the implementation to choose a text-directed or regex-directed engine. A BRE that includes backreferences needs to be evaluated using a regex-directed engine. But a BRE without backreferences or an ERE can be evaluated using a text-directed engine. But the POSIX standard does mandate that the longest match be returned, even when a regex-directed engine is used. Such an engine cannot be eager. It has to continue trying all alternatives even after a match is found, in order to find the longest one. This can result in very poor performance when a regex contains multiple quantifiers or a combination of quantifiers and alternation, as all combinations have to be tried to find the longest match.

The Tcl and GNU flavors also work this way.

13. Optional Items

The question mark makes the preceding token in the regular expression optional. `colou?r` matches both `colour` and `color`. The question mark is called a quantifier.

You can make several tokens optional by grouping them together using parentheses, and placing the question mark after the closing parenthesis. E.g.: `Nov(ember)?` matches `Nov` and `November`.

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

You can also use curly braces to make something optional. `colou{0,1}r` is the same as `colou?r`. POSIX BRE and GNU BRE do not support either syntax. These flavors require backslashes to *give* curly braces their special meaning: `colou\{0,1\}r`.

Important Regex Concept: Greediness

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

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

The discussion about the other repetition operators has more details on greedy and lazy quantifiers.

Looking Inside The Regex Engine

Let's apply the regular expression `colou?r` to the string `The colonel likes the color green`.

The first token in the regex is the literal `c`. The first position where it matches successfully is the `c` in `colone``l`. The engine continues, and finds that `o` matches `o`, `l` matches `l` and another `o` matches `o`. Then the engine checks whether `u` matches `n`. This fails. However, the question mark tells the regex engine that failing to match `u` is acceptable. Therefore, the engine skips ahead to the next regex token: `r`. But this fails to match `n` as well. Now, the engine can only conclude that the entire regular expression cannot be matched starting at the `c` in `colone``l`. Therefore, the engine starts again trying to match `c` to the first o in `colone``l`.

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

14. Repetition with Star and Plus

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

The asterisk or star tells the engine to attempt to match the preceding token zero or more times. The plus tells the engine to attempt to match the preceding token once or more. `<[A-Za-z][A-Za-z0-9]*>` matches an HTML tag without any attributes. The angle brackets are literals. The first character class matches a letter. The second character class matches a letter or digit. The star repeats the second character class. Because we used the star, it's OK if the second character class matches nothing. So our regex will match a tag like ``. When matching `<HTML>`, the first character class will match `H`. The star will cause the second character class to be repeated three times, matching `T`, `M` and `L` with each step.

I could also have used `<[A-Za-z0-9]+>`. I did not, because this regex would match `<1>`, which is not a valid HTML tag. But this regex may be sufficient if you know the string you are searching through does not contain any such invalid tags.

Limiting Repetition

There's an additional quantifier that allows you to specify how many times a token can be repeated. The syntax is `{min,max}`, where *min* is zero or a positive integer number indicating the minimum number of matches, and *max* is an integer equal to or greater than *min* indicating the maximum number of matches. If the comma is present but *max* is omitted, the maximum number of matches is infinite. So `{0,1}` is the same as `?`, `{0,}` is the same as `*`, and `{1,}` is the same as `+`. Omitting both the comma and *max* tells the engine to repeat the token exactly *min* times.

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

Watch Out for The Greediness!

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

Most people new to regular expressions will attempt to use `<.+>`. They will be surprised when they test it on a string like `This is a first test`. You might expect the regex to match `` and when continuing after that match, ``.

But it does not. The regex will match `first`. Obviously not what we wanted. The reason is that the plus is *greedy*. That is, the plus causes the regex engine to repeat the preceding token as often as possible. Only if that causes the entire regex to fail, will the regex engine *backtrack*. That is, it will go back to the plus, make it give up the last iteration, and proceed with the remainder of the regex. Let's take a look inside the regex engine to see in detail how this works and why this causes our regex to fail. After that, I will present you with two possible solutions.

Like the plus, the star and the repetition using curly braces are greedy.

Looking Inside The Regex Engine

The first token in the regex is `<`. This is a literal. As we already know, the first place where it will match is the first `<` in the string. The next token is the dot, which matches any character except newlines. The dot is repeated by the plus. The plus is *greedy*. Therefore, the engine will repeat the dot as many times as it can. The dot matches `E`, so the regex continues to try to match the dot with the next character. `M` is matched, and the dot is repeated once more. The next character is the `>`. You should see the problem by now. The dot matches the `>`, and the engine continues repeating the dot. The dot will match all remaining characters in the string. The dot fails when the engine has reached the void after the end of the string. Only at this point does the regex engine continue with the next token: `>`.

So far, `<.+` has matched `first test` and the engine has arrived at the end of the string. `>` cannot match here. The engine remembers that the plus has repeated the dot more often than is required. (Remember that the plus *requires* the dot to match only once.) Rather than admitting failure, the engine will *backtrack*. It will reduce the repetition of the plus by one, and then continue trying the remainder of the regex.

So the match of `.+` is reduced to `EM>first tes`. The next token in the regex is still `>`. But now the next character in the string is the last `t`. Again, these cannot match, causing the engine to backtrack further. The total match so far is reduced to `first te`. But `>` still cannot match. So the engine continues backtracking until the match of `.+` is reduced to `EM>first`. Now, `>` can match the next character in the string. The last token in the regex has been matched. The engine reports that `first` has been successfully matched.

Remember that the regex engine is *eager* to return a match. It will not continue backtracking further to see if there is another possible match. It will report the first valid match it finds. Because of greediness, this is the leftmost longest match.

Laziness Instead of Greediness

The quick fix to this problem is to make the plus *lazy* instead of greedy. Lazy quantifiers are sometimes also called “ungreedy” or “reluctant”. You can do that by putting a question mark after 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 `` has been successfully matched. That’s more like it.

An Alternative to Laziness

In this case, there is a better option than making the plus lazy. We can use a greedy plus and a negated character class: `<[!>]+>`. The reason why this is better is because of the backtracking. When using the lazy plus, the engine has to backtrack for each character in the HTML tag that it is trying to match. When using

the negated character class, no backtracking occurs at all when the string contains valid HTML code. Backtracking slows down the regex engine. You will not notice the difference when doing a single search in a text editor. But you will save plenty of CPU cycles when using such a regex repeatedly in a tight loop in a script that you are writing, or perhaps in a custom syntax coloring scheme for EditPad Pro.

Only regex-directed engines backtrack. Text-directed engines don't and thus do not get the speed penalty. But they also do not support lazy quantifiers.

Repeating `\Q...\E` Escape Sequences

The `\Q...\E` sequence escapes a string of characters, matching them as literal characters. The escaped characters are treated as individual characters. If you place a quantifier after the `\E`, it will only be applied to the last character. E.g. if you apply `\Q*\d+*\E+` to `*\d+*\d+*`, the match will be `*\d+*`. Only the asterisk is repeated. Java 4 and 5 have a bug that causes the whole `\Q...E` sequence to be repeated, yielding the whole subject string as the match. This was fixed in Java 6.

15. Use Parentheses for Grouping and Capturing

By placing part of a regular expression inside round brackets or parentheses, you can group that part of the regular expression together. This allows you to apply a quantifier to the entire group or to restrict alternation to part of the regex.

Only parentheses can be used for grouping. Square brackets define a character class, and curly braces are used by a quantifier with specific limits.

Parentheses Create Numbered Capturing Groups

Besides grouping part of a regular expression together, parentheses also create a numbered capturing group. It stores the part of the string matched by the part of the regular expression inside the parentheses.

The regex `Set(Value)?` matches `Set` or `SetValue`. In the first case, the first (and only) capturing group remains empty. In the second case, the first capturing group matches `Value`.

Non-Capturing Groups

If you do not need the group to capture its match, you can optimize this regular expression into `Set(?:Value)?`. The question mark and the colon after the opening parenthesis are the syntax that creates a non-capturing group. The question mark after the opening parenthesis is unrelated to the question mark at the end of the regex. The final question mark is the quantifier that makes the previous token optional. This quantifier cannot appear after an opening parenthesis, because there is nothing to be made optional at the start of a group. Therefore, there is no ambiguity between the question mark as an operator to make a token optional and the question mark as part of the syntax for non-capturing groups, even though this may be confusing at first. There are other kinds of groups that use the `(?)` syntax in combination with other characters than the colon that are explained later in this tutorial.

`color=(?:red|green|blue)` is another regex with a non-capturing group. This regex has no quantifiers.

Regex flavors that support named capture often have an option to turn all unnamed groups into non-capturing groups.

Using Text Matched By Capturing Groups

Capturing groups make it easy to extract part of the regex match. You can reuse the text inside the regular expression via a backreference. Backreferences can also be used in replacement strings. Please check the replacement text tutorial for details.

16. Using Backreferences To Match The Same Text Again

Backreferences match the same text as previously matched by a capturing group. Suppose you want to match a pair of opening and closing HTML tags, and the text in between. By putting the opening tag into a backreference, we can reuse the name of the tag for the closing tag. Here's how: `<([A-Z][A-Z0-9]*)\b[^\>]*>.*?</\1>`. This regex contains only one pair of parentheses, which capture the string matched by `[A-Z][A-Z0-9]*`. This is the opening HTML tag. (Since HTML tags are case insensitive, this regex requires case insensitive matching.) The backreference `\1` (backslash one) references the first capturing group. `\1` matches the exact same text that was matched by the first capturing group. The `/` before it is a literal character. It is simply the forward slash in the closing HTML tag that we are trying to match.

To figure out the number of a particular backreference, scan the regular expression from left to right. Count the opening parentheses of all the numbered capturing groups. The first parenthesis starts backreference number one, the second number two, etc. Skip parentheses that are part of other syntax such as non-capturing groups. This means that non-capturing parentheses have another benefit: you can insert them into a regular expression without changing the numbers assigned to the backreferences. This can be very useful when modifying a complex regular expression.

You can reuse the same backreference more than once. `([a-c])x\1x\1` matches `axaxa`, `bxbbx` and `cxcxc`.

Most regex flavors support up to 99 capturing groups and double-digit backreferences. So `\99` is a valid backreference if your regex has 99 capturing groups.

Looking Inside The Regex Engine

Let's see how the regex engine applies the regex `<([A-Z][A-Z0-9]*)\b[^\>]*>.*?</\1>` to the string `Testing <I>bold italic</I> text`. The first token in the regex is the literal `<`. The regex engine traverses the string until it can match at the first `<` in the string. The next token is `[A-Z]`. The regex engine also takes note that it is now inside the first pair of capturing parentheses. `[A-Z]` matches `B`. The engine advances to `[A-Z0-9]` and `>`. This match fails. However, because of the star, that's perfectly fine. The position in the string remains at `>`. The word boundary `\b` matches at the `>` because it is preceded by `B`. The word boundary does not make the engine advance through the string. The position in the regex is advanced to `[^\>]`.

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

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

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

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

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

Backtracking Into Capturing Groups

You may have wondered about the word boundary `\b` in the `<([A-Z][A-Z0-9]*)\b[>]*>.*?</\1>` mentioned above. This is to make sure the regex won't match incorrectly paired tags such as `<boo>bold`. You may think that cannot happen because the capturing group matches `boo` which causes `\1` to try to match the same, and fail. That is indeed what happens. But then the regex engine backtracks.

Let's take the regex `<([A-Z][A-Z0-9]*)[>]*>.*?</\1>` without the word boundary and look inside the regex engine at the point where `\1` fails the first time. First, `.*` continues to expand until it has reached the end of the string, and `</\1>` has failed to match each time `.*` matched one more character.

Then the regex engine backtracks into the capturing group. `[A-Z0-9]*` has matched `oo`, but would just as happily match `o` or nothing at all. When backtracking, `[A-Z0-9]*` is forced to give up one character. The regex engine continues, exiting the capturing group a second time. Since `[A-Z][A-Z0-9]*` has now matched `bo`, that is what is stored into the capturing group, overwriting `boo` that was stored before. `[>]*` matches the second `o` in the opening tag. `>.*?</` matches `>bold</`. `\1` fails again.

The regex engine does all the same backtracking once more, until `[A-Z0-9]*` is forced to give up another character, causing it to match nothing, which the star allows. The capturing group now stores just `b`. `[>]*` now matches `oo`. `>.*?</` once again matches `>bold<`. `\1` now succeeds, as does `>` and an overall match is found. But not the one we wanted.

There are several solutions to this. One is to use the word boundary. When `[A-Z0-9]*` backtracks the first time, reducing the capturing group to `bo`, `\b` fails to match between `o` and `o`. This forces `[A-Z0-9]*` to backtrack again immediately. The capturing group is reduced to `b` and the word boundary fails between `b` and `o`. There are no further backtracking positions, so the whole match attempt fails.

The reason we need the word boundary is that we're using `[>]*` to skip over any attributes in the tag. If your paired tags never have any attributes, you can leave that out, and use `<([A-Z][A-Z0-9]*)>.*?</\1>`. Each time `[A-Z0-9]*` backtracks, the `>` that follows it fails to match, quickly ending the match attempt.

If you don't want the regex engine to backtrack into capturing groups, you can use an atomic group. The tutorial section on atomic grouping has all the details.

Repetition and Backreferences

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

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

Useful Example: Checking for Doubled Words

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

Parentheses and Backreferences Cannot Be Used Inside Character Classes

Parentheses cannot be used inside character classes, at least not as metacharacters. When you put a parenthesis in a character class, it is treated as a literal character. So the regex `[(a)b]` matches `a`, `b`, `(`, and `)`.

Backreferences, too, cannot be used inside a character class. The `\1` in a regex like `(a)[\1b]` is either an error or a needlessly escaped literal 1. In JavaScript it's an octal escape.

17. Backreferences to Failed Groups

The previous topic on backreferences applies to all regex flavors, except those few that don't support backreferences at all. Flavors behave differently when you start doing things that don't fit the "match the text matched by a previous capturing group" job description.

There is a difference between a backreference to a capturing group that matched nothing, and one to a capturing group that did not participate in the match at all. The regex `(q?)b\1` matches `b`. `q?` is optional and matches nothing, causing `(q?)` to successfully match and capture nothing. `b` matches `b` and `\1` successfully matches the nothing captured by the group.

In most flavors, the regex `(q)?b\1` fails to match `b`. `(q)` fails to match at all, so the group never gets to capture anything at all. Because the whole group is optional, the engine does proceed to match `b`. The engine now arrives at `\1` which references a group that did not participate in the match attempt at all. This causes the backreference to fail to match at all, mimicking the result of the group. Since there's no `?` making `\1` optional, the overall match attempt fails.

One of the few exceptions is JavaScript. According to the official ECMA standard, a backreference to a non-participating capturing group must successfully match nothing just like a backreference to a participating group that captured nothing does. In other words, in JavaScript, `(q?)b\1` and `(q)?b\1` both match `b`. XPath also works this way.

Dinkumware's implementation of `std::regex` handles backreferences like JavaScript for all its grammars that support backreferences. Boost did so too until version 1.46. As of version 1.47, Boost fails backreferences to non-participating groups when using the ECMAScript grammar, but still lets them successfully match nothing when using the basic and `grep` grammars.

Backreferences to Non-Existent Capturing Groups

Backreferences to groups that do not exist, such as `(one)\7`, are an error in most regex flavors. There are exceptions though. JavaScript treats `\1` through `\7` as octal escapes when there are fewer capturing groups in the regex than the digit after the backslash. `\8` and `\9` are an error because 8 and 9 are not valid octal digits.

Java treats backreferences to groups that don't exist as backreferences to groups that exist but never participate in the match. They are not an error, but simply never match anything.

.NET is a little more complicated. .NET supports single-digit and double-digit backreferences as well as double-digit octal escapes without a leading zero. Backreferences trump octal escapes. So `\12` is a line feed (octal 12 = decimal 10) in a regex with fewer than 12 capturing groups. It would be a backreference to the 12th group in a regex with 12 or more capturing groups. .NET does not support single-digit octal escapes. So `\7` is an error in a regex with fewer than 7 capturing groups.

Forward References

Many modern regex flavors, including JGsoft, .NET, Java, Perl, PCRE, PHP, Delphi, and Ruby allow forward references. They allow you to use a backreference to a group that appears later in the regex. Forward references are obviously only useful if they're inside a repeated group. Then there can be situations in which

the regex engine evaluates the backreference after the group has already matched. Before the group is attempted, the backreference fails like a backreference to a failed group does.

If forward references are supported, the regex `(\2two|(one))+` matches `oneonetwo`. At the start of the string, `\2` fails. Trying the other alternative, `one` is matched by the second capturing group, and subsequently by the first group. The first group is then repeated. This time, `\2` matches `one` as captured by the second group. `two` then matches `two`. With two repetitions of the first group, the regex has matched the whole subject string.

JavaScript does not support forward references, but does not treat them as an error. In JavaScript, forward references always find a zero-length match, just as backreferences to non-participating groups do in JavaScript. Because this is not particularly useful, XRegExp makes them an error. In `std::regex`, Boost, Python, Tcl, and VBScript forward references are an error.

Nested References

A nested reference is a backreference inside the capturing group that it references. Like forward references, nested references are only useful if they're inside a repeated group, as in `(\1two|(one))+`. When nested references are supported, this regex also matches `oneonetwo`. At the start of the string, `\1` fails. Trying the other alternative, `one` is matched by the second capturing group, and subsequently by the first group. The first group is then repeated. This time, `\1` matches `one` as captured by the last iteration of the first group. It doesn't matter that the regex engine has re-entered the first group. The text matched by the group was stored into the backreference when the group was previously exited. `two` then matches `two`. With two repetitions of the first group, the regex has matched the whole subject string. If you retrieve the text from the capturing groups after the match, the first group stores `one two` while the second group captured the first occurrence of `one` in the string.

The JGsoft, .NET, Java, Perl, and VBScript flavors all support nested references. PCRE does too, but had bugs with backtracking into capturing groups with nested backreferences. Instead of fixing the bugs, PCRE 8.01 worked around them by forcing capturing groups with nested references to be atomic. So in PCRE, `(\1two|(one))+` is the same as `(?>(\1two|(one)))+`. This affects languages with regex engines based on PCRE, such as PHP, Delphi, and R.

JavaScript and Ruby do not support nested references, but treat them as backreferences to non-participating groups instead of as errors. In JavaScript that means they always match a zero-length string, while in Ruby they always fail to match. In `std::regex`, Boost, Python, and Tcl, nested references are an error.

18. Named Capturing Groups and Backreferences

Nearly all modern regular expression engines support numbered capturing groups and numbered backreferences. Long regular expressions with lots of groups and backreferences may be hard to read. They can be particularly difficult to maintain as adding or removing a capturing group in the middle of the regex upsets the numbers of all the groups that follow the added or removed group.

Python’s `re` module was the first to offer a solution: named capturing groups and named backreferences. `(?P<name>group)` captures the match of `group` into the backreference “name”. `name` must be an alphanumeric sequence starting with a letter. `group` can be any regular expression. You can reference the contents of the group with the named backreference `(?P=name)`. The question mark, P, angle brackets, and equals signs are all part of the syntax. Though the syntax for the named backreference uses parentheses, it’s just a backreference that doesn’t do any capturing or grouping. The HTML tags example can be written as `<(P<tag>[A-Z][A-Z0-9]*)\b[^\>]*>. *?</(?P=tag)>`.

.NET also supports named capture. Microsoft’s developers invented their own syntax, rather than follow the one pioneered by Python and copied by PCRE (the only two regex engines that supported named capture at that time). `(?<name>group)` or `(?'name'group)` captures the match of `group` into the backreference “name”. The named backreference is `\k<name>` or `\k'name'`. Compared with Python, there is no P in the syntax for named groups. The syntax for named backreferences is more similar to that of numbered backreferences than to what Python uses. You can use single quotes or angle brackets around the name. This makes absolutely no difference in the regex. You can use both styles interchangeably. The syntax using angle brackets is preferable in programming languages that use single quotes to delimit strings, while the syntax using single quotes is preferable when adding your regex to an XML file, as this minimizes the amount of escaping you have to do to format your regex as a literal string or as XML content.

Because Python and .NET introduced their own syntax, we refer to these two variants as the “Python syntax” and the “.NET syntax” for named capture and named backreferences. Today, many other regex flavors have copied this syntax.

Perl 5.10 added support for both the Python and .NET syntax for named capture and backreferences. It also adds two more syntactic variants for named backreferences: `\k{one}` and `\g{two}`. There’s no difference between the five syntaxes for named backreferences in Perl. All can be used interchangeably. In the replacement text, you can interpolate the variable `$+{name}` to insert the text matched by a named capturing group.

PCRE 7.2 and later support all the syntax for named capture and backreferences that Perl 5.10 supports. Old versions of PCRE supported the Python syntax, even though that was not “Perl-compatible” at the time. Languages like PHP, Delphi, and R that implement their regex support using PCRE also support all this syntax. Unfortunately, neither PHP or R support named references in the replacement text. You’ll have to use numbered references to the named groups. PCRE does not support search-and-replace at all.

Java 7 and XRegExp copied the .NET syntax, but only the variant with angle brackets. Ruby 1.9 and supports both variants of the .NET syntax. The JGsoft flavor supports the Python syntax and both variants of the .NET syntax.

Boost 1.42 and later support named capturing groups using the .NET syntax with angle brackets or quotes and named backreferences using the `\g` syntax with curly braces from Perl 5.10. Boost 1.47 additionally supports backreferences using the `\k` syntax with angle brackets and quotes from .NET. Boost 1.47 allowed these variants to multiply. Boost 1.47 allows named and numbered backreferences to be specified with `\g` or

`\k` and with curly braces, angle brackets, or quotes. So Boost 1.47 and later have six variations of the backreference syntax on top of the basic `\1` syntax. This puts Boost in conflict with Ruby, PCRE, PHP, R, and JGsoft which treat `\g` with angle brackets or quotes as a subroutine call.

Numbers for Named Capturing Groups

Mixing named and numbered capturing groups is not recommended because flavors are inconsistent in how the groups are numbered. If a group doesn't need to have a name, make it non-capturing using the `(?:group)` syntax. In .NET you can make all unnamed groups non-capturing by setting `RegexOptions.ExplicitCapture`. In Delphi, set `roExplicitCapture`. With `XRegExp`, use the `/n` flag. Perl supports `/n` starting with Perl 5.22. With PCRE, set `PCRE_NO_AUTO_CAPTURE`. The JGsoft flavor and .NET support the `(?n)` mode modifier. If you make all unnamed groups non-capturing, you can skip this section and save yourself a headache.

Most flavors number both named and unnamed capturing groups by counting their opening parentheses from left to right. Adding a named capturing group to an existing regex still upsets the numbers of the unnamed groups. In .NET, however, unnamed capturing groups are assigned numbers first, counting their opening parentheses from left to right, skipping all named groups. After that, named groups are assigned the numbers that follow by counting the opening parentheses of the named groups from left to right.

The JGsoft regex engine copied the Python and the .NET syntax at a time when only Python and PCRE used the Python syntax, and only .NET used the .NET syntax. Therefore it also copied the numbering behavior of both Python and .NET, so that regexes intended for Python and .NET would keep their behavior. It numbers Python-style named groups along unnamed ones, like Python does. It numbers .NET-style named groups afterward, like .NET does. These rules apply even when you mix both styles in the same regex.

As an example, the regex `(a)(?P<x>b)(c)(?P<y>d)` matches `abcd` as expected. If you do a search-and-replace with this regex and the replacement `\1\2\3\4` or `$1$2$3$4` (depending on the flavor), you will get `abcd`. All four groups were numbered from left to right, from one till four.

Things are a bit more complicated with .NET. The regex `(a)(?<x>b)(c)(?<y>d)` again matches `abcd`. However, if you do a search-and-replace with `$1$2$3$4` as the replacement, you will get `acbd`. First, the unnamed groups `(a)` and `(c)` got the numbers 1 and 2. Then the named groups "x" and "y" got the numbers 3 and 4.

In all other flavors that copied the .NET syntax the regex `(a)(?<x>b)(c)(?<y>d)` still matches `abcd`. But in all those flavors, except the JGsoft flavor, the replacement `\1\2\3\4` or `$1$2$3$4` (depending on the flavor) gets you `abcd`. All four groups were numbered from left to right.

In PowerGREP, which uses the JGsoft flavor, named capturing groups play a special role. Groups with the same name are shared between all regular expressions and replacement texts in the same PowerGREP action. This allows captured by a named capturing group in one part of the action to be referenced in a later part of the action. Because of this, PowerGREP does not allow numbered references to named capturing groups at all. When mixing named and numbered groups in a regex, the numbered groups are still numbered following the Python and .NET rules, like the JGsoft flavor always does.

Multiple Groups with The Same Name

The .NET framework and the JGsoft flavor allow multiple groups in the regular expression to have the same name. All groups with the same name share the same storage for the text they match. Thus, a backreference to that name matches the text that was matched by the group with that name that most recently captured something. A reference to the name in the replacement text inserts the text matched by the group with that name that was the last one to capture something.

Perl and Ruby also allow groups with the same name. But these flavors only use smoke and mirrors to make it look like the all the groups with the same name act as one. In reality, the groups are separate. In Perl, a backreference matches the text captured by the leftmost group in the regex with that name that matched something. In Ruby, a backreference matches the text captured by any of the groups with that name. Backtracking makes Ruby try all the groups.

So in Perl and Ruby, you can only meaningfully use groups with the same name if they are in separate alternatives in the regex, so that only one of the groups with that name could ever capture any text. Then backreferences to that group sensibly match the text captured by the group.

For example, if you want to match “a” followed by a digit 0..5, or “b” followed by a digit 4..7, and you only care about the digit, you could use the regex `a(?<digit>[0-5])|b(?<digit>[4-7])`. In these four flavors, the group named “digit” will then give you the digit 0..7 that was matched, regardless of the letter. If you want this match to be followed by c and the exact same digit, you could use `(?:a(?<digit>[0-5])|b(?<digit>[4-7]))c\k<digit>`

PCRE does not allow duplicate named groups by default. PCRE 6.7 and later allow them if you turn on that option or use the mode modifier `(?J)`. But prior to PCRE 8.36 that wasn’t very useful as backreferences always pointed to the first capturing group with that name in the regex regardless of whether it participated in the match. Starting with PCRE 8.36 (and thus PHP 5.6.9 and R 3.1.3) and also in PCRE2, backreferences point to the first group with that name that actually participated in the match. Though PCRE and Perl handle duplicate groups in opposite directions the end result is the same if you follow the advice to only use groups with the same name in separate alternatives.

Boost allows duplicate named groups. Prior to Boost 1.47 that wasn’t useful as backreferences always pointed to the last group with that name that appears before the backreference in the regex. In Boost 1.47 and later backreferences point to the first group with that name that actually participated in the match just like in PCRE 8.36 and later.

Python, Java, and XRegExp 3 do not allow multiple groups to use the same name. Doing so will give a regex compilation error. XRegExp 2 allowed them, but did not handle them correctly.

In Perl 5.10, PCRE 8.00, PHP 5.2.14, and Boost 1.42 (or later versions of these) it is best to use a branch reset group when you want groups in different alternatives to have the same name, as in `(?|a(?<digit>[0-5])|b(?<digit>[4-7]))c\k<digit>`. With this special syntax—group opened with `(?|` instead of `(?:`—the two groups named “digit” really are one and the same group. Then backreferences to that group are always handled correctly and consistently between these flavors. (Older versions of PCRE and PHP may support branch reset groups, but don’t correctly handle duplicate names in branch reset groups.)

19. Relative Backreferences

Some applications support relative backreferences. These use a negative number to reference a group preceding the backreference. To find the group that the relative backreference refers to, take the absolute number of the backreference and count that many opening parentheses of (named or unnamed) capturing groups starting at the backreference and going from right to left through the regex. So `(a)(b)(c)\k<-1>` matches `abcc` and `(a)(b)(c)\k<-3>` matches `abca`. If the backreference is inside a capturing group, then you also need to count that capturing group's opening parenthesis. So `(a)(b)(c\k<-2>)` matches `abcb`. `(a)(b)(c\k<-1>)` either fails to match or is an error depending on whether your application allows nested backreferences.

The syntax for nested backreferences varies widely. It is generally an extension of the syntax for named backreferences. JGsoft V2 and Ruby 1.9 and later support `\k<-1>` and `\k'-1'`. Though this looks like the .NET syntax for named capture, .NET itself does not support relative backreferences.

Perl 5.10, PCRE 7.0, PHP 5.2.2, and R support `\g{-1}` and `\g-1`.

Boost supports the Perl syntax starting with Boost 1.42. Boost adds the Ruby syntax starting with Boost 1.47. To complicate matters, Boost 1.47 allowed these variants to multiply. Boost 1.47 and later allow relative backreferences to be specified with `\g` or `\k` and with curly braces, angle brackets, or quotes. That makes six variations plus `\g-1` for a total of seven variations. This puts Boost in conflict with Ruby, PCRE, PHP, R, and JGsoft which treat `\g` with angle brackets or quotes and a negative number as a relative subroutine call.

20. Branch Reset Groups

Perl 5.10 introduced a new regular expression feature called a branch reset group. JGsoft V2 and PCRE 7.2 and later also support this, as do languages like PHP, Delphi, and R that have regex functions based on PCRE. Boost added them to its ECMAScript grammar in version 1.42.

Alternatives inside a branch reset group share the same capturing groups. The syntax is `(?regex)` where `(?)` opens the group and `regex` is any regular expression. If you don't use any alternation or capturing groups inside the branch reset group, then its special function doesn't come into play. It then acts as a non-capturing group.

The regex `(?(a)|(b)|(c))` consists of a single branch reset group with three alternatives. This regex matches either `a`, `b`, or `c`. The regex has only a single capturing group with number 1 that is shared by all three alternatives. After the match, `$1` holds `a`, `b`, or `c`.

Compare this with the regex `(a)|(b)|(c)` that lacks the branch reset group. This regex also matches `a`, `b`, or `c`. But it has three capturing groups. After the match, `$1` holds `a` or nothing at all, `$2` holds `b` or nothing at all, while `$3` holds `c` or nothing at all.

Backreferences to capturing groups inside branch reset groups work like you'd expect. `(?(a)|(b)|(c))\1` matches `aa`, `bb`, or `cc`. Since only one of the alternatives inside the branch reset group can match, the alternative that participates in the match determines the text stored by the capturing group and thus the text matched by the backreference.

The alternatives in the branch reset group don't need to have the same number of capturing groups. `(?abc|(d)(e)(f)|g(h)i)` has three capturing groups. When this regex matches `abc`, all three groups are empty. When `def` is matched, `$1` holds `d`, `$2` holds `e` and `$3` holds `f`. When `ghi` is matched, `$1` holds `h` while the other two are empty.

You can have capturing groups before and after the branch reset group. Groups before the branch reset group are numbered as usual. Groups in the branch reset group are numbered continued from the groups before the branch reset group, which each alternative resetting the number. Groups after the branch reset group are numbered continued from the alternative with the most groups, even if that is not the last alternative. So `(x)?(abc|(d)(e)(f)|g(h)i)(y)` defines five capturing groups. `(x)` is group 1, `(d)` and `(h)` are group 2, `(e)` is group 3, `(f)` is group 4, and `(y)` is group 5.

Named Capturing Groups in Branch Reset Groups

You can use named capturing groups inside branch reset groups. If you do, you should use the same names for the groups that will get the same numbers. Otherwise you'll get undesirable behavior in Perl or Boost. PowerGREP treats mismatched group names as an error. PCRE only reliably supports named groups inside branch reset groups starting with version 8.00. This means Delphi only does so starting with XE7 and PHP starting with version 5.2.14.

`(?'before'x)?(?'left'd)(?'middle'e)(?'right'f)|g(?'left'hi)(?'after'y)` is the same as the previous regex. It names the five groups "before", "left", "middle", "right", and "after". Notice that because the 3rd alternative has only one capturing group, that must be the name of the first group in the other alternatives.

If you omit the names in some alternatives, the groups will still share the names with the other alternatives. In the regex `(?'before'x)(?'|abc|(?'left'd)(?'middle'e)(?'right'f)|g(h)i)(?'after'y)` the group `(h)` is still named “left” because the branch reset group makes it share the name and number of `(?'left'd)`.

In Perl, PCRE, and Boost, it is best to use a branch reset group when you want groups in different alternatives to have the same name. That’s the only way in Perl, PCRE, and Boost to make sure that groups with the same name really are one and the same group.

In PowerGREP, groups with the same name are always treated as one and the same group. So you don’t really need to use a branch reset group in PowerGREP when using named capturing groups.

Day and Month with Accurate Number of Days

It’s time for a more practical example. These two regular expressions match a date in m/d or mm/dd format. They exclude invalid dates such as 2/31.

```
^(?:|(0?[13578]|1[02])/|(3[01]|1[2][0-9]|0?[1-9]) # 31 days
| (0?[469]|11)/(30|[12][0-9]|0?[1-9]) # 30 days
| (0?2)/([12][0-9]|0?[1-9]) # 29 days
)$
```

The first version uses a non-capturing group `(?:...)` to group the alternatives. It has six separate capturing groups. `$1` and `$2` hold the month and the day for months with 31 days. `$3` and `$4` hold them for months with 30 days. `$5` and `$6` are only used for February.

```
^(?|(0?[13578]|1[02])/|(3[01]|1[2][0-9]|0?[1-9]) # 31 days
| (0?[469]|11)/(30|[12][0-9]|0?[1-9]) # 30 days
| (0?2)/([12][0-9]|0?[1-9]) # 29 days
)$
```

The second version uses a branch reset group `(?|...)` to group the alternatives and merge their capturing groups. The 4th character is the only difference between these two regexes. Now there are only two capturing groups. These are shared between the three alternatives. When a match is found `$1` always holds the month and `$2` always holds the day, regardless of the number of days in the month.

21. Free-Spacing Regular Expressions

Most modern regex flavors support a variant of the regular expression syntax called free-spacing mode. This mode allows for regular expressions that are much easier for people to read. Of the flavors discussed in this tutorial, only XML Schema and the POSIX and GNU flavors don't support it. Plain JavaScript doesn't either, but XRegExp does. The mode is usually enabled by setting an option or flag outside the regex. With flavors that support mode modifiers, you can put `(?x)` the very start of the regex to make the remainder of the regex free-spacing.

In free-spacing mode, whitespace between regular expression tokens is ignored. Whitespace includes spaces, tabs, and line breaks. Note that only whitespace *between* tokens is ignored. `a b c` is the same as `abc` in free-spacing mode. But `\ d` and `\d` are not the same. The former matches `d`, while the latter matches a digit. `\d` is a single regex token composed of a backslash and a "d". Breaking up the token with a space gives you an escaped space (which matches a space), and a literal "d".

Likewise, grouping modifiers cannot be broken up. `(?>atomic)` is the same as `(?> ato mic)` and as `(?>ato mic)`. They all match the same atomic group. They're not the same as `(? >atomic)`. The latter is a syntax error. The `?>` grouping modifier is a single element in the regex syntax, and must stay together. This is true for all such constructs, including lookahead, named groups, etc.

Exactly which spaces and line breaks are ignored depends on the regex flavor. All flavors discussed in this tutorial ignore the ASCII space, tab, line feed, carriage return, and form feed characters. JGsoft V2 and Boost are the only flavors that ignore all Unicode spaces and line breaks. JGsoft V1 almost does but misses the next line control character (U+0085). Perl always treats non-ASCII spaces as literals. Perl 5.22 and later ignore non-ASCII line breaks. Perl 5.16 and prior treat them as literals. Perl 5.18 and 5.20 treated unescaped non-ASCII line breaks as errors in free-spacing mode to give developers a transition period.

Free-Spacing in Character Classes

A character class is generally treated as a single token. `[abc]` is not the same as `[a b c]`. The former matches one of three letters, while the latter matches those three letters or a space. In other words: free-spacing mode has no effect inside character classes. Spaces and line breaks inside character classes will be included in the character class. This means that in free-spacing mode, you can use `\` or `[]` to match a single space. Use whichever you find more readable. The hexadecimal escape `\x20` also works, of course.

Java, however, does not treat a character class as a single token in free-spacing mode. Java does ignore spaces, line breaks, and comments inside character classes. So in Java's free-spacing mode, `[abc]` is identical to `[a b c]`. To add a space to a character class, you'll have to escape it with a backslash. But even in free-spacing mode, the negating caret must appear immediately after the opening bracket. `[^ a b c]` matches any of the four characters `^`, `a`, `b` or `c` just like `[abc^]` would. With the negating caret in the proper place, `[^ a b c]` matches any character that is not `a`, `b` or `c`.

Perl 5.26 offers limited free-spacing within character classes as an option. The `/x` flag enables free-spacing outside character classes only, as in previous versions of Perl. The double `/xx` flag additionally makes Perl 5.26 treat unescaped spaces and tabs inside character classes as free whitespace. Line breaks are still literals inside character classes. PCRE2 10.30 supports the same `/xx` mode as Perl 5.26 if you pass the flag `PCRE2_EXTENDED_MORE` to `pcre2_compile()`.

Perl 5.26 and PCRE 10.30 also add a new mode modifier `(?xx)` which enables free-spacing both inside and outside character classes. `(?x)` turns on free-spacing outside character classes like before, but also turns off free-spacing inside character classes. `(?-x)` and `(?-xx)` both completely turn off free-spacing.

Java treats the `^` in `[^ a]` as a literal. Even when spaces are ignored they still break the special meaning of the caret in Java. Perl 5.26 and PCRE2 10.30 treat `^` in `[^ a]` as a negation caret in `/xx` mode. Perl 5.26 and PCRE2 10.30 totally ignore free whitespace. They still consider the caret to be at the start of the character class.

Comments in Free-Spacing Mode

Another feature of free-spacing mode is that the `#` character starts a comment. The comment runs until the end of the line. Everything from the `#` until the next newline character is ignored. Most flavors do not recognize any other line break characters as the end of a comment, even if they recognize other line breaks as free whitespace or allow anchors to match at other line breaks. JGsoft V2 is the only flavor that recognizes all Unicode line breaks. Boost misses the vertical tab.

XPath and Oracle do not support comments within the regular expression, even though they have a free-spacing mode. They always treat `#` as a literal character.

Java is the only flavor that treats `#` as the start of a comment inside character classes in free-spacing mode. The comment runs until the end of the line, so you can use a `]` to close a comment. All other flavors treat `#` as a literal inside character classes. That includes Perl 5.26 in `/xx` mode.

Putting it all together, the regex to match a valid date can be clarified by writing it across multiple lines:

```
# Match a 20th or 21st century date in yyyy-mm-dd format
((?:19|20)\d\d)      # year (group 1)
[- /.]              # separator
(0[1-9]|1[012])    # month (group 2)
[- /.]              # separator
(0[1-9]|1[12][0-9]|3[01]) # day (group 3)
```

The screenshot shows the RegxBuddy application window. The main text area contains the following regex pattern with comments:

```
# Match a 20th or 21st century date in yyyy-mm-dd format
(?:19|20)\d\d # year (group 1)
[-./] # separator
(?:0[1-9]|1[012]) # month (group 2)
[-./] # separator
(?:0[1-9]|12|0[0-9])3[01] # day (group 3)
```

The application interface includes a menu bar with options like Match, Replace, Split, Copy, Paste, and a toolbar with buttons for Create, Convert, Test, Debug, Use, Library, GREP, and Forum. The main area shows a detailed explanation of the regex components, including comments and token descriptions.

Comments Without Free-Spacing

Many flavors also allow you to add comments to your regex without using free-spacing mode. The syntax is `(?#comment)` where “comment” can be whatever you want, as long as it does not contain a closing parenthesis. The regex engine ignores everything after the `(?#` until the first closing parenthesis.

Of the flavors discussed in this tutorial, all flavors that support comment in free-spacing mode, except Java and Tcl, also support `(?#comment)`. The flavors that don’t support comments in free-spacing mode or don’t support free-spacing mode at all don’t support `(?#comment)` either.

22. Unicode Regular Expressions

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

Unfortunately, Unicode brings its own requirements and pitfalls when it comes to regular expressions. Of the regex flavors discussed in this tutorial, Java, XML and .NET use Unicode-based regex engines. Perl supports Unicode starting with version 5.6. PCRE can optionally be compiled with Unicode support. Note that PCRE is far less flexible in what it allows for the `\p` tokens, despite its name “Perl-compatible”. The PHP `preg` functions, which are based on PCRE, support Unicode when the `/u` option is appended to the regular expression. Ruby supports Unicode escapes and properties in regular expressions starting with version 1.9. `XRegExp` brings support for Unicode properties to JavaScript.

RegexBuddy’s regex engine is fully Unicode-based starting with version 2.0.0. RegexBuddy 1.x.x did not support Unicode at all. PowerGREP uses the same Unicode regex engine starting with version 3.0.0. Earlier versions would convert Unicode files to ANSI prior to grepping with an 8-bit (i.e. non-Unicode) regex engine. EditPad Pro supports Unicode starting with version 6.0.0.

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

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

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

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

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

How to Match a Single Unicode Grapheme

Matching a single grapheme, whether it’s encoded as a single code point, or as multiple code points using combining marks, is easy in Perl, PCRE, PHP, Boost, Ruby 2.0, Java 9, and the Just Great Software applications: simply use `\X`. You can consider `\X` the Unicode version of the dot. There is one difference,

though: `\X` always matches line break characters, whereas the dot does not match line break characters unless you enable the dot matches newline matching mode.

In .NET, Java 8 and prior, and Ruby 1.9 you can use `\P{M}\p{M}*` or `(?>\P{M}\p{M}*)` as a reasonably close substitute. To match any number of graphemes, use `(?>\P{M}\p{M}*)+` as a substitute for `\X+`.

Matching a Specific Code Point

To match a specific Unicode code point, use `\uFFFF` where FFFF is the hexadecimal number of the code point you want to match. You must always specify 4 hexadecimal digits. E.g. `\u00E0` matches `à`, but only when encoded as a single code point U+00E0.

Perl, PCRE, Boost, and `std::regex` do not support the `\uFFFF` syntax. They use `\x{FFFF}` instead. You can omit leading zeros in the hexadecimal number between the curly braces. Since `\x` by itself is not a valid regex token, `\x{1234}` can never be confused to match `\x` 1234 times. It always matches the Unicode code point U+1234. `\x{1234}{5678}` will try to match code point U+1234 exactly 5678 times.

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

JavaScript, which does not offer any Unicode support through its `RegExp` class, does support `\uFFFF` for matching a single Unicode code point as part of its string syntax.

XML Schema and XPath do not have a regex token for matching Unicode code points. However, you can easily use XML entities like `&#xXXXX`; to insert literal code points into your regular expression.

Unicode Categories

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

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

You should now understand why `\P{M}\p{M}*` is the equivalent of `\X`. `\P{M}` matches a code point that is not a combining mark, while `\p{M}*` matches zero or more code points that are combining marks. To match a letter including any diacritics, use `\p{L}\p{M}*`. This last regex will always match `à`, regardless of how it is encoded. The possessive quantifier makes sure that backtracking doesn't cause `\P{M}\p{M}*` to match a non-mark without the combining marks that follow it, which `\X` would never do.

PCRE, PHP, and .NET are case sensitive when it checks the part between curly braces of a `\p` token. `\p{Zs}` will match any kind of space character, while `\p{zs}` will throw an error. All other regex engines described in this tutorial will match the space in both cases, ignoring the case of the category between the curly braces. Still, I recommend you make a habit of using the same uppercase and lowercase combination as I did in the list of properties below. This will make your regular expressions work with all Unicode regex engines.

In addition to the standard notation, `\p{L}`, Java, Perl, PCRE, the JGsoft engine, and XRegExp 3 allow you to use the shorthand `\pL`. The shorthand only works with single-letter Unicode properties. `\pLl` is *not* the equivalent of `\p{Ll}`. It is the equivalent of `\p{L}l` which matches `Al` or `àl` or any Unicode letter followed by a literal `l`.

Perl, XRegExp, and the JGsoft engine also support the longhand `\p{Letter}`. You can find a complete list of all Unicode properties below. You may omit the underscores or use hyphens or spaces instead.

- `\p{L}` or `\p{Letter}`: any kind of letter from any language.
 - `\p{Ll}` or `\p{Lowercase_Letter}`: a lowercase letter that has an uppercase variant.
 - `\p{Lu}` or `\p{Uppercase_Letter}`: an uppercase letter that has a lowercase variant.
 - `\p{Lt}` or `\p{Titlecase_Letter}`: a letter that appears at the start of a word when only the first letter of the word is capitalized.
 - `\p{L&}` or `\p{Cased_Letter}`: a letter that exists in lowercase and uppercase variants (combination of Ll, Lu and Lt).
 - `\p{Lm}` or `\p{Modifier_Letter}`: a special character that is used like a letter.
 - `\p{Lo}` or `\p{Other_Letter}`: a letter or ideograph that does not have lowercase and uppercase variants.
- `\p{M}` or `\p{Mark}`: a character intended to be combined with another character (e.g. accents, umlauts, enclosing boxes, etc.).
 - `\p{Mn}` or `\p{Non_Spacing_Mark}`: a character intended to be combined with another character without taking up extra space (e.g. accents, umlauts, etc.).
 - `\p{Mc}` or `\p{Spacing_Combining_Mark}`: a character intended to be combined with another character that takes up extra space (vowel signs in many Eastern languages).
 - `\p{Me}` or `\p{Enclosing_Mark}`: a character that encloses the character it is combined with (circle, square, keycap, etc.).
- `\p{Z}` or `\p{Separator}`: any kind of whitespace or invisible separator.
 - `\p{Zs}` or `\p{Space_Separator}`: a whitespace character that is invisible, but does take up space.
 - `\p{Zl}` or `\p{Line_Separator}`: line separator character U+2028.
 - `\p{Zp}` or `\p{Paragraph_Separator}`: paragraph separator character U+2029.
- `\p{S}` or `\p{Symbol}`: math symbols, currency signs, dingbats, box-drawing characters, etc.
 - `\p{Sm}` or `\p{Math_Symbol}`: any mathematical symbol.
 - `\p{Sc}` or `\p{Currency_Symbol}`: any currency sign.
 - `\p{Sk}` or `\p{Modifier_Symbol}`: a combining character (mark) as a full character on its own.
 - `\p{So}` or `\p{Other_Symbol}`: various symbols that are not math symbols, currency signs, or combining characters.
- `\p{N}` or `\p{Number}`: any kind of numeric character in any script.
 - `\p{Nd}` or `\p{Decimal_Digit_Number}`: a digit zero through nine in any script except ideographic scripts.
 - `\p{Nl}` or `\p{Letter_Number}`: a number that looks like a letter, such as a Roman numeral.

- `\p{No}` or `\p{Other_Number}`: a superscript or subscript digit, or a number that is not a digit 0–9 (excluding numbers from ideographic scripts).
- `\p{P}` or `\p{Punctuation}`: any kind of punctuation character.
 - `\p{Pd}` or `\p{Dash_Punctuation}`: any kind of hyphen or dash.
 - `\p{Ps}` or `\p{Open_Punctuation}`: any kind of opening bracket.
 - `\p{Pe}` or `\p{Close_Punctuation}`: any kind of closing bracket.
 - `\p{Pi}` or `\p{Initial_Punctuation}`: any kind of opening quote.
 - `\p{Pf}` or `\p{Final_Punctuation}`: any kind of closing quote.
 - `\p{Pc}` or `\p{Connector_Punctuation}`: a punctuation character such as an underscore that connects words.
 - `\p{Po}` or `\p{Other_Punctuation}`: any kind of punctuation character that is not a dash, bracket, quote or connector.
- `\p{C}` or `\p{Other}`: invisible control characters and unused code points.
 - `\p{Cc}` or `\p{Control}`: an ASCII or Latin-1 control character: 0x00–0x1F and 0x7F–0x9F.
 - `\p{Cf}` or `\p{Format}`: invisible formatting indicator.
 - `\p{Co}` or `\p{Private_Use}`: any code point reserved for private use.
 - `\p{Cs}` or `\p{Surrogate}`: one half of a surrogate pair in UTF-16 encoding.
 - `\p{Cn}` or `\p{Unassigned}`: any code point to which no character has been assigned.

Unicode Scripts

The Unicode standard places each assigned code point (character) into one script. A script is a group of code points used by a particular human writing system. Some scripts like Thai correspond with a single human language. Other scripts like Latin span multiple languages.

Some languages are composed of multiple scripts. There is no Japanese Unicode script. Instead, Unicode offers the Hiragana, Katakana, Han, and Latin scripts that Japanese documents are usually composed of.

A special script is the Common script. This script contains all sorts of characters that are common to a wide range of scripts. It includes all sorts of punctuation, whitespace and miscellaneous symbols.

All assigned Unicode code points (those matched by `\p{Cn}`) are part of exactly one Unicode script. All unassigned Unicode code points (those matched by `\p{Cn}`) are not part of any Unicode script at all.

The JGsoft engine, Perl, PCRE, PHP, Ruby 1.9, Delphi, and XRegExp can match Unicode scripts. Here’s a list:

1. `\p{Common}`
2. `\p{Arabic}`
3. `\p{Armenian}`
4. `\p{Bengali}`
5. `\p{Bopomofo}`
6. `\p{Braille}`
7. `\p{Buhid}`
8. `\p{Canadian_Aboriginal}`
9. `\p{Cherokee}`
10. `\p{Cyrillic}`
11. `\p{Devanagari}`
12. `\p{Ethiopic}`

13. `\p{Georgian}`
14. `\p{Greek}`
15. `\p{Gujarati}`
16. `\p{Gurmukhi}`
17. `\p{Han}`
18. `\p{Hangul}`
19. `\p{Hanunoo}`
20. `\p{Hebrew}`
21. `\p{Hiragana}`
22. `\p{Inherited}`
23. `\p{Kannada}`
24. `\p{Katakana}`
25. `\p{Khmer}`
26. `\p{Lao}`
27. `\p{Latin}`
28. `\p{Limbu}`
29. `\p{Malayalam}`
30. `\p{Mongolian}`
31. `\p{Myanmar}`
32. `\p{Ogham}`
33. `\p{Oriya}`
34. `\p{Runic}`
35. `\p{Sinhala}`
36. `\p{Syriac}`
37. `\p{Tagalog}`
38. `\p{Tagbanwa}`
39. `\p{TaiLe}`
40. `\p{Tamil}`
41. `\p{Telugu}`
42. `\p{Thaana}`
43. `\p{Thai}`
44. `\p{Tibetan}`
45. `\p{Yi}`

Perl and the JGsoft flavor allow you to use `\p{IsLatin}` instead of `\p{Latin}`. The “Is” syntax is useful for distinguishing between scripts and blocks, as explained in the next section. PCRE, PHP, and XRegExp do not support the “Is” prefix.

Java 7 adds support for Unicode scripts. Unlike the other flavors, Java 7 requires the “Is” prefix.

Unicode Blocks

The Unicode standard divides the Unicode character map into different blocks or ranges of code points. Each block is used to define characters of a particular script like “Tibetan” or belonging to a particular group like “Braille Patterns”. Most blocks include unassigned code points, reserved for future expansion of the Unicode standard.

Note that Unicode blocks do not correspond 100% with scripts. An essential difference between blocks and scripts is that a block is a single contiguous range of code points, as listed below. Scripts consist of characters taken from all over the Unicode character map. Blocks may include unassigned code points (i.e. code points matched by `\p{Cn}`). Scripts never include unassigned code points. Generally, if you’re not sure whether to use a Unicode script or Unicode block, use the script.

For example, the Currency block does not include the dollar and yen symbols. Those are found in the Basic_Latin and Latin-1_Supplement blocks instead, even though both are currency symbols, and the yen symbol is not a Latin character. This is for historical reasons, because the ASCII standard includes the dollar sign, and the ISO-8859 standard includes the yen sign. You should not blindly use any of the blocks listed below based on their names. Instead, look at the ranges of characters they actually match. A tool like RegexpBuddy can be very helpful with this. The Unicode property `\p{Sc}` or `\p{Currency_Symbol}` would be a better choice than the Unicode block `\p{InCurrency_Symbols}` when trying to find all currency symbols.

1. `\p{InBasic_Latin}`: U+0000–U+007F
2. `\p{InLatin-1_Supplement}`: U+0080–U+00FF
3. `\p{InLatin_Extended-A}`: U+0100–U+017F
4. `\p{InLatin_Extended-B}`: U+0180–U+024F
5. `\p{InIPA_Extensions}`: U+0250–U+02AF
6. `\p{InSpacing_Modifier_Letters}`: U+02B0–U+02FF
7. `\p{InCombining_Diacritical_Marks}`: U+0300–U+036F
8. `\p{InGreek_and_Coptic}`: U+0370–U+03FF
9. `\p{InCyrillic}`: U+0400–U+04FF
10. `\p{InCyrillic_Supplementary}`: U+0500–U+052F
11. `\p{InArmenian}`: U+0530–U+058F
12. `\p{InHebrew}`: U+0590–U+05FF
13. `\p{InArabic}`: U+0600–U+06FF
14. `\p{InSyriac}`: U+0700–U+074F
15. `\p{InThaana}`: U+0780–U+07BF
16. `\p{InDevanagari}`: U+0900–U+097F
17. `\p{InBengali}`: U+0980–U+09FF
18. `\p{InGurmukhi}`: U+0A00–U+0A7F
19. `\p{InGujarati}`: U+0A80–U+0AFF
20. `\p{InOriya}`: U+0B00–U+0B7F
21. `\p{InTamil}`: U+0B80–U+0BFF
22. `\p{InTelugu}`: U+0C00–U+0C7F
23. `\p{InKannada}`: U+0C80–U+0CFF
24. `\p{InMalayalam}`: U+0D00–U+0D7F
25. `\p{InSinhala}`: U+0D80–U+0DFF
26. `\p{InThai}`: U+0E00–U+0E7F
27. `\p{InLao}`: U+0E80–U+0EFF
28. `\p{InTibetan}`: U+0F00–U+0FFF
29. `\p{InMyanmar}`: U+1000–U+109F
30. `\p{InGeorgian}`: U+10A0–U+10FF
31. `\p{InHangul_Jamo}`: U+1100–U+11FF
32. `\p{InEthiopic}`: U+1200–U+137F
33. `\p{InCherokee}`: U+13A0–U+13FF
34. `\p{InUnified_Canadian_Aboriginal_Syllabics}`: U+1400–U+167F
35. `\p{InOgham}`: U+1680–U+169F
36. `\p{InRunic}`: U+16A0–U+16FF
37. `\p{InTagalog}`: U+1700–U+171F
38. `\p{InHanunoo}`: U+1720–U+173F
39. `\p{InBuhid}`: U+1740–U+175F
40. `\p{InTagbanwa}`: U+1760–U+177F
41. `\p{InKhmer}`: U+1780–U+17FF
42. `\p{InMongolian}`: U+1800–U+18AF

43. \p{InLimbu}: U+1900–U+194F
44. \p{InTai_Le}: U+1950–U+197F
45. \p{InKhmer_Symbols}: U+19E0–U+19FF
46. \p{InPhonetic_Extensions}: U+1D00–U+1D7F
47. \p{InLatin_Extended_Additional}: U+1E00–U+1EFF
48. \p{InGreek_Extended}: U+1F00–U+1FFF
49. \p{InGeneral_Punctuation}: U+2000–U+206F
50. \p{InSuperscripts_and_Subscripts}: U+2070–U+209F
51. \p{InCurrency_Symbols}: U+20A0–U+20CF
52. \p{InCombining_Diacritical_Marks_for_Symbols}: U+20D0–U+20FF
53. \p{InLetterlike_Symbols}: U+2100–U+214F
54. \p{InNumber_Forms}: U+2150–U+218F
55. \p{InArrows}: U+2190–U+21FF
56. \p{InMathematical_Operators}: U+2200–U+22FF
57. \p{InMiscellaneous_Technical}: U+2300–U+23FF
58. \p{InControl_Pictures}: U+2400–U+243F
59. \p{InOptical_Character_Recognition}: U+2440–U+245F
60. \p{InEnclosed_Alphanumerics}: U+2460–U+24FF
61. \p{InBox_Drawing}: U+2500–U+257F
62. \p{InBlock_Elements}: U+2580–U+259F
63. \p{InGeometric_Shapes}: U+25A0–U+25FF
64. \p{InMiscellaneous_Symbols}: U+2600–U+26FF
65. \p{InDingbats}: U+2700–U+27BF
66. \p{InMiscellaneous_Mathematical_Symbols-A}: U+27C0–U+27EF
67. \p{InSupplemental_Arrows-A}: U+27F0–U+27FF
68. \p{InBraille_Patterns}: U+2800–U+28FF
69. \p{InSupplemental_Arrows-B}: U+2900–U+297F
70. \p{InMiscellaneous_Mathematical_Symbols-B}: U+2980–U+29FF
71. \p{InSupplemental_Mathematical_Operators}: U+2A00–U+2AFF
72. \p{InMiscellaneous_Symbols_and_Arrows}: U+2B00–U+2BFF
73. \p{InCJK_Radicals_Supplement}: U+2E80–U+2EFF
74. \p{InKangxi_Radicals}: U+2F00–U+2FDF
75. \p{InIdeographic_Description_Characters}: U+2FF0–U+2FFF
76. \p{InCJK_Symbols_and_Punctuation}: U+3000–U+303F
77. \p{InHiragana}: U+3040–U+309F
78. \p{InKatakana}: U+30A0–U+30FF
79. \p{InBopomofo}: U+3100–U+312F
80. \p{InHangul_Compatibility_Jamo}: U+3130–U+318F
81. \p{InKanbun}: U+3190–U+319F
82. \p{InBopomofo_Extended}: U+31A0–U+31BF
83. \p{InKatakana_Phonetic_Extensions}: U+31F0–U+31FF
84. \p{InEnclosed_CJK_Letters_and_Months}: U+3200–U+32FF
85. \p{InCJK_Compatibility}: U+3300–U+33FF
86. \p{InCJK_Unified_Ideographs_Extension_A}: U+3400–U+4DBF
87. \p{InYijing_Hexagram_Symbols}: U+4DC0–U+4DFF
88. \p{InCJK_Unified_Ideographs}: U+4E00–U+9FFF
89. \p{InYi_Syllables}: U+A000–U+A48F
90. \p{InYi_Radicals}: U+A490–U+A4CF
91. \p{InHangul_Syllables}: U+AC00–U+D7AF
92. \p{InHigh_Surrogates}: U+D800–U+DB7F
93. \p{InHigh_Private_Use_Surrogates}: U+DB80–U+DBFF
94. \p{InLow_Surrogates}: U+DC00–U+DFFF

95. `\p{InPrivate_Use_Area}`: U+E000–U+F8FF
96. `\p{InCJK_Compatibility_Ideographs}`: U+F900–U+FAFF
97. `\p{InAlphabetic_Presentation_Forms}`: U+FB00–U+FB4F
98. `\p{InArabic_Presentation_Forms-A}`: U+FB50–U+FDFF
99. `\p{InVariation_Selectors}`: U+FE00–U+FE0F
100. `\p{InCombining_Half_Marks}`: U+FE20–U+FE2F
101. `\p{InCJK_Compatibility_Forms}`: U+FE30–U+FE4F
102. `\p{InSmall_Form_Variants}`: U+FE50–U+FE6F
103. `\p{InArabic_Presentation_Forms-B}`: U+FE70–U+FEFF
104. `\p{InHalfwidth_and_Fullwidth_Forms}`: U+FF00–U+FFEF
105. `\p{InSpecials}`: U+FFF0–U+FFFF

Not all Unicode regex engines use the same syntax to match Unicode blocks. Java, Ruby 2.0, and XRegExp use the `\p{InBlock}` syntax as listed above. .NET and XML use `\p{IsBlock}` instead. Perl and the JGsoft flavor support both notations. I recommend you use the “In” notation if your regex engine supports it. “In” can only be used for Unicode blocks, while “Is” can also be used for Unicode properties and scripts, depending on the regular expression flavor you’re using. By using “In”, it’s obvious you’re matching a block and not a similarly named property or script.

In .NET and XML, you must omit the underscores but keep the hyphens in the block names. E.g. Use `\p{IsLatinExtended-A}` instead of `\p{InLatin_Extended-A}`. In Java, you must omit the hyphens. .NET and XML also compare the names case sensitively, while Perl, Ruby, and the JGsoft flavor compare them case insensitively. Java 4 is case sensitive. Java 5 and later are case sensitive for the “Is” prefix but not for the block names themselves.

The actual names of the blocks are the same in all regular expression engines. The block names are defined in the Unicode standard. PCRE and PHP do not support Unicode blocks, even though they support Unicode scripts.

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. The regex `à` encoded as U+00E0 matches `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 always uses the one-on-one substitution. Since all the

Windows or ISO-8859 code pages encode accented characters as a single code point, nearly all software uses a single Unicode code point for each character when converting the file to Unicode.

23. Specifying Modes Inside The Regular Expression

Normally, matching modes are specified outside the regular expression. In a programming language, you pass them as a flag to the regex constructor or append them to the regex literal. In an application, you'd toggle the appropriate buttons or checkboxes. You can find the specifics in the tools and languages section of this book.

Sometimes, the tool or language does not provide the ability to specify matching options. The handy `String.matches()` method in Java does not take a parameter for matching options like `Pattern.compile()` does. Or, the regex flavor may support matching modes that aren't exposed as external flags. The regex functions in R have `ignore.case` as their only option, even though the underlying PCRE library has more matching modes than any other discussed in this tutorial.

In those situation, you can add the following mode modifiers to the start of the regex. To specify multiple modes, simply put them together as in `(?ismx)`.

- `(?i)` makes the regex case insensitive.
- `(?c)` makes the regex case sensitive. Only supported by Tcl.
- `(?x)` turn on free-spacing mode.
- `(?t)` turn off free-spacing mode. Only supported by Tcl.
- `(?xx)` turn on free-spacing mode, also in character classes. Supported by Perl 5.26 and PCRE2 10.30.
- `(?s)` for “single line mode” makes the dot match all characters, including line breaks. Not supported by Ruby. In Tcl, `(?s)` also makes the caret and dollar match at the start and end of the string only.
- `(?m)` for “multi-line mode” makes the caret and dollar match at the start and end of each line in the subject string. In Ruby, `(?m)` makes the dot match all characters, without affecting the caret and dollar which always match at the start and end of each line in Ruby. In Tcl, `(?m)` also prevents the dot from matching line breaks.
- `(?p)` in Tcl makes the caret and dollar match at the start and the end of each line, and makes the dot match line breaks.
- `(?w)` in Tcl makes the caret and dollar match only at the start and the end of the subject string, and prevents the dot from matching line breaks.
- `(?n)` turns all unnamed groups into non-capturing groups. Only supported by .NET, XRegExp, and the JGsoft flavor. In Tcl, `(?n)` is the same as `(?m)`.
- `(?J)` allows duplicate group names. Only supported by PCRE and languages that use it such as Delphi, PHP and R.
- `(?U)` turns on “ungreedy mode”, which switches the syntax for greedy and lazy quantifiers. So `(?U)a*` is lazy and `(?U)a*?` is greedy. Only supported by PCRE and languages that use it. Its use is strongly discouraged because it confuses the meaning of the standard quantifier syntax.
- `(?d)` corresponds with `UNIX_LINES` in Java, which makes the dot, caret, and dollar treat only the newline character `\n` as a line break, instead of recognizing all line break characters from the Unicode standard. Whether they match or don't match (at) line breaks depends on `(?s)` and `(?m)`.
- `(?b)` makes Tcl interpret the regex as a POSIX BRE.
- `(?e)` makes Tcl interpret the regex as a POSIX ERE.
- `(?q)` makes Tcl interpret the regex as a literal string (minus the `(?q)` characters).
- `(?X)` makes escaping letters with a backslash an error if that combination is not a valid regex token. Only supported by PCRE and languages that use it.

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 then the modifier only applies to the part of the regex to the right of the modifier. With these flavors, 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.

If a flavor can't apply modifiers to only part of the regex then it treats modifiers in the middle of the regex as an error. Python is an exception to this. In Python, putting a modifier in the middle of the regex affects the whole regex. So in Python, `(?i)caseless` and `caseless(?i)` are both case insensitive. In all other flavors, the trailing mode modifier either has no effect or is an error.

You can quickly test how the regex flavor you're using handles mode modifiers. The regex `(?i)te(?-i)st` should match `test` and `TEst`, but not `teST` or `TEST`.

Modifier Spans

Instead of using two modifiers, one to turn an option on, and one to turn it off, you use a modifier span. `(?i)caseless(?-i)cased(?i)caseless` is equivalent to `(?i)caseless(?-i:cased)caseless`. This syntax resembles that of the non-capturing group `(?:group)`. You could think of a non-capturing group as a modifier span that does not change any modifiers. But there are flavors, like JavaScript, Python, and Tcl that support non-capturing groups even though they do not support modifier spans. Like a non-capturing group, the modifier span does not create a backreference.

Modifier spans are supported by all regex flavors that allow you to use mode modifiers in the middle of the regular expression, and by those flavors only. These include the JGsoft engine, .NET, Java, Perl and PCRE, PHP, Delphi, and R.

24. Atomic Grouping

An atomic group is a group that, when the regex engine exits from it, automatically throws away all backtracking positions remembered by any tokens inside the group. Atomic groups are non-capturing. The syntax is `(?>group)`. Lookaround groups are also atomic. Atomic grouping is supported by most modern regular expression flavors, including the JGsoft flavor, Java, PCRE, .NET, Perl, Boost, and Ruby. Most of these also support possessive quantifiers, which are essentially a notational convenience for atomic grouping. Python supports atomic grouping and possessive quantifiers starting with Python version 3.11.

An example will make the behavior of atomic groups clear. The regular expression `a(bc|b)c` (capturing group) matches `abcc` and `abc`. The regex `a(?>bc|b)c` (atomic group) matches `abcc` but not `abc`.

When applied to `abc`, both regexes will match `a` to `a`, `bc` to `bc`, and then `c` will fail to match at the end of the string. Here their paths diverge. The regex with the capturing group has remembered a backtracking position for the alternation. The group will give up its match, `b` then matches `b` and `c` matches `c`. Match found!

The regex with the atomic group, however, exited from an atomic group after `bc` was matched. At that point, all backtracking positions for tokens inside the group are discarded. In this example, the alternation's option to try `b` at the second position in the string is discarded. As a result, when `c` fails, the regex engine has no alternatives left to try.

Of course, the above example isn't very useful. But it does illustrate very clearly how atomic grouping eliminates certain matches. Or more importantly, it eliminates certain match attempts.

Regex Optimization Using Atomic Grouping

Consider the regex `\b(integer|insert|in)\b` and the subject `integers`. Obviously, because of the word boundaries, these don't match. What's not so obvious is that the regex engine will spend quite some effort figuring this out.

`\b` matches at the start of the string, and `integer` matches `integer`. The regex engine makes note that there are two more alternatives in the group, and continues with `\b`. This fails to match between the `r` and `s`. So the engine backtracks to try the second alternative inside the group. The second alternative matches `in`, but then fails to match `s`. So the engine backtracks once more to the third alternative. `in` matches `in`. `\b` fails between the `n` and `t` this time. The regex engine has no more remembered backtracking positions, so it declares failure.

This is quite a lot of work to figure out `integers` isn't in our list of words. We can optimize this by telling the regular expression engine that if it can't match `\b` after it matched `integer`, then it shouldn't bother trying any of the other words. The word we've encountered in the subject string is a longer word, and it isn't in our list.

We can do this by turning the capturing group into an atomic group: `\b(?>integer|insert|in)\b`. Now, when `integer` matches, the engine exits from an atomic group, and throws away the backtracking positions it stored for the alternation. When `\b` fails, the engine gives up immediately. This savings can be significant when scanning a large file for a long list of keywords. This savings will be vital when your alternatives contain repeated tokens (not to mention repeated groups) that lead to catastrophic backtracking.

Don't be too quick to make all your groups atomic. As we saw in the first example above, atomic grouping can exclude valid matches too. Compare how `\b(?:integer|insert|in)\b` and `\b(?:in|integer|insert)\b` behave when applied to `insert`. The former regex matches, while the latter fails. If the groups weren't atomic, both regexes would match. Remember that alternation tries its alternatives from left to right. If the second regex matches `in`, it won't try the two other alternatives due to the atomic group.

25. Possessive Quantifiers

The topic on repetition operators or quantifiers explains the difference between greedy and lazy repetition. Greediness and laziness determine the order in which the regex engine tries the possible permutations of the regex pattern. A greedy quantifier first tries to repeat the token as many times as possible, and gradually gives up matches as the engine backtracks to find an overall match. A lazy quantifier first repeats the token as few times as required, and gradually expands the match as the engine backtracks through the regex to find an overall match.

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

Possessive quantifiers are a way to prevent the regex engine from trying all permutations. This is primarily useful for performance reasons. You can also use possessive quantifiers to eliminate certain matches.

Of the regex flavors discussed in this tutorial, possessive quantifiers are supported by JGsoft, Java, and PCRE. That includes languages with regex support based on PCRE such as PHP, Delphi, and R. Python supports possessive quantifiers starting with Python 3.11, Perl supports them starting with Perl 5.10, Ruby starting with Ruby 1.9, and Boost starting with Boost 1.42.

How Possessive Quantifiers Work

Like a greedy quantifier, a possessive quantifier repeats the token as many times as possible. Unlike a greedy quantifier, it does *not* give up matches as the engine backtracks. With a possessive quantifier, the deal is all or nothing. You can make a quantifier possessive by placing an extra + after it. * is greedy, *? is lazy, and *+ is possessive. ++, ?+ and {n,m}+ are all possessive as well.

Let's see what happens if we try to match "[^"]*+" against "abc". The " matches the ". [^"] matches a, b and c as it is repeated by the star. The final " then matches the final " and we found an overall match. In this case, the end result is the same, whether we use a greedy or possessive quantifier. There is a slight performance increase though, because the possessive quantifier doesn't have to remember any backtracking positions.

The performance increase can be significant in situations where the regex fails. If the subject is "abc (no closing quote), the above matching process happens in the same way, except that the second " fails. When using a possessive quantifier, there are no steps to backtrack to. The regular expression does not have any alternation or non-possessive quantifiers that can give up part of their match to try a different permutation of the regular expression. So the match attempt fails immediately when the second " fails.

Had we used "[^"]*" with a greedy quantifier instead, the engine would have backtracked. After the " failed at the end of the string, the [^"]* would give up one match, leaving it with ab. The " would then fail to match c. [^"]* backtracks to just a, and " fails to match b. Finally, [^"]* backtracks to match zero characters, and " fails a. Only at this point have all backtracking positions been exhausted, and does the engine give up the match attempt. Essentially, this regex performs as many needless steps as there are characters following the unmatched opening quote.

When Possessive Quantifiers Matter

The main practical benefit of possessive quantifiers is to speed up your regular expression. In particular, possessive quantifiers allow your regex to fail faster. In the above example, when the closing quote fails to match, we *know* the regular expression couldn't possibly have skipped over a quote. So there's no need to backtrack and check for the quote. We make the regex engine aware of this by making the quantifier possessive. In fact, some engines, including the JGsoft engine, detect that `[^"]*` and `"` are mutually exclusive when compiling your regular expression, and automatically make the star possessive.

Now, linear backtracking like a regex with a single quantifier does is pretty fast. It's unlikely you'll notice the speed difference. However, when you're nesting quantifiers, a possessive quantifier may save your day. Nesting quantifiers means that you have one or more repeated tokens inside a group, and the group is also repeated. That's when catastrophic backtracking often rears its ugly head. In such cases, you'll depend on possessive quantifiers and/or atomic grouping to save the day.

Possessive Quantifiers Can Change The Match Result

Using possessive quantifiers can change the result of a match attempt. Since no backtracking is done, and matches that would require a greedy quantifier to backtrack will not be found with a possessive quantifier. For example, `".*"` matches `"abc"` in `"abc"x`, but `".*+"` does not match this string at all.

In both regular expressions, the first `"` matches the first `"` in the string. The repeated dot then matches the remainder of the string `abc"x`. The second `"` then fails to match at the end of the string.

Now, the paths of the two regular expressions diverge. The possessive dot-star wants it all. No backtracking is done. Since the `"` failed, there are no permutations left to try, and the overall match attempt fails. The greedy dot-star, while initially grabbing everything, is willing to give back. It will backtrack one character at a time. Backtracking to `abc"`, `"` fails to match `x`. Backtracking to `abc`, `"` matches `"`. An overall match `"abc"` is found.

Essentially, the lesson here is that when using possessive quantifiers, you need to make sure that whatever you're applying the possessive quantifier to should not be able to match what should follow it. The problem in the above example is that the dot also matches the closing quote. This prevents us from using a possessive quantifier. The negated character class in the previous section cannot match the closing quote, so we can make it possessive.

Using Atomic Grouping Instead of Possessive Quantifiers

Technically, possessive quantifiers are a notational convenience to place an atomic group around a single quantifier. All regex flavors that support possessive quantifiers also support atomic grouping. But not all regex flavors that support atomic grouping support possessive quantifiers. With those flavors, you can achieve the exact same results using an atomic group.

Basically, instead of `X*+`, write `(?>X*)`. It is important to notice that both the quantified token X and the quantifier are inside the atomic group. Even if X is a group, you still need to put an extra atomic group around it to achieve the same effect. `(?:a|b)*+` is equivalent to `(?>(?:a|b)*)` but not to `(?>a|b)*`. The

latter is a valid regular expression, but it won't have the same effect when used as part of a larger regular expression.

To illustrate, `(?:a|b)*+b` and `(?>(?:a|b)*b)` both fail to match `b`. `a|b` matches the `b`. The star is satisfied, and the fact that it's possessive or the atomic group will cause the star to forget all its backtracking positions. The second `b` in the regex has nothing left to match, and the overall match attempt fails.

In the regex `(?>a|b)*b`, the atomic group forces the alternation to give up its backtracking positions. This means that if an `a` is matched, it won't come back to try `b` if the rest of the regex fails. Since the star is outside of the group, it is a normal, greedy star. When the second `b` fails, the greedy star backtracks to zero iterations. Then, the second `b` matches the `b` in the subject string.

This distinction is particularly important when converting a regular expression written by somebody else using possessive quantifiers to a regex flavor that doesn't have possessive quantifiers. You could, of course, let a tool like `RegexBuddy` do the conversion for you.

26. Lookahead and Lookbehind Zero-Length Assertions

Lookahead and lookbehind, collectively called “lookaround”, are zero-length assertions just like the start and end of line, and start and end of word anchors explained earlier in this tutorial. The difference is that lookaround actually matches characters, but then gives up the match, returning only the result: match or no match. That is why they are called “assertions”. They do not consume characters in the string, but only assert whether a match is possible or not. Lookaround allows you to create regular expressions that are impossible to create without them, or that would get very longwinded without them.

Positive and Negative Lookahead

Negative lookahead is indispensable if you want to match something not followed by something else. When explaining character classes, this tutorial explained why you cannot use a negated character class to match a `q` not followed by a `u`. Negative lookahead provides the solution: `q(?:u)`. The negative lookahead construct is the pair of parentheses, with the opening parenthesis followed by a question mark and an exclamation point. Inside the lookahead, we have the trivial regex `u`.

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

You can use any regular expression inside the lookahead (but not lookbehind, as explained below). Any valid regular expression can be used inside the lookahead. If it contains capturing groups then those groups will capture as normal and backreferences to them will work normally, even outside the lookahead. (The only exception is Tcl, which treats all groups inside lookahead as non-capturing.) The lookahead itself is not a capturing group. It is not included in the count towards numbering the backreferences. If you want to store the match of the regex inside a lookahead, you have to put capturing parentheses around the regex inside the lookahead, like this: `(?=(regex))`. The other way around will not work, because the lookahead will already have discarded the regex match by the time the capturing group is to store its match.

Regex Engine Internals

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

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

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

Let's take one more look inside, to make sure you understand the implications of the lookahead. Let's apply `q(?:u)i` to `quit`. The lookahead is now positive and is followed by another token. Again, `q` matches `q` and `u` matches `u`. Again, the match from the lookahead must be discarded, so the engine steps back from `i` in the string to `u`. The lookahead was successful, so the engine continues with `i`. But `i` cannot match `u`. So this match attempt fails. All remaining attempts fail as well, because there are no more `q`'s in the string.

The regex `q(?:u)i` can never match anything. It tries to match `u` and `i` at the same position. If there is a `u` immediately after the `q` then the lookahead succeeds but then `i` fails to match `u`. If there is anything other than a `u` immediately after the `q` then the lookahead fails.

Positive and Negative Lookbehind

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

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

More Regex Engine Internals

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

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

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

Important Notes About Lookbehind

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 matches `John` and the latter matches `John'` (including the apostrophe). I will leave it up to you to figure out why. (Hint: `\b` matches between the apostrophe and the `s`). The latter also doesn't match single-letter words like “a” or “I”. The correct regex without using lookbehind is `\b\w*[^s\W]\b` (star instead of plus, and `\W` in the character class). Personally, I find the lookbehind easier to understand. The last regex, which works correctly, has a double negation (the `\W` in the negated character class). Double negations tend to be confusing to humans. Not to regex engines, though. (Except perhaps for Tcl, which treats negated shorthands in negated character classes as an error.)

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. The regular expression engine needs to be able to figure out how many characters to step back before checking the lookbehind. When evaluating the lookbehind, the regex engine determines the length of the regex inside the lookbehind, steps back that many characters in the subject string, and then applies the regex inside the lookbehind from left to right just as it would with a normal regex.

Many regex flavors, including those used by Perl, Python, and Boost only allow fixed-length strings. You can use literal text, character escapes, Unicode escapes other than `\x`, and character classes. You cannot use quantifiers or backreferences. You can use alternation, but only if all alternatives have the same length. These flavors evaluate lookbehind by first stepping back through the subject string for as many characters as the lookbehind needs, and then attempting the regex inside the lookbehind from left to right.

Perl 5.30 supports variable-length lookbehind as an experimental feature. But there are many cases in which it does not work correctly. So in practice, the above is still true for Perl 5.30.

PCRE is not fully Perl-compatible when it comes to lookbehind. While Perl requires alternatives inside lookbehind to have the same length, PCRE allows alternatives of variable length. PHP, Delphi, R, and Ruby also allow this. Each alternative still has to be fixed-length. Each alternative is treated as a separate fixed-length lookbehind.

Java takes things a step further by allowing finite repetition. You can use the question mark and the curly braces with the *max* parameter specified. Java determines the minimum and maximum possible lengths of the lookbehind. The lookbehind in the regex `(?<!ab{2,4}c{3,5}d)test` has 5 possible lengths. It can be from 7 through 11 characters long. When Java (version 6 or later) tries to match the lookbehind, it first steps back the minimum number of characters (7 in this example) in the string and then evaluates the regex inside the lookbehind as usual, from left to right. If it fails, Java steps back one more character and tries again. If the lookbehind continues to fail, Java continues to step back until the lookbehind either matches or it has stepped back the maximum number of characters (11 in this example). This repeated stepping back through the subject string kills performance when the number of possible lengths of the lookbehind grows. Keep this in mind. Don't choose an arbitrarily large maximum number of repetitions to work around the lack of infinite quantifiers inside lookbehind. Java 4 and 5 have bugs that cause lookbehind with alternation or variable quantifiers to fail when it should succeed in some situations. These bugs were fixed in Java 6.

Java 13 allows you to use the star and plus inside lookbehind, as well as curly braces without an upper limit. But Java 13 still uses the laborious method of matching lookbehind introduced with Java 6. Java 13 also does not correctly handle lookbehind with multiple quantifiers if one of them is unbounded. In some situations you may get an error. In other situations you may get incorrect matches. So for both correctness and

performance, we recommend you only use quantifiers with a low upper bound in lookbehind with Java 6 through 13.

The only regex engines that allow you to use a full regular expression inside lookbehind, including infinite repetition and backreferences, are the JGsoft engine and the .NET RegEx classes. These regex engines really apply the regex inside the lookbehind backwards, going through the regex inside the lookbehind and through the subject string from right to left. They only need to evaluate the lookbehind once, regardless of how many different possible lengths it has.

Finally, flavors like `std::regex` and `Tcl` do not support lookbehind at all, even though they do support lookahead. JavaScript was like that for the longest time since its inception. But now lookbehind is part of the ECMAScript 2018 specification. As of this writing (late 2019), Google's Chrome browser is the only popular JavaScript implementation that supports lookbehind. So if cross-browser compatibility matters, you can't use lookbehind in JavaScript.

Lookaround Is Atomic

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

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

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

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

If you don't use capturing groups inside lookahead, then all this doesn't matter. Either the lookahead condition can be satisfied or it cannot be. In how many ways it can be satisfied is irrelevant.

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

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

A more practical example makes this clear. Let's say we want to find a word that is six letters long and contains the three consecutive letters `cat`. Actually, we can match this without lookaround. We just specify all the options and lump them together using alternation: `cat\w{3}|\wcat\w{2}|\w{2}cat\w|\w{3}cat`. Easy enough. But this method gets unwieldy if you want to find any word between 6 and 12 letters long containing either “cat”, “dog” or “mouse”.

Lookaround to The Rescue

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

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

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

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

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

Optimizing Our Solution

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

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

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

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

You could replace the final `\w*` with `\w{0,3}` too. But it wouldn’t make any difference. The lookahead has already checked that we’re at a 6-letter word, and `\w{0,3}cat` has already matched 3 to 6 letters of that word. Whether we end the regex with `\w*` or `\w{0,3}` doesn’t matter, because either way, we’ll be matching all the remaining word characters. Because the resulting match and the speed at which it is found are the same, we may just as well use the version that is easier to type.

A More Complex Problem

So, what would you use to find any word between 6 and 12 letters long containing either “cat”, “dog” or “mouse”? Again we have two requirements, which we can easily combine using a lookahead: `\b(?:\w{6,12}\b)\w{0,9}(cat|dog|mouse)\w*`. Very easy, once you get the hang of it. This regex will also put “cat”, “dog” or “mouse” into the first backreference.

28. Keep The Text Matched So Far out of The Overall Regex Match

Lookbehind is often used to match certain text that is preceded by other text, without including the other text in the overall regex match. `(?<=h)d` matches only the second `d` in `adhd`. While a lot of regex flavors support lookbehind, most regex flavors only allow a subset of the regex syntax to be used inside lookbehind. Perl and Boost require the lookbehind to be of fixed length. PCRE and Ruby allow alternatives of different length, but still don't allow quantifiers other than the fixed-length `{n}`.

To overcome the limitations of lookbehind, Perl 5.10, PCRE 7.2, Ruby 2.0, and Boost 1.42 introduced a new feature that can be used instead of lookbehind for its most common purpose. `\K` keeps the text matched so far out of the overall regex match. `h\d` matches only the second `d` in `adhd`.

The JGsoft flavor has always supported unrestricted lookbehind, which is much more flexible than `\K`. Still, JGsoft V2 adds support for `\K` if you prefer this way of working.

Looking Inside The Regex Engine

Let's see how `h\d` works. The engine begins the match attempt at the start of the string. `h` fails to match `a`. There are no further alternatives to try. The match attempt at the start of the string has failed.

The engine advances one character through the string and attempts the match again. `h` fails to match `d`.

Advancing again, `h` matches `h`. The engine advances through the regex. The regex has now reached `\K` in the regex and the position between `h` and the second `d` in the string. `\K` does nothing other than to tell that if this match attempt ends up succeeding, the regex engine should pretend that the match attempt started at the present position between `h` and `d`, rather than between the first `d` and `h` where it really started.

The engine advances through the regex. `d` matches the second `d` in the string. An overall match is found. Because of the position saved by `\K`, the second `d` in the string is returned as the overall match.

`\K` only affects the position returned after a successful match. It does not move the start of the match attempt during the matching process. The regex `hhh\d` matches the `d` in `hhhd`. This regex first matches `hhh` at the start of the string. Then `\K` notes the position between `hhh` and `hd` in the string. Then `d` fails to match the fourth `h` in the string. The match attempt at the start of the string has failed.

Now the engine must advance one character in the string before starting the next match attempt. It advances from the actual start of the match attempt, which was at the start of the string. The position stored by `\K` does not change this. So the second match attempt begins at the position after the first `h` in the string. Starting there, `hhh` matches `hhh`, `\K` notes the position, and `d` matches `d`. Now, the position remembered by `\K` is taken into account, and `d` is returned as the overall match.

`\K` Can Be Used Anywhere

You can use `\K` pretty much anywhere in any regular expression. You should only avoid using it inside lookbehind. You can use it inside groups, even when they have quantifiers. You can have as many instances

of `\K` in your regex as you like. `(ab\Kc|d\Ke)f` matches `cf` when preceded by `ab`. It also matches `ef` when preceded by `d`.

`\K` does not affect capturing groups. When `(ab\Kc|d\Ke)f` matches `cf`, the capturing group captures `abc` as if the `\K` weren't there. When the regex matches `ef`, the capturing group stores `de`.

Limitations of `\K`

Because `\K` does not affect the way the regex engine goes through the matching process, it offers a lot more flexibility than lookbehind in Perl, PCRE, and Ruby. You can put anything to the left of `\K`, but you're limited to what you can put inside lookbehind.

But this flexibility does come at a cost. Lookbehind really goes backwards through the string. This allows lookbehind check for a match before the start of the match attempt. When the match attempt was started at the end of the previous match, lookbehind can match text that was part of the previous match. `\K` cannot do this, precisely because it does not affect the way the regex engine goes through the matching process.

If you iterate over all matches of `(?<=a)a` in the string `aaaa`, you will get three matches: the second, third, and fourth `a` in the string. The first match attempt begins at the start of the string and fails because the lookbehind fails. The second match attempt begins between the first and second `a`, where the lookbehind succeeds and the second `a` is matched. The third match attempt begins after the second `a` that was just matched. Here the lookbehind succeeds too. It doesn't matter that the preceding `a` was part of the previous match. Thus the third match attempt matches the third `a`. Similarly, the fourth match attempt matches the fourth `a`. The fifth match attempt starts at the end of the string. The lookbehind still succeeds, but there are no characters left for `a` to match. The match attempt fails. The engine has reached the end of the string and the iteration stops. Five match attempts have found three matches.

Things are different when you iterate over `a\Ka` in the string `aaaa`. You will get only two matches: the second and the fourth `a`. The first match attempt begins at the start of the string. The first `a` in the regex matches the first `a` in the string. `\K` notes the position. The second `a` matches the second `a` in the string, which is returned as the first match. The second match attempt begins after the second `a` that was just matched. The first `a` in the regex matches the third `a` in the string. `\K` notes the position. The second `a` matches the fourth `a` in the string, which is returned as the first match. The third match attempt begins at the end of the string. `a` fails. The engine has reached the end of the string and the iteration stops. Three match attempts have found two matches.

Basically, you'll run into this issue when the part of the regex before the `\K` can match the same text as the part of the regex after the `\K`. If those parts can't match the same text, then a regex using `\K` will find the same matches than the same regex rewritten using lookbehind. In that case, you should use `\K` instead of lookbehind as that will give you better performance in Perl, PCRE, and Ruby.

Another limitation is that while lookbehind comes in positive and negative variants, `\K` does not provide a way to negate anything. `(?<!a)b` matches the string `b` entirely, because it is a "b" not preceded by an "a". `[^a]\Kb` does not match the string `b` at all. When attempting the match, `[^a]` matches `b`. The regex has now reached the end of the string. `\K` notes this position. But now there is nothing left for `b` to match. The match attempt fails. `[^a]\Kb` is the same as `(?<=[^a])b`, which are both different from `(?<!a)b`.

29. If-Then-Else Conditionals in Regular Expressions

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

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

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

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

For the *then* and *else*, you can use any regular expression. If you want to use alternation, you will have to group the *then* or *else* together using parentheses, like in `(?(?=condition)(then1|then2|then3)|else1|else2|else3)`. Otherwise, there is no need to use parentheses around the *then* and *else* parts.

Looking Inside The Regex Engine

The regex `(a)?b(?(1)c|d)` consists of the optional capturing group `(a)?`, the literal `b`, and the conditional `(?(1)c|d)` that tests the capturing group. This regex matches `bd` and `abc`. It does not match `bc`, but does match `bd` in `abd`. Let's see how this regular expression works on each of these four subject strings.

When applied to `bd`, `a` fails to match. Since the capturing group containing `a` is optional, the engine continues with `b` at the start of the subject string. Since the whole group was optional, the group did not take part in the match. Any subsequent backreference to it like `\1` will fail. Note that `(a)?` is very different from `(a?)`. In the former regex, the capturing group does not take part in the match if `a` fails, and backreferences to the group will fail. In the latter group, the capturing group always takes part in the match, capturing either `a` or nothing. Backreferences to a capturing group that took part in the match and captured nothing always succeed. Conditionals evaluating such groups execute the “then” part. In short: if you want to use a reference to a group in a conditional, use `(a)?` instead of `(a?)`.

Continuing with our regex, `b` matches `b`. The regex engine now evaluates the conditional. The first capturing group did not take part in the match at all, so the “else” part or `d` is attempted. `d` matches `d` and an overall match is found.

Moving on to our second subject string `abc`, `a` matches `a`, which is captured by the capturing group. Subsequently, `b` matches `b`. The regex engine again evaluates the conditional. The capturing group took part in the match, so the “then” part or `c` is attempted. `c` matches `c` and an overall match is found.

Our third subject `bc` does not start with `a`, so the capturing group does not take part in the match attempt, like we saw with the first subject string. `b` still matches `b`, and the engine moves on to the conditional. The first capturing group did not take part in the match at all, so the “else” part or `d` is attempted. `d` does not match `c` and the match attempt at the start of the string fails. The engine does try again starting at the second character in the string, but fails since `b` does not match `c`.

The fourth subject `abd` is the most interesting one. Like in the second string, the capturing group grabs the `a` and the `b` matches. The capturing group took part in the match, so the “then” part or `c` is attempted. `c` fails to match `d`, and the match attempt fails. Note that the “else” part is not attempted at this point. The capturing group took part in the match, so only the “then” part is used. However, the regex engine isn’t done yet. It restarts the regular expression from the beginning, moving ahead one character in the subject string.

Starting at the second character in the string, `a` fails to match `b`. The capturing group does not take part in the second match attempt which started at the second character in the string. The regex engine moves beyond the optional group, and attempts `b`, which matches. The regex engine now arrives at the conditional in the regex, and at the third character in the subject string. The first capturing group did not take part in the current match attempt, so the “else” part or `d` is attempted. `d` matches `d` and an overall match `bd` is found.

If you want to avoid this last match result, you need to use anchors. `^(a)?b(?:c|d)$` does not find any matches in the last subject string. The caret fails to match before the second and third characters in the string.

Named and Relative Conditionals

Conditionals are supported by the JGsoft engine, Perl, PCRE, Python, and .NET. Ruby supports them starting with version 2.0. Languages such as Delphi, PHP, and R that have regex features based on PCRE also support conditionals.

All these flavors also support named capturing groups. You can use the name of a capturing group instead of its number as the *if* test. The syntax is slightly inconsistent between regex flavors. In Python, .NET, and the JGsoft applications, you simply specify the name of the group between parentheses. `(?<test>a)?b(?:test)c|d` is the regex from the previous section using named capture. In Perl or Ruby, you have to put angle brackets or quotes around the name of the group, and put that between the conditional’s parentheses: `(?<test>a)?b(?:<test>c|d)` or `(?'test'a)?b(?:'test'c|d)`. PCRE supports all three variants.

PCRE 7.2 and later and JGsoft V2 also support relative conditionals. The syntax is the same as that of a conditional that references a numbered capturing group with an added plus or minus sign before the group number. The conditional then counts the opening parentheses to the left (minus) or to the right (plus) starting at the `(?(` that opens the conditional. `(a)?b(?:(-1)c|d)` is another way of writing the above regex. The benefit is that this regex won’t break if you add capturing groups at the start or the end of the regex.

Python supports conditionals using a numbered or named capturing group. Python does not support conditionals using lookahead, even though Python does support lookahead outside conditionals. Instead of a conditional like `(?(?=regex)then|else)`, you can alternate two opposite lookaheads: `(?=regex)then|(?!regex)else`.

Conditionals Referencing Non-Existent Capturing Groups

Boost and Ruby treat a conditional that references a non-existent capturing group as an error. The latest versions of all other flavors discussed in this tutorial don't. They simply let such conditionals always attempt the "else" part. A few flavors changed their minds, though. Python 3.4 and prior and PCRE 7.6 and prior (and thus PHP 5.2.5 and prior) used to treat them as errors.

Example: Extract Email Headers

The regex `^((From|To|Subject): ((?(2)\w+\@\w+\.[a-z]+|.+))` extracts the From, To, and Subject headers from an email message. The name of the header is captured into the first backreference. If the header is the From or To header, it is captured into the second backreference as well.

The second part of the pattern is the if-then-else conditional `((?(2)\w+\@\w+\.[a-z]+|.+))`. The *if* part checks whether the second capturing group took part in the match thus far. It will have taken part if the header is the From or To header. In that case, the *then* part of the conditional `\w+\@\w+\.[a-z]+` tries to match an email address. To keep the example simple, we use an overly simple regex to match the email address, and we don't try to match the display name that is usually also part of the From or To header.

If the second capturing group did not participate in the match this far, the *else* part `.+` is attempted instead. This simply matches the remainder of the line, allowing for any test subject.

Finally, we place an extra pair of parentheses around the conditional. This captures the contents of the email header matched by the conditional into the third backreference. The conditional itself does not capture anything. When implementing this regular expression, the first capturing group will store the name of the header ("From", "To", or "Subject"), and the third capturing group will store the value of the header.

You could try to match even more headers by putting another conditional into the "else" part. E.g. `^((From|To|Date|Subject): ((?(2)\w+\@\w+\.[a-z]+|(?(3)mm/dd/yyyy|.+))` would match a "From", "To", "Date" or "Subject", and use the regex `mm/dd/yyyy` to check whether the date is valid. Obviously, the date validation regex is just a dummy to keep the example simple. The header is captured in the first group, and its validated contents in the fourth group.

As you can see, regular expressions using conditionals quickly become unwieldy. I recommend that you only use them if one regular expression is all your tool allows you to use. When programming, you're far better off using the regex `^(From|To|Date|Subject): (.+)` to capture one header with its unvalidated contents. In your source code, check the name of the header returned in the first capturing group, and then use a second regular expression to validate the contents of the header returned in the second capturing group of the first regex. Though you'll have to write a few lines of extra code, this code will be much easier to understand and maintain. If you precompile all the regular expressions, using multiple regular expressions will be just as fast, if not faster, than the one big regex stuffed with conditionals.

30. Matching Nested Constructs with Balancing Groups

The .NET regex flavor has a special feature called balancing groups. The main purpose of balancing groups is to match balanced constructs or nested constructs, which is where they get their name from. A technically more accurate name for the feature would be capturing group subtraction. That's what the feature really does. It's .NET's solution to a problem that other regex flavors like Perl, PCRE, and Ruby handle with regular expression recursion. JGsoft V2 supports both balancing groups and recursion.

`(?<capture-subtract>regex)` or `(?'capture-subtract'regex)` is the basic syntax of a balancing group. It's the same syntax used for named capturing groups in .NET but with two group names delimited by a minus sign. The name of this group is "capture". You can omit the name of the group. `(?<-subtract>regex)` or `(?'-subtract'regex)` is the syntax for a non-capturing balancing group.

The name "subtract" must be the name of another group in the regex. When the regex engine enters the balancing group, it subtracts one match from the group "subtract". If the group "subtract" did not match yet, or if all its matches were already subtracted, then the balancing group fails to match. You could think of a balancing group as a conditional that tests the group "subtract", with "regex" as the "if" part and an "else" part that always fails to match. The difference is that the balancing group has the added feature of subtracting one match from the group "subtract", while a conditional leaves the group untouched.

If the balancing group succeeds and it has a name ("capture" in this example), then the group captures the text between the end of the match that was subtracted from the group "subtract" and the start of the match of the balancing group itself ("regex" in this example).

The reason this works in .NET is that capturing groups in .NET keep a stack of everything they captured during the matching process that wasn't backtracked or subtracted. Most other regex engines only store the most recent match of each capturing groups. When `(\w)+` matches `abc` then `Match.Groups[1].Value` returns `c` as with other regex engines, but `Match.Groups[1].Captures` stores all three iterations of the group: `a`, `b`, and `c`.

Looking Inside The Regex Engine

Let's apply the regex `(?'open'o)+(?'between-open'c)+` to the string `ooccc`. `(?'open'o)` matches the first `o` and stores that as the first capture of the group "open". The quantifier `+` repeats the group. `(?'open'o)` matches the second `o` and stores that as the second capture. Repeating again, `(?'open'o)` fails to match the first `c`. But the `+` is satisfied with two repetitions.

The regex engine advances to `(?'between-open'c)`. Before the engine can enter this balancing group, it must check whether the subtracted group "open" has captured something. It has captured the second `o`. The engine enters the group, subtracting the most recent capture from "open". This leaves the group "open" with the first `o` as its only capture. Now inside the balancing group, `c` matches `c`. The engine exits the balancing group. The group "between" captures the text between the match subtracted from "open" (the second `o`) and the `c` just matched by the balancing group. This is an empty string but it is captured anyway.

The balancing group too has `+` as its quantifier. The engine again finds that the subtracted group "open" captured something, namely the first `o`. The regex enters the balancing group, leaving the group "open" without any matches. `c` matches the second `c` in the string. The group "between" captures `oc` which is the

text between the match subtracted from “open” (the first `o`) and the second `c` just matched by the balancing group.

The balancing group is repeated again. But this time, the regex engine finds that the group “open” has no matches left. The balancing group fails to match. The group “between” is unaffected, retaining its most recent capture.

The `+` is satisfied with two iterations. The engine has reached the end of the regex. It returns `ooCC` as the overall match. `Match.Groups['open'].Success` will return `false`, because all the captures of that group were subtracted. `Match.Groups['between'].Value` returns `"oc"`.

Matching Balanced Pairs

We need to modify this regex if we want it to match a balanced number of o’s and c’s. To make sure that the regex won’t match `ooCCC`, which has more c’s than o’s, we can add anchors: `^(?'open'o)+(?'-open'c)+$`. This regex goes through the same matching process as the previous one. But after `(?'-open'c)+` fails to match its third iteration, the engine reaches `$` instead of the end of the regex. This fails to match. The regex engine will backtrack trying different permutations of the quantifiers, but they will all fail to match. No match can be found.

But the regex `^(?'open'o)+(?'-open'c)+$` still matches `ooC`. The matching process is again the same until the balancing group has matched the first `c` and left the group ‘open’ with the first `o` as its only capture. The quantifier makes the engine attempt the balancing group again. The engine again finds that the subtracted group “open” captured something. The regex enters the balancing group, leaving the group “open” without any matches. But now, `c` fails to match because the regex engine has reached the end of the string.

The regex engine must now backtrack out of the balancing group. When backtracking a balancing group, .NET also backtracks the subtraction. Since the capture of the first `o` was subtracted from “open” when entering the balancing group, this capture is now restored while backtracking out of the balancing group. The repeated group `(?'-open'c)+` is now reduced to a single iteration. But the quantifier is fine with that, as `+` means “once or more” as it always does. Still at the end of the string, the regex engine reaches `$` in the regex, which matches. The whole string `ooC` is returned as the overall match. `Match.Groups['open'].Captures` will hold the first `o` in the string as the only item in the `CaptureCollection`. That’s because, after backtracking, the second `o` was subtracted from the group, but the first `o` was not.

To make sure the regex matches `oc` and `ooCC` but not `ooC`, we need to check that the group “open” has no captures left when the matching process reaches the end of the regex. We can do this with a conditional. `(?(open)(?!))` is a conditional that checks whether the group “open” matched something. In .NET, having matched something means still having captures on the stack that weren’t backtracked or subtracted. If the group has captured something, the “if” part of the conditional is evaluated. In this case that is the empty negative lookahead `(?!)`. The empty string inside this lookahead always matches. Because the lookahead is negative, this causes the lookahead to always fail. Thus the conditional always fails if the group has captured something. If the group has not captured anything, the “else” part of the conditional is evaluated. In this case there is no “else” part. This means that the conditional always succeeds if the group has not captured something. This makes `(?(open)(?!))` a proper test to verify that the group “open” has no captures left.

The regex `^(?'open'o)+(?'-open'c)+(?(open)(?!))$` fails to match `ooC`. When `c` fails to match because the regex engine has reached the end of the string, the engine backtracks out of the balancing group,

leaving “open” with a single capture. The regex engine now reaches the conditional, which fails to match. The regex engine will backtrack trying different permutations of the quantifiers, but they will all fail to match. No match can be found.

The regex `^(?'open'o)+(?'-open'c)+(?(open)(?!))$` does match `oocc`. After `(?'-open'c)+` has matched `cc`, the regex engine cannot enter the balancing group a third time, because “open” has no captures left. The engine advances to the conditional. The conditional succeeds because “open” has no captures left and the conditional does not have an “else” part. Now `$` matches at the end of the string.

Matching Balanced Constructs

`^(?:('open'o)+(''-open'c)+)+(?(open)(?!))$` wraps the capturing group and the balancing group in a non-capturing group that is also repeated. This regex matches any string like `ooocoooccccccoc` that contains any number of perfectly balanced o’s and c’s, with any number of pairs in sequence, nested to any depth. The balancing group makes sure that the regex never matches a string that has more c’s at any point in the string than it has o’s to the left of that point. The conditional at the end, which must remain outside the repeated group, makes sure that the regex never matches a string that has more o’s than c’s.

`^(?>('open'o)+(''-open'c)+)+(?(open)(?!))$` optimizes the previous regex by using an atomic group instead of the non-capturing group. The atomic group, which is also non-capturing, eliminates nearly all backtracking when the regular expression cannot find a match, which can greatly increase performance when used on long strings with lots of o’s and c’s but that aren’t properly balanced at the end. The atomic group does not change how the regex matches strings that do have balanced o’s and c’s.

`^m*(?>('open'o)m*)+(?>(''-open'c)m*)+(?(open)(?!))$` allows any number of letters `m` anywhere in the string, while still requiring all o’s and c’s to be balanced. `m*` at the start of the regex allows any number of m’s before the first o. `(?'open'o)+` was changed into `(?>('open'o)m*)+` to allow any number of m’s after each o. Similarly, `(?'-open'c)+` was changed into `(?>(''-open'c)m*)+` to allow any number of m’s after each c.

This is the generic solution for matching balanced constructs using .NET’s balancing groups or capturing group subtraction feature. You can replace `o`, `m`, and `c` with any regular expression, as long as no two of these three can match the same text.

`^[^()]*(>(>('open'\([^\()]*+(>(''-open'\))^[^()]*+)+(?(open)(?!))$` applies this technique to match a string in which all parentheses are perfectly balanced.

Backreferences To Subtracted Groups

You can use backreferences to groups that have their matches subtracted by a balancing group. The backreference matches the group’s most recent match that wasn’t backtracked or subtracted. The regex `(?'x'[ab]){2}(?'-x')\k'x'` matches `aaa`, `aba`, `bab`, or `bbb`. It does not match `aab`, `abb`, `baa`, or `bba`. The first and third letters of the string have to be the same.

Let’s see how `(?'x'[ab]){2}(?'-x')\k'x'` matches `aba`. The first iteration of `(?'x'[ab])` captures `a`. The second iteration captures `b`. Now the regex engine reaches the balancing group `(?'-x')`. It checks whether the group “x” has matched, which it has. The engine enters the balancing group, subtracting the

match **b** from the stack of group “x”. There are no regex tokens inside the balancing group. It matches without advancing through the string. Now the regex engine reaches the backreference `\k'x'`. The match at the top of the stack of group “x” is **a**. The next character in the string is also an **a** which the backreference matches. `aba` is found as an overall match.

When you apply this regex to `abb`, the matching process is the same, except that the backreference fails to match the second **b** in the string. Since the regex has no other permutations that the regex engine can try, the match attempt fails.

Matching Palindromes

`^(?'letter'[a-z])+[a-z]?(?:\k'letter'(?'-letter'))+(?(letter)(?!))$` matches palindrome words of any length. This regular expression takes advantage of the fact that backreferences and capturing group subtraction work well together. It also uses an empty balancing group as the regex in the previous section.

Let’s see how this regex matches the palindrome `radar`. `^` matches at the start of the string. Then `(?'letter'[a-z])+` iterates five times. The group “letter” ends up with five matches on its stack: **r**, **a**, **d**, **a**, and **r**. The regex engine is now at the end of the string and at `[a-z]?` in the regex. It doesn’t match, but that’s fine, because the quantifier makes it optional. The engine now reaches the backreference `\k'letter'`. The group “letter” has **r** at the top of its stack. This fails to match the void after the end of the string.

The regex engine backtracks. `(?'letter'[a-z])+` is reduced to four iterations, leaving **r**, **a**, **d**, and **a** on the stack of the group “letter”. `[a-z]?` matches **r**. The backreference again fails to match the void after the end of the string. The engine backtracks, forcing `[a-z]?` to give up its match. Now “letter” has **a** at the top of its stack. This causes the backreference to fail to match **r**.

More backtracking follows. `(?'letter'[a-z])+` is reduced to three iterations, leaving **d** at the top of the stack of the group “letter”. The engine again proceeds with `[a-z]?`. It fails again because there is no **d** for the backreference to match.

Backtracking once more, the capturing stack of group “letter” is reduced to **r** and **a**. Now the tide turns. `[a-z]?` matches **d**. The backreference matches **a** which is the most recent match of the group “letter” that wasn’t backtracked. The engine now reaches the empty balancing group `(?'-letter')`. This matches, because the group “letter” has a match **a** to subtract.

The backreference and balancing group are inside a repeated non-capturing group, so the engine tries them again. The backreference matches **r** and the balancing group subtracts it from “letter”’s stack, leaving the capturing group without any matches. Iterating once more, the backreference fails, because the group “letter” has no matches left on its stack. This makes the group act as a non-participating group. Backreferences to non-participating groups always fail in .NET, as they do in most regex flavors.

`(?:\k'letter'(?'-letter'))+` has successfully matched two iterations. Now, the conditional `(?(letter)(?!))` succeeds because the group “letter” has no matches left. The anchor `$` also matches. The palindrome `radar` has been matched.

31. Regular Expression Recursion

Perl 5.10, PCRE 4.0, Ruby 2.0, and all later versions of these three, support regular expression recursion. Perl uses the syntax `(?R)` with `(?0)` as a synonym. Ruby 2.0 uses `\g<0>`. PCRE supports all three as of version 7.7. Earlier versions supported only the Perl syntax (which Perl actually copied from PCRE). Recent versions of Delphi, PHP, and R also support all three, as their regex functions are based on PCRE. JGsoft V2 also supports all variations of regex recursion.

While Ruby 1.9 does not have any syntax for regex recursion, it does support capturing group recursion. So you could recurse the whole regex in Ruby 1.9 if you wrap the whole regex in a capturing group. .NET does not support recursion, but it supports balancing groups that can be used instead of recursion to match balanced constructs.

As we'll see later, there are differences in how Perl, PCRE, and Ruby deal with backreferences and backtracking during recursion. While they copied each other's syntax, they did not copy each other's behavior. JGsoft V2, however, copied their syntax and their behavior. So JGsoft V2 has three different ways of doing regex recursion, which you choose by using a different syntax. But these differences do not come into play in the basic example on this page.

Boost 1.42 copied the syntax from Perl. But its implementation is marred by bugs. Boost 1.60 attempted to fix the behavior of quantifiers on recursion, but it's still quite different from other flavors and incompatible with previous versions of Boost. Boost 1.64 finally stopped crashing upon infinite recursion. But recursion of the whole regex still attempts only the first alternative.

Simple Recursion

The regexes `a(?R)?z`, `a(?0)?z`, and `a\g<0>?z` all match one or more letters `a` followed by exactly the same number of letters `z`. Since these regexes are functionally identical, we'll use the syntax with `R` for recursion to see how this regex matches the string `aaazzz`.

First, `a` matches the first `a` in the string. Then the regex engine reaches `(?R)`. This tells the engine to attempt the whole regex again at the present position in the string. Now, `a` matches the second `a` in the string. The engine reaches `(?R)` again. On the second recursion, `a` matches the third `a`. On the third recursion, `a` fails to match the first `z` in the string. This causes `(?R)` to fail. But the regex uses a quantifier to make `(?R)` optional. So the engine continues with `z` which matches the first `z` in the string.

Now, the regex engine has reached the end of the regex. But since it's two levels deep in recursion, it hasn't found an overall match yet. It only has found a match for `(?R)`. Exiting the recursion after a successful match, the engine also reaches `z`. It now matches the second `z` in the string. The engine is still one level deep in recursion, from which it exits with a successful match. Finally, `z` matches the third `z` in the string. The engine is again at the end of the regex. This time, it's not inside any recursion. Thus, it returns `aaazzz` as the overall regex match.

Matching Balanced Constructs

The main purpose of recursion is to match balanced constructs or nested constructs. The generic regex is `b(?:m|(?R))*e` where `b` is what begins the construct, `m` is what can occur in the middle of the construct,

and `e` is what can occur at the end of the construct. For correct results, no two of `b`, `m`, and `e` should be able to match the same text. You can use an atomic group instead of the non-capturing group for improved performance: `b(?:>m|(?:R))*e`.

A common real-world use is to match a balanced set of parentheses. `\((?>[^()]|(?:R))*\)` matches a single pair of parentheses with any text in between, including an unlimited number of parentheses, as long as they are all properly paired. If the subject string contains unbalanced parentheses, then the first regex match is the leftmost pair of balanced parentheses, which may occur after unbalanced opening parentheses. If you want a regex that does not find any matches in a string that contains unbalanced parentheses, then you need to use a subroutine call instead of recursion. If you want to find a sequence of multiple pairs of balanced parentheses as a single match, then you also need a subroutine call.

Recursion with Alternation

If what may appear in the middle of the balanced construct may also appear on its own without the beginning and ending parts then the generic regex is `b(?:R)*e|m`. Again, `b`, `m`, and `e` all need to be mutually exclusive. `\((?:R)*\)|[^()]+` matches a pair of balanced parentheses like the regex in the previous section. But it also matches any text that does not contain any parentheses at all.

This regular expression does not work correctly in Boost. If a regex has alternation that is not inside a group then recursion of the whole regex in Boost only attempts the first alternative. So `\((?:R)*\)|[^()]+` in Boost matches any number of balanced parentheses nested arbitrarily deep with no text in between, or any text that does not contain any parentheses at all. If you flip the alternatives then `[^()]+|\((?:R)*\)` in Boost matches any text without any parentheses or a single pair of parentheses with any text without parentheses in between. In all other flavors these two regexes find the same matches.

The solution for Boost is to put the alternation inside a group. `(?:\((?:R)*\)|[^()]+)` and `(?:[^()]+|\((?:R)*\))` find the same matches in all flavors discussed in this tutorial that support recursion.

32. Regular Expression Subroutines

Perl 5.10, PCRE 4.0, and Ruby 1.9 support regular expression subroutine calls. These are very similar to regular expression recursion. Instead of matching the entire regular expression again, a subroutine call only matches the regular expression inside a capturing group. You can make a subroutine call to any capturing group from anywhere in the regex. If you place a call inside the group that it calls, you'll have a recursive capturing group.

As with regex recursion, there is a wide variety of syntax that you can use for exactly the same thing. Perl uses `(?1)` to call a numbered group, `(?+1)` to call the next group, `(?-1)` to call the preceding group, and `(?&name)` to call a named group. You can use all of these to reference the same group. `(?+1)(?'name'[abc])(?1)(?-1)(?&name)` matches a string that is five letters long and consists only of the first three letters of the alphabet. This regex is exactly the same as `[abc](?'name'[abc])[abc][abc][abc]`.

PCRE was the first regex engine to support subroutine calls. `(?P<name>[abc])(?1)(?P>name)` matches three letters like `(?P<name>[abc])[abc][abc]` does. `(?1)` is a call to a numbered group and `(?P>name)` is a call to a named group. The latter is called the “Python syntax” in the PCRE man page. While this syntax mimics the syntax Python uses for named capturing groups, it is a PCRE invention. Python does not support subroutine calls or recursion. PCRE 7.2 added `(?+1)` and `(?-1)` for relative calls. PCRE 7.7 adds all the syntax used by Perl 5.10 and Ruby 2.0. Recent versions of PHP, Delphi, and R also support all this syntax, as their regex functions are based on PCRE.

The syntax used by Ruby 1.9 and later looks more like that of backreferences. `\g<1>` and `\g'1'` call a numbered group, `\g<name>` and `\g'name'` call a named group, while `\g<-1>` and `\g'-1'` call the preceding group. Ruby 2.0 adds `\g<+1>` and `\g'+1'` to call the next group. `\g<+1>(?'name'[abc])\g<1>\g<-1>\g<name>` and `\g'+1'(?'name'[abc])\g'1'\g'-1'\g'name'` match the same 5-letter string in Ruby 2.0 as the Perl example does in Perl. The syntax with angle brackets and with quotes can be used interchangeably.

JGsoft V2 supports all three sets of syntax. As we'll see later, there are differences in how Perl, PCRE, and Ruby deal with capturing, backreferences, and backtracking during subroutine calls. While they copied each other's syntax, they did not copy each other's behavior. JGsoft V2, however, copied their syntax and their behavior. So JGsoft V2 has three different ways of doing regex recursion, which you choose by using a different syntax. But these differences do not come into play in the basic examples on this page.

Boost 1.42 copied the syntax from Perl but its implementation is marred by bugs, which are still not all fixed in version 1.62. Most significantly, quantifiers other than `*` or `{0,}` cause subroutine calls to misbehave. This is partially fixed in Boost 1.60 which correctly handles `?` and `{0,1}` too.

Boost does not support the Ruby syntax for subroutine calls. In Boost `\g<1>` is a backreference—not a subroutine call—to capturing group 1. So `([ab])\g<1>` can match `aa` and `bb` but not `ab` or `ba`. In Ruby the same regex would match all four strings. No other flavor discussed in this tutorial uses this syntax for backreferences.

Matching Balanced Constructs

Recursion into a capturing group is a more flexible way of matching balanced constructs than recursion of the whole regex. We can wrap the regex in a capturing group, recurse into the capturing group instead of the whole regex, and add anchors outside the capturing group. `\A(b(?m|(?1))*e)\z` is the generic regex for checking that a string consists entirely of a correctly balanced construct. Again, `b` is what begins the construct, `m` is what can occur in the middle of the construct, and `e` is what can occur at the end of the construct. For correct results, no two of `b`, `m`, and `e` should be able to match the same text. You can use an atomic group instead of the non-capturing group for improved performance: `\A(b(?>m|(?1))*e)\z`.

Similarly, `\Ao*(b(?m|(?1))*eo*)+\z` and the optimized `\Ao*+(b(?>m|(?1))*eo*+)+\z` match a string that consists of nothing but a sequence of one or more correctly balanced constructs, with possibly other text in between. Here, `o` is what can occur outside the balanced constructs. It will often be the same as `m`. `o` should not be able to match the same text as `b` or `e`.

`\A(\((?>[^\(\)]|(?1))*\))\z` matches a string that consists of nothing but a correctly balanced pair of parentheses, possibly with text between them. `\A[^\(\)]*+(\((?>[^\(\)]|(?1))*\)[^\(\)]*+)+\z`.

Matching The Same Construct More Than Once

A regex that needs to match the same kind of construct (but not the exact same text) more than once in different parts of the regex can be shorter and more concise when using subroutine calls. Suppose you need a regex to match patient records like these:

```
Name: John Doe
Born: 17-Jan-1964
Admitted: 30-Jul-2013
Released: 3-Aug-2013
```

Further suppose that you need to match the date format rather accurately so the regex can filter out valid records, leaving invalid records for human inspection. In most regex flavors you could easily do this with this regex, using free-spacing syntax:

```
^Name: \ (.*)\r?\n
Born: \ (? : 3 [01] | [12] [0-9] | [1-9] )
      - (? : Jan | Feb | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov | Dec )
      - (? : 19 | 20 ) [0-9] [0-9] \r?\n
Admitted: \ (? : 3 [01] | [12] [0-9] | [1-9] )
           - (? : Jan | Feb | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov | Dec )
           - (? : 19 | 20 ) [0-9] [0-9] \r?\n
Released: \ (? : 3 [01] | [12] [0-9] | [1-9] )
           - (? : Jan | Feb | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov | Dec )
           - (? : 19 | 20 ) [0-9] [0-9] $
```

With subroutine calls you can make this regex much shorter, easier to read, and easier to maintain:

```
^Name: \ (.*)\r?\n
Born: \ (? 'date' (? : 3 [01] | [12] [0-9] | [1-9] )
      - (? : Jan | Feb | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov | Dec )
      - (? : 19 | 20 ) [0-9] [0-9] ) \r?\n
```



```
Admitted: \ \g'date'\r?\n
Released: \ \g'date'$
```

Separate Subroutine Definitions

In Perl, PCRE, and JGsoft V2, you can take this one step further using the special DEFINE group: `(?(DEFINE)(?'subroutine'regex))`. While this looks like a conditional that references the non-existent group DEFINE containing a single named group “subroutine”, the DEFINE group is a special syntax. The fixed text `(?(DEFINE)` opens the group. A parenthesis closes the group. This special group tells the regex engine to ignore its contents, other than to parse it for named and numbered capturing groups. You can put as many capturing groups inside the DEFINE group as you like. The DEFINE group itself never matches anything, and never fails to match. It is completely ignored. The regex `foo(?(DEFINE)(?'subroutine'skipped))bar` matches `foobar`. The DEFINE group is completely superfluous in this regex, as there are no calls to any of the groups inside it.

With a DEFINE group, our regex becomes:

```
(?(DEFINE)(?'date'(?:(?:3[01]||[12][0-9]||[1-9])
- (?:(?:Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)
- (?:(?:19|20)[0-9][0-9]))))
^Name: \ (.*)\r?\n
Born: \ (?P>date)\r?\n
Admitted: \ (?P>date)\r?\n
Released: \ (?P>date)$
```

Quantifiers On Subroutine Calls

Quantifiers on subroutine calls work just like a quantifier on recursion. The call is repeated as many times in sequence as needed to satisfy the quantifier. `([abc])(?1){3}` matches `abcb` and any other combination of four-letter combination of the first three letters of the alphabet. First the group matches once, and then the call matches three times. This regex is equivalent to `([abc])[abc]{3}`.

Quantifiers on the group are ignored by the subroutine call. `([abc]){3}(?1)` also matches `abcb`. First, the group matches three times, because it has a quantifier. Then the subroutine call matches once, because it has no quantifier. `([abc]){3}(?1){3}` matches six letters, such as `abbcab`, because now both the group and the call are repeated 3 times. These two regexes are equivalent to `([abc]){3}[abc]` and `([abc]){3}[abc]{3}`.

While Ruby does not support subroutine definition groups, it does support subroutine calls to groups that are repeated zero times. `(a){0}\g<1>{3}` matches `aaa`. The group itself is skipped because it is repeated zero times. Then the subroutine call matches three times, according to its quantifier. This also works in PCRE 7.7 and later. It doesn't work (reliably) in older versions of PCRE or in any version of Perl because of bugs.

The Ruby version of the patient record example can be further cleaned up as:

```
(?'date'(?:(?:3[01]||[12][0-9]||[1-9])
- (?:(?:Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)
- (?:(?:19|20)[0-9][0-9])){0}
^Name: \ (.*)\r?\n
```


Born: \ \g'date'\r?\n
Admitted: \ \g'date'\r?\n
Released: \ \g'date'\$

33. Infinite Recursion

Regular expressions such as `(?R)?z` or `a?(?R)?z` or `a|(?R)z` that use recursion without having anything that must be matched in front of the recursion can result in infinite recursion. If the regex engine reaches the recursion without having advanced through the text then the next recursion will again reach the recursion without having advanced through the text. With the first regex this happens immediately at the start of the match attempt. With the other two this happens as soon as there are no further letters `a` to be matched.

JGsoft V2 and Boost 1.64 treat the first two regexes as a syntax error because they always lead to infinite recursion. They allow the third regex because that one can match `a`. Ruby 1.9 and later, all versions of PCRE, and PCRE2 10.20 and prior treat all three forms of potential infinite recursion as a syntax error. Perl, PCRE2 10.21 and later, and Boost 1.63 and prior allow all three forms.

Circular Infinite Subroutine Calls

Subroutine calls can also lead to infinite recursion. All flavors handle the potentially infinite recursion in `((?1)?z)` or `(a?(?1)?z)` or `(a|(?1)z)` in the same way as they handle potentially infinite recursion of the entire regex.

But subroutine calls that are not recursive by themselves may end up being recursive if the group they call has another subroutine call that calls a parent group of the first subroutine call. When subroutine calls are forced to go around in a circle that too leads to infinite recursion. Detecting such circular calls when compiling a regex is more complicated than checking for straight infinite recursion. Only JGsoft V2 and Ruby 1.9 and later are able to detect this and treat it as a syntax error. All other flavors allow these regexes.

Errors and Crashes

When infinite recursion does occur, whether it's straight recursion or subroutine calls going in circles, JGsoft V2, Perl, and PCRE2 treat it as a matching error that aborts the entire match attempt. Boost 1.64 handles this by not attempting the recursion and acting as if the recursion failed. If the recursion is optional then Boost 1.64 may find matches where other flavors throw errors.

Boost 1.63 and prior and PCRE 8.12 and prior crash when infinite recursion occurs. This also affects Delphi up to version XE6 and PHP up to version 5.4.8 as they are based on older PCRE versions.

Endless Recursion

A regex such as `a(?R)z` that has a recursion token that is not optional and is not have an alternative without the same recursion leads to endless recursion. Such a regular expression can never find a match. When `a` matches the regex engine attempts the recursion. If it can match another `a` then it has to attempt the recursion again. Eventually `a` will run out of letters to match. The recursion then fails. Because it's not optional the regex fails to match.

JGsoft V2 and Ruby detect this situation when compiling your regular expression. They flag endless recursion as a syntax error. Perl, PCRE, PCRE2, and Boost do not detect endless recursion. They simply go through the matching process which finds no matches.

34. Quantifiers On Recursion

The introduction to recursion shows how `a(?:R)?z` matches `aaazzz`. The quantifier `?` makes the preceding token optional. In other words, it repeats the token between zero or one times. In `a(?:R)?z` the `(?:R)` is made optional by the `?` that follows it. You may wonder why the regex attempted the recursion three times, instead of once or not at all.

The reason is that upon recursion, the regex engine takes a fresh start in attempting the whole regex. All quantifiers and alternatives behave as if the matching process prior to the recursion had never happened at all, other than that the engine advanced through the string. The regex engine restores the states of all quantifiers and alternatives when it exits from a recursion, whether the recursion matched or failed. Basically, the matching process continues normally as if the recursion never happened, other than that the engine advanced through the string.

If you're familiar with procedural programming languages, regex recursion is basically a recursive function call and the quantifiers are local variables in the function. Each recursion of the function gets its own set of local variables that don't affect and aren't affected by the same local variables in recursions higher up the stack. Quantifiers on recursion work this way in all flavors, except Boost.

Let's see how `a(?:R){3}z|q` behaves (Boost excepted). The simplest possible match is `q`, found by the second alternative in the regex.

The simplest match in which the first alternative matches is `aqqqz`. After `a` is matches, the regex engine begins a recursion. `a` fails to match `q`. Still inside the recursion, the engine attempts the second alternative. `q` matches `q`. The engine exits from the recursion with a successful match. The engine now notes that the quantifier `{3}` has successfully repeated once. It needs two more repetitions, so the engine begins another recursion. It again matches `q`. On the third iteration of the quantifier, the third recursion matches `q`. Finally, `z` matches `z` and an overall match is found.

This regex does not match `aqqz` or `aqqqqz`. `aqqz` fails because during the third iteration of the quantifier, the recursion fails to match `z`. `aqqqqz` fails because after `a(?:R){3}` has matched `aqqq`, `z` fails to match the fourth `q`.

The regex can match longer strings such as `aaqqqqzqz`. With this string, during the second iteration of the quantifier, the recursion matches `aqqqz`. Since each recursion tracks the quantifier separately, the recursion needs three consecutive recursions of its own to satisfy its own instance of the quantifier. This can lead to arbitrarily long matches such as `aaaqqqqqzzzqqqqzqzqqqaaqqqzqqzzz`.

How Boost Handles Quantifiers on Recursion

Boost has its own ideas about how quantifiers should work on recursion. Recursion only works the same in Boost as in other flavors if the recursion operator either has no quantifier at all or if it has `*` as its quantifier. Any other quantifier may lead to very different matches (or lack thereof) in Boost 1.59 or prior versus Boost 1.60 and later versus other regex flavors. Boost 1.60 attempted to fix some of the differences between Boost and other flavors but it only resulted in a different incompatible behavior.

In Boost 1.59 and prior, quantifiers on recursion count both iteration and recursion throughout the entire recursion stack. So possible matches for `a(?:R){3}z|q` in Boost 1.59 include `aaaazzzz`, `aaaqzzz`, `aaqqzz`,

aaqzqz, and aqaqzzz. In all these matches the number of recursions and iterations add up to 3. No other flavor would find these matches because they require 3 iterations during each recursion. So other flavors can match things like aaqqqzaqqqzaqqqzz or aqqqaqqqzz. Boost 1.59 would match only aqqqz within these strings.

Boost 1.60 attempts to iterate quantifiers at each recursion level like other flavors, but does so incorrectly. Any quantifier that makes the recursion optional allows for infinite repetition. So Boost 1.60 and later treat a(?R)?z the same as a(?R)*z. While this fixes the problem that a(?R)?z could not match aaazzz entirely in Boost 1.59, it also allows matches such as aazzz that other flavors won't find with this regex. If the quantifier is not optional, then Boost 1.60 only allows it to match during the first recursion. So a(?R){3}z|q could only ever match q or aqqqz.

Boost's issues with quantifiers on recursion also affect quantifiers on parent groups of the recursion token. They also affect quantifiers on subroutine calls and quantifiers on groups that contain a subroutine call to a parent group of the group with the quantifier.

Quantifiers on Other Tokens in The Recursion

Quantifiers on other tokens in the regex behave normally during recursion. They track their iterations separately at each recursion. So a{2}(?R)z|q matches aaqz, aaaaqzz, aaaaaaqzzz, and so on. a has to match twice during each recursion.

Quantifiers like these that are inside the recursion but do not repeat the recursion itself do work correctly in Boost.

35. Subroutine Calls May or May Not Capture

This tutorial introduced regular expression subroutines with this example that we want to match accurately:

```
Name: John Doe
Born: 17-Jan-1964
Admitted: 30-Jul-2013
Released: 3-Aug-2013
```

In Ruby or PCRE, we can use this regular expression:

```
^Name: \ (.*)\n
Born: \ (? 'date' (? :3 [01] || [12] [0-9] || [1-9] )
      - (? :Jan || Feb || Mar || Apr || May || Jun || Jul || Aug || Sep || Oct || Nov || Dec )
      - (? :19 || 20) [0-9] [0-9] )\n
Admitted: \ \g'date'\n
Released: \ \g'date'$
```

Perl needs slightly different syntax, which also works in PCRE:

```
^Name: \ (.*)\n
Born: \ (? 'date' (? :3 [01] || [12] [0-9] || [1-9] )
      - (? :Jan || Feb || Mar || Apr || May || Jun || Jul || Aug || Sep || Oct || Nov || Dec )
      - (? :19 || 20) [0-9] [0-9] )\n
Admitted: \ (?&date)\n
Released: \ (?&date)$
```

Unfortunately, there are differences in how these three regex flavors treat subroutine calls beyond their syntax. First of all, in Ruby a subroutine call makes the capturing group store the text matched during the subroutine call. In Perl, PCRE, and Boost a subroutine call does not affect the group that is called.

When the Ruby solution matches the sample above, retrieving the contents of the capturing group “date” will get you `3-Aug-2013` which was matched by the last subroutine call to that group. When the Perl solution matches the same, retrieving `${date}` will get you `17-Jan-1964`. In Perl, the subroutine calls did not capture anything at all. But the “Born” date was matched with a normal named capturing group which stored the text that it matched normally. Any subroutine calls to the group don’t change that. PCRE behaves as Perl in this case, even when you use the Ruby syntax with PCRE.

JGsoft V2 behaves like Ruby when you use the first regular expression. You can remember this by the fact that the `\g` syntax is a Ruby invention, later copied by PCRE. JGsoft V2 behaves like Perl when you use the second regular expression. You can remember this by the fact that Perl uses ampersands for subroutine calls in procedural code too.

If you want to extract the dates from the match, the best solution is to add another capturing group for each date. Then you can ignore the text stored by the “date” group and this particular difference between these flavors. In Ruby or PCRE:

```
^Name: \ (.*)\n
Born: \ (? 'born' (? 'date' (? :3 [01] || [12] [0-9] || [1-9] )
      - (? :Jan || Feb || Mar || Apr || May || Jun || Jul || Aug || Sep || Oct || Nov || Dec )
      - (? :19 || 20) [0-9] [0-9] ) )\n
```

```
Admitted:\ (? 'admitted' \g'date' ) \n
Released:\ (? 'released' \g'date' ) $
```

Perl needs slightly different syntax, which also works in PCRE:

```
^Name:\ (.*) \n
Born:\ (? 'born' (? 'date' (? : 3 [01] [12] [0-9] [1-9] )
- (? : Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec )
- (? : 19 20 [0-9] [0-9] ) ) \n
Admitted:\ (? 'admitted' (?&date) ) \n
Released:\ (? 'released' (?&date) ) $
```

Capturing Groups Inside Recursion or Subroutine Calls

There are further differences between Perl, PCRE, and Ruby when your regex makes a subroutine call or recursive call to a capturing group that contains other capturing groups. The same issues also affect recursion of the whole regular expression if it contains any capturing groups. For the remainder of this topic, the term “recursion” applies equally to recursion of the whole regex, recursion into a capturing group, or a subroutine call to a capturing group.

PCRE and Boost back up and restores capturing groups when entering and exiting recursion. When the regex engine enters recursion, it internally makes a copy of all capturing groups. This does not affect the capturing groups. Backreferences inside the recursion match text captured prior to the recursion unless and until the group they reference captures something during the recursion. After the recursion, all capturing groups are replaced with the internal copy that was made at the start of the recursion. Text captured during the recursion is discarded. This means you cannot use capturing groups to retrieve parts of the text that were matched during recursion.

Perl 5.10, the first version to have recursion, through version 5.18, isolated capturing groups between each level of recursion. When Perl 5.10’s regex engine enters recursion, all capturing groups appear as they have not participated in the match yet. Initially, all backreferences will fail. During the recursion, capturing groups capture as normal. Backreferences match text captured during the same recursion as normal. When the regex engine exits from the recursion, all capturing groups revert to the state they were in prior to the recursion. Perl 5.20 changed Perl’s behavior to back up and restore capturing groups the way PCRE does.

For most practical purposes, however, you’ll only use backreferences after their corresponding capturing groups. Then the difference between the way Perl 5.10 through 5.18 deal with capturing groups during recursion and the way PCRE and later versions of Perl do is academic.

Ruby’s behavior is completely different. When Ruby’s regex engine enters or exits recursion, it makes no changes to the text stored by capturing groups at all. Backreferences match the text stored by the capturing group during the group’s most recent match, irrespective of any recursion that may have happened. After an overall match is found, each capturing group still stores the text of its most recent match, even if that was during a recursion. This means you can use capturing groups to retrieve part of the text matched during the last recursion.

JGsoft V2 behaves like Ruby when you use the `\g` syntax borrowed from Ruby. It behaves like Perl 5.20 and PCRE when you use any other syntax.

Odd Length Palindromes in Perl and PCRE

In Perl and PCRE you can use `\b(?:'word'(?'letter'[a-z])(?&word)\k'letter'|[a-z])\b` to match palindrome words such as `a`, `dad`, `radar`, `racecar`, and `redivider`. This regex only matches palindrome words that are an odd number of letters long. This covers most palindrome words in English. To extend the regex to also handle palindrome words that are an even number of characters long we have to worry about differences in how Perl and PCRE backtrack after a failed recursion attempt which is discussed later in this tutorial. We gloss over these differences here because they only come into play when the subject string is not a palindrome and no match can be found.

Let's see how this regex matches `radar`. The word boundary `\b` matches at the start of the string. The regex engine enters the two capturing groups. `[a-z]` matches `r` which is then stored in the capturing group "letter". Now the regex engine enters the first recursion of the group "word". At this point, Perl forgets that the "letter" group matched `r`. PCRE does not. But this does not matter. `(?'letter'[a-z])` matches and captures `a`. The regex enters the second recursion of the group "word". `(?'letter'[a-z])` captures `d`. During the next two recursions, the group captures `a` and `r`. The fifth recursion fails because there are no characters left in the string for `[a-z]` to match. The regex engine must backtrack.

Because `(?&word)` failed to match, `(?'letter'[a-z])` must give up its match. The group reverts to `a`, which was the text the group held at the start of the recursion. (It becomes empty in Perl 5.18 and prior.) Again, this does not matter because the regex engine must now try the second alternative inside the group "word", which contains no backreferences. The second `[a-z]` matches the final `r` in the string. The engine now exits from a successful recursion. The text stored by the group "letter" is restored to what it had captured prior to entering the fourth recursion, which is `a`.

After matching `(?&word)` the engine reaches `\k'letter'`. The backreference fails because the regex engine has already reached the end of the subject string. So it backtracks once more, making the capturing group give up the `a`. The second alternative now matches the `a`. The regex engine exits from the third recursion. The group "letter" is restored to the `d` matched during the second recursion.

The regex engine has again matched `(?&word)`. The backreference fails again because the group stores `d` while the next character in the string is `r`. Backtracking again, the second alternative matches `d` and the group is restored to the `a` matched during the first recursion.

Now, `\k'letter'` matches the second `a` in the string. That's because the regex engine has arrived back at the first recursion during which the capturing group matched the first `a`. The regex engine exits the first recursion. The capturing group is restored to the `r` which it matched prior to the first recursion.

Finally, the backreference matches the second `r`. Since the engine is not inside any recursion any more, it proceeds with the remainder of the regex after the group. `\b` matches at the end of the string. The end of the regex is reached and `radar` is returned as the overall match. If you query the groups "word" and "letter" after the match you'll get `radar` and `r`. That's the text matched by these groups outside of all recursion.

Why This Regex Does Not Work in Ruby

To match palindromes this way in Ruby, you need to use a special backreference that specifies a recursion level. If you use a normal backreference as in `\b(?:'word'(?'letter'[a-z])\g'word'\k'letter'|[a-z])\b`, Ruby will not complain. But it will not match palindromes longer than three letters either. Instead this regex matches things like `a`, `dad`, `radaa`, `raceccc`, and `rediviii`.

Let's see why this regex does not match `radar` in Ruby. Ruby starts out like Perl and PCRE, entering the recursions until there are no characters left in the string for `[a-z]` to match.

Because `\g'word'` failed to match, `(?'letter'[a-z])` must give up its match. Ruby reverts it to `a`, which was the text the group most recently matched. The second `[a-z]` matches the final `r` in the string. The engine now exits from a successful recursion. The group "letter" continues to hold its most recent match `a`.

After matching `\g'word'` the engine reaches `\k'letter'`. The backreference fails because the regex engine has already reached the end of the subject string. So it backtracks once more, reverting the group to the previously matched `d`. The second alternative now matches the `a`. The regex engine exits from the third recursion.

The regex engine has again matched `\g'word'`. The backreference fails again because the group stores `d` while the next character in the string is `r`. Backtracking again, the group reverts to `a` and the second alternative matches `d`.

Now, `\k'letter'` matches the second `a` in the string. The regex engine exits the first recursion which successfully matched `ada`. The capturing group continues to hold `a` which is its most recent match that wasn't backtracked.

The regex engine is now at the last character in the string. This character is `r`. The backreference fails because the group still holds `a`. The engine can backtrack once more, forcing `(?'letter'[a-z])\g'word'\k'letter'` to give up the `rada` it matched so far. The regex engine is now back at the start of the string. It can still try the second alternative in the group. This matches the first `r` in the string. Since the engine is not inside any recursion any more, it proceeds with the remainder of the regex after the group. `\b` fails to match after the first `r`. The regex engine has no further permutations to try. The match attempt has failed.

If the subject string is `radaa`, Ruby's engine goes through nearly the same matching process as described above. Only the events described in the last paragraph change. When the regex engine reaches the last character in the string, that character is now `a`. This time, the backreference matches. Since the engine is not inside any recursion any more, it proceeds with the remainder of the regex after the group. `\b` matches at the end of the string. The end of the regex is reached and `radaa` is returned as the overall match. If you query the groups "word" and "letter" after the match you'll get `radaa` and `a`. Those are the most recent matches of these groups that weren't backtracked.

Basically, in Ruby this regex matches any word that is an odd number of letters long and in which all the characters to the right of the middle letter are identical to the character just to the left of the middle letter. That's because Ruby only restores capturing groups when they backtrack, but not when it exits from recursion.

The solution, specific to Ruby, is to use a backreference that specifies a recursion level instead of the normal backreference used in the regex on this page.

36. Backreferences That Specify a Recursion Level

Earlier topics in this tutorial explain regular expression recursion and regular expression subroutines. In this topic the word “recursion” refers to recursion of the whole regex, recursion of capturing groups, and subroutine calls to capturing groups. The previous topic also explained that these features handle capturing groups differently in Ruby than they do in Perl and PCRE.

Perl, PCRE, and Boost restore capturing groups when they exit from recursion. This means that backreferences in Perl, PCRE, and Boost match the same text that was matched by the capturing group at the same recursion level. This makes it possible to do things like matching palindromes.

Ruby does not restore capturing groups when it exits from recursion. Normal backreferences match the text that is the same as the most recent match of the capturing group that was not backtracked, regardless of whether the capturing group found its match at the same or a different recursion level as the backreference. Basically, normal backreferences in Ruby don’t pay any attention to recursion.

But while the normal capturing group storage in Ruby does not get any special treatment for recursion, Ruby actually stores a full stack of matches for each capturing groups at all recursion levels. This stack even includes recursion levels that the regex engine has already exited from.

Backreferences in Ruby can match the same text as was matched by a capturing group at any recursion level relative to the recursion level that the backreference is evaluated at. You can do this with the same syntax for named backreferences by adding a sign and a number after the name. In most situations you will use `+0` to specify that you want the backreference to reuse the text from the capturing group at the same recursion level. You can specify a positive number to reference the capturing group at a deeper level of recursion. This would be a recursion the regex engine has already exited from. You can specify a negative number to reference the capturing group a level that is less deep. This would be a recursion that is still in progress.

JGsoft V2 also supports backreferences that specify a recursion level using the same syntax as Ruby. To get the same behavior with JGsoft V2 as with Ruby, you have to use Ruby’s `\g` syntax for your subroutine calls.

Odd Length Palindromes in Ruby

In Ruby you can use `\b(?:'word'(?:'letter'[a-z])\g'word'\k'letter+0'|[a-z])\b` to match palindrome words such as `a`, `dad`, `radar`, `racecar`, and `redivider`. To keep this example simple, this regex only matches palindrome words that are an odd number of letters long.

Let’s see how this regex matches `radar`. The word boundary `\b` matches at the start of the string. The regex engine enters the capturing group “word”. `[a-z]` matches `r` which is then stored in the stack for the capturing group “letter” at recursion level zero. Now the regex engine enters the first recursion of the group “word”. `(?:'letter'[a-z])` matches and captures `a` at recursion level one. The regex enters the second recursion of the group “word”. `(?:'letter'[a-z])` captures `d` at recursion level two. During the next two recursions, the group captures `a` and `r` at levels three and four. The fifth recursion fails because there are no characters left in the string for `[a-z]` to match. The regex engine must backtrack.

The regex engine must now try the second alternative inside the group “word”. The second `[a-z]` in the regex matches the final `r` in the string. The engine now exits from a successful recursion, going one level back up to the third recursion.

After matching `\g'word'` the engine reaches `\k'letter+0'`. The backreference fails because the regex engine has already reached the end of the subject string. So it backtracks once more. The second alternative now matches the `a`. The regex engine exits from the third recursion.

The regex engine has again matched `\g'word'` and needs to attempt the backreference again. The backreference specifies `+0` or the present level of recursion, which is 2. At this level, the capturing group matched `d`. The backreference fails because the next character in the string is `r`. Backtracking again, the second alternative matches `d`.

Now, `\k'letter+0'` matches the second `a` in the string. That's because the regex engine has arrived back at the first recursion during which the capturing group matched the first `a`. The regex engine exits the first recursion.

The regex engine is now back outside all recursion. At this level, the capturing group stored `r`. The backreference can now match the final `r` in the string. Since the engine is not inside any recursion any more, it proceeds with the remainder of the regex after the group. `\b` matches at the end of the string. The end of the regex is reached and `radar` is returned as the overall match.

Backreferences to Other Recursion Levels

Backreferences to other recursion levels can be easily understood if we modify our palindrome example. `abcdefdcba` is also a palindrome matched by the previous regular expression. Consider the regular expression `\b(?:'word'(?:'letter'[a-z])\g'word'(?:\k'letter-1'|z)|[a-z])\b`. The backreference now wants a match the text one level less deep on the capturing group's stack. It is alternated with the letter `z` so that something can be matched when the backreference fails to match.

The new regex matches things like `abcdefdcbaz`. After a whole bunch of matching and backtracking, the second `[a-z]` matches `f`. The regex engine exits from a successful fifth recursion. The capturing group "letter" has stored the matches `a`, `b`, `c`, `d`, and `e` at recursion levels zero to four. Other matches by that group were backtracked and thus not retained.

Now the engine evaluates the backreference `\k'letter-1'`. The present level is 4 and the backreference specifies `-1`. Thus the engine attempts to match `d`, which succeeds. The engine exits from the fourth recursion.

The backreference continues to match `c`, `b`, and `a` until the regex engine has exited the first recursion. Now, outside all recursion, the regex engine again reaches `\k'letter-1'`. The present level is 0 and the backreference specifies `-1`. Since recursion level `-1` never happened, the backreference fails to match. This is not an error but simply a backreference to a non-participating capturing group. But the backreference has an alternative. `z` matches `z` and `\b` matches at the end of the string. `abcdefdcbaz` was matched successfully.

You can take this as far as you like. The regular expression `\b(?:'word'(?:'letter'[a-z])\g'word'(?:\k'letter-2'|z)|[a-z])\b` matches `abcdefcbazz`. `\b(?:'word'(?:'letter'[a-z])\g'word'(?:\k'letter-99'|z)|[a-z])\b` matches `abcdefzxxxxx`.

Going in the opposite direction, `\b(?:'word'(?:'letter'[a-z])\g'word'(?:\k'letter+1'|z)|[a-z])\b` matches `abcdefzedcb`. Again, after a whole bunch of matching and backtracking, the second `[a-z]` matches `f`, the regex engine is back at recursion level 4, and the group "letter" has `a`, `b`, `c`, `d`, and `e` at recursion levels zero to four on its stack.

Now the engine evaluates the backreference `\k'letter+1'`. The present level is 4 and the backreference specifies +1. The capturing group was backtracked at recursion level 5. This means we have a backreference to a non-participating group, which fails to match. The alternative `z` does match. The engine exits from the fourth recursion.

At recursion level 3, the backreference points to recursion level 4. Since the capturing group successfully matched at recursion level 4, it still has that match on its stack, even though the regex engine has already exited from that recursion. Thus `\k'letter+1'` matches `e`. Recursion level 3 is exited successfully.

The backreference continues to match `d` and `c` until the regex engine has exited the first recursion. Now, outside all recursion, the regex engine again reaches `\k'letter+1'`. The present level is 0 and the backreference specifies +1. The capturing group still retains all its previous successful recursion levels. So the backreference can still match the `b` that the group captured during the first recursion. Now `\b` matches at the end of the string. `abcdefzdc b` was matched successfully.

You can take this as far as you like in this direction too. The regular expression `\b(?:'word'(?'letter'[a-z])\g'word'(?:\k'letter+2'|z)|[a-z])\b` matches `abcdefzzedc`. `\b(?:'word'(?'letter'[a-z])\g'word'(?:\k'letter+99'|z)|[a-z])\b` matches `abcdefzzzzzzz`.

37. Recursion and Subroutine Calls May or May Not Be Atomic

Earlier topics in this tutorial explain regular expression recursion and regular expression subroutines. In this topic the word “recursion” refers to recursion of the whole regex, recursion of capturing groups, and subroutine calls to capturing groups.

Perl and Ruby backtrack into recursion if the remainder of the regex after the recursion fails. They try all permutations of the recursion as needed to allow the remainder of the regex to match. PCRE treats recursion as atomic. PCRE backtracks normally during the recursion, but once the recursion has matched, it does not try any further permutations of the recursion, even when the remainder of the regex fails to match. The result is that Perl and Ruby may find regex matches that PCRE cannot find, or that Perl and Ruby may find different regex matches.

Consider the regular expression `aa$|a(?:R)a|a` in Perl or the equivalent `aa$|a\g'0'a|a` in Ruby 2.0. PCRE supports either syntax. Let’s see how Perl, Ruby, and PCRE go through the matching process of this regex when `aaa` is the subject string.

The first alternative `aa$` fails because the anchor cannot be matched between the second and third `a` in the string. Attempting the second alternative at the start of the string, `a` matches `a`. Now the regex engine enters the first recursion.

Inside the recursion, the first alternative matches the second and third `a` in the string. The regex engine exits a successful recursion. But now, the `a` that follows `(?:R)` or `\g'0'` in the regex fails to match because the regex engine has already reached the end of the string. Thus the regex engine must backtrack. Here is where PCRE behaves differently than Perl or Ruby.

Perl and Ruby remember that inside the recursion the regex matched the second alternative and that there are three possible alternatives. Perl and Ruby backtrack *into* the recursion. The second alternative inside the recursion is backtracked, reducing the match so far to the first `a` in the string. Now the third alternative is attempted. `a` matches the second `a` in the string. The regex engine again exits successfully from the same recursion. This time, the `a` that follows `(?:R)` or `\g'0'` in the regex matches the third `a` in the string. `aaa` is found as the overall match.

PCRE, on the other hand, remembers nothing about the recursion other than that it matched `aa` at the end of the string. PCRE does backtrack *over* the recursion, reducing the match so far to the first `a` in the string. But this leaves the second alternative in the regex without any further permutations to try. Thus the `a` at the start of the second alternative is also backtracked, reducing the match so far to nothing. PCRE continues the match attempt at the start of the string with the third alternative and finds that `a` matches `a` at the start of the string. In PCRE, this is the overall match.

You can make recursion in Perl and Ruby atomic by adding an atomic group. `aa$|a(?:>R)a|a` in Perl and `aa$|a(?:>\g'0')a|a` in Ruby is the same as the original regexes in PCRE.

JGsoft V2 lets you choose whether recursion should be atomic or not. Atomic recursion gives better performance, but may exclude certain matches or find different matches as illustrated above. `aa$|a(?:P>0)a|a` is atomic in JGsoft V2. You can remember this because this syntax for recursion uses an angle bracket just like an atomic group. You can use a number or a name instead of zero for an atomic

subroutine call to a numbered or named capturing group. Any other syntax for recursion is not atomic in JGsoft V2.

Boost is of two minds. Recursion of the whole regex is atomic in Boost, like in PCRE. But Boost will backtrack into subroutine calls and into recursion of capturing groups, like Perl. So you can do non-atomic recursion in Boost by wrapping the whole regex into a capturing group and then calling that.

PCRE2 originally behaved like PCRE, treating all recursion and subroutine calls as atomic. PCRE2 10.30 changed this, trying to be more like Perl, but ending up like Boost. PCRE2 10.30 will backtrack into subroutine calls and recursion of capturing groups like Perl does. But PCRE2 is still not able to backtrack into recursion of the whole regex. In the examples below, “PCRE” means the original PCRE only. For PCRE2 10.22 and prior, follow the PCRE example. For PCRE2 10.30 and later, follow the Perl example.

Palindromes of Any Length in Perl and Ruby

The topic about recursion and capturing groups explains a regular expression to match palindromes that are an odd number of characters long. The solution seems trivial. `\b(?:'word'(? 'letter'[a-z]))(?:&word)\k'letter'|[a-z]?)\b` does the trick in Perl. The quantifier `?` makes the `[a-z]` that matches the letter in the middle of the palindrome optional. In Ruby we can use `\b(?:'word'(? 'letter'[a-z])\g'word'\k'letter+0'|[a-z]?)\b` which adds the same quantifier to the solution that specifies the recursion level for the backreference. In PCRE, the Perl solution still matches odd-length palindromes, but not even-length palindromes.

Let’s see how these regexes match or fail to match `deed`. PCRE starts off the same as Perl and Ruby, just as in the original regex. The group “letter” matches `d`. During three consecutive recursions, the group captures `e`, `e`, and `d`. The fourth recursion fails, because there are no characters left to match. Back in the third recursion, the first alternative is backtracked and the second alternative matches `d` at the end of the string. The engine exits the third recursion with a successful match. Back in the second recursion, the backreference fails because there are no characters left in the string.

Here the behavior diverges. Perl and Ruby backtrack *into* the third recursion and backtrack the quantifier `?` that makes the second alternative optional. In the third recursion, the second alternative gives up the `d` that it matched at the end of the string. The engine exits the third recursion again, this time with a successful zero-length match. Back in the second recursion, the backreference still fails because the group stored `e` for the second recursion but the next character in the string is `d`. Thus the first alternative is backtracked and the second alternative matches the second `e` in the string. The second recursion is exited with success.

In the first recursion, the backreference again fails. The group stored `e` for the first recursion but the next character in the string is `d`. Again, Perl and Ruby backtrack into the second recursion to try the permutation where the second alternative finds a zero-length match. Back in the first recursion again, the backreference now matches the second `e` in the string. The engine leaves the first recursion with success. Back in the overall match attempt, the backreference matches the final `d` in the string. The word boundary succeeds and an overall match is found.

PCRE, however, does not backtrack into the third recursion. It does backtrack *over* the third recursion when it backtracks the first alternative in the second recursion. Now, the second alternative in the second recursion matches the second `e` in the string. The second recursion is exited with success.

In the first recursion, the backreference again fails. The group stored **e** for the first recursion but the next character in the string is **d**. Again, PCRE does not backtrack into the second recursion, but immediately fails the first alternative in the first recursion. The second alternative in the first recursion now matches the first **e** in the string. PCRE exits the first recursion with success. Back in the overall match attempt, the backreference fails, because the group captured **d** prior to the recursion, and the next character is the second **e** in the string. Backtracking again, the second alternative in the overall regex match now matches the first **d** in the string. Then the word boundary fails. PCRE did not find any matches.

Palindromes of Any Length in PCRE

To match palindromes of any length in PCRE, we need a regex that matches words of an even number of characters and of an odd number of characters separately. Free-spacing mode makes this regex easier to read:

```
\b(?:'word'
  (? 'oddword' (? 'oddletter' [a-z]))(?P>oddword) \k'oddletter' [a-z])
| (? 'evenword' (? 'evenletter' [a-z]))(?P>evenword)?\k'evenletter'
)\b
```

Basically, this is two copies of the original regex combined with alternation. The first alternative has the groups “word” and “letter” renamed to “oddword” and “oddletter”. The second alternative has the groups “word” and “letter” renamed to “evenword” and “evenletter”. The call `(?P>evenword)` is now made optional with a question mark instead of the alternative `[a-z]`. A new group “word” combines the two groups “oddword” and “evenword” so that the word boundaries still apply to the whole regex.

The first alternative “oddword” in this regex matches a palindrome of odd length like **radar** in exactly the same way as the regex discussed in the topic about recursion and capturing groups does. The second alternative in the new regex is never attempted.

When the string is a palindrome of even length like **deed**, the new regex first tries all permutations of the first alternative. The second alternative “evenword” is attempted only after the first alternative fails to find a match.

The second alternative starts off in the same way as the original regex. The group “evenletter” matches **d**. During three consecutive recursions, the group captures **e**, **e**, and **d**. The fourth recursion fails, because there are no characters left to match. Back in the third recursion, the regex engine notes that recursive call `(?P>evenword)?` is optional. It proceeds to the backreference `\k'evenletter'`. The backreference fails because there are no characters left in the string. Since the recursion has no further alternatives to try, it is backtracked. The group “evenletter” must give up its most recent match and PCRE exits from the failed third recursion.

In the second recursion, the backreference fails because the capturing group matched **e** during that recursion but the next character in the string is **d**. The group gives up another match and PCRE exits from the failed second recursion.

Back in the first recursion, the backreference succeeds. The group matched the first **e** in the string during that recursion and the backreference matches the second. PCRE exits from the successful first recursion.

Back in the overall match attempt, the backreference succeeds again. The group matched the **d** at the start of the string during the overall match attempt, and the backreference matches the final **d**. Exiting the groups “evenword” and “word”, the word boundary matches at the end of the string. **deed** is the overall match.

38. POSIX Bracket Expressions

POSIX bracket expressions are a special kind of character classes. POSIX bracket expressions match one character out of a set of characters, just like regular character classes. They use the same syntax with square brackets. A hyphen creates a range, and a caret at the start negates the bracket expression.

One key syntactic difference is that the backslash is NOT a metacharacter in a POSIX bracket expression. So in POSIX, the regular expression `[\\d]` matches a `\` or a `d`. To match a `]`, put it as the first character after the opening `[` or the negating `^`. To match a `-`, put it right before the closing `]`. To match a `^`, put it before the final literal `-` or the closing `]`. Put together, `[]\\d^-]` matches `]`, `\`, `d`, `^` or `-`.

The main purpose of bracket expressions is that they adapt to the user's or application's locale. A locale is a collection of rules and settings that describe language and cultural conventions, like sort order, date format, etc. The POSIX standard defines these locales.

Generally, only POSIX-compliant regular expression engines have proper and full support for POSIX bracket expressions. Some non-POSIX regex engines support POSIX character classes, but usually don't support collating sequences and character equivalents. Regular expression engines that support Unicode use Unicode properties and scripts to provide functionality similar to POSIX bracket expressions. In Unicode regex engines, shorthand character classes like `\w` normally match all relevant Unicode characters, alleviating the need to use locales.

Character Classes

Don't confuse the POSIX term "character class" with what is normally called a regular expression character class. `[x-z0-9]` is an example of what this tutorial calls a "character class" and what POSIX calls a "bracket expression". `[:digit:]` is a POSIX character class, used inside a bracket expression like `[x-z[:digit:]]`. The POSIX character class names must be written all lowercase.

When used on ASCII strings, these two regular expressions find exactly the same matches: a single character that is either `x`, `y`, `z`, or a digit. When used on strings with non-ASCII characters, the `[:digit:]` class may include digits in other scripts, depending on the locale.

The POSIX standard defines 12 character classes. The table below lists all 12, plus the `[:ascii:]` and `[:word:]` classes that some regex flavors also support. The table also shows equivalent character classes that you can use in ASCII and Unicode regular expressions if the POSIX classes are unavailable. The ASCII equivalents correspond exactly what is defined in the POSIX standard. The Unicode equivalents correspond to what most Unicode regex engines match. The POSIX standard does not define a Unicode locale. Some classes also have Perl-style shorthand equivalents.

Java does not support POSIX bracket expressions, but does support POSIX character classes using the `\p` operator. Though the `\p` syntax is borrowed from the syntax for Unicode properties, the POSIX classes in Java only match ASCII characters as indicated below. The class names are case sensitive. Unlike the POSIX syntax which can only be used inside a bracket expression, Java's `\p` can be used inside and outside bracket expressions.

In Java 8 and prior, it does not matter whether you use the `Is` prefix with the `\p` syntax or not. So in Java 8, `\p{Alnum}` and `\p{IsAlnum}` are identical. In Java 9 and later there is a difference. Without the `Is` prefix,

the behavior is exactly the same as in previous versions of Java. The syntax with the `Is` prefix now matches Unicode characters too. For `\p{IsPunct}` this also means that it no longer matches the ASCII characters that are in the Symbol Unicode category.

The JGsoft flavor supports both the POSIX and Java syntax. Originally it matched Unicode characters using either syntax. As of JGsoft V2, it matches only ASCII characters when using the POSIX syntax, and Unicode characters when using the Java syntax.

POSIX	Description	ASCII	Unicode	Shorthand Java
<code>[:alnum:]</code>	Alphanumeric characters	<code>[a-zA-Z0-9]</code>	<code>[\p{L}\p{Nl}\p{Nd}]</code>	<code>\p{Alnum}</code>
<code>[:alpha:]</code>	Alphabetic characters	<code>[a-zA-Z]</code>	<code>\p{L}\p{Nl}</code>	<code>\p{Alpha}</code>
<code>[:ascii:]</code>	ASCII characters	<code>[\x00-\x7F]</code>	<code>\p{InBasicLatin}</code>	<code>\p{ASCII}</code>
<code>[:blank:]</code>	Space and tab	<code>[\t]</code>	<code>[\p{Zs}\t]</code>	<code>\h</code> <code>\p{Blank}</code>
<code>[:cntrl:]</code>	Control characters	<code>[\x00-\x1F\x7F]</code>	<code>\p{Cc}</code>	<code>\p{Cntrl}</code>
<code>[:digit:]</code>	Digits	<code>[0-9]</code>	<code>\p{Nd}</code>	<code>\d</code> <code>\p{Digit}</code>
<code>[:graph:]</code>	Visible characters (anything except spaces and control characters)	<code>[\x21-\x7E]</code>	<code>[\p{Z}\p{C}]</code>	<code>\p{Graph}</code>
<code>[:lower:]</code>	Lowercase letters	<code>[a-z]</code>	<code>\p{Ll}</code>	<code>\l</code> <code>\p{Lower}</code>
<code>[:print:]</code>	Visible characters and spaces (anything except control characters)	<code>[\x20-\x7E]</code>	<code>\p{C}</code>	<code>\p{Print}</code>
<code>[:punct:]</code>	Punctuation symbols. (and symbols).	<code>[!\"#\$%&'()*+,-./:;<=>?@[\^_`{ }~]</code>	<code>\p{P}</code>	<code>\p{Punct}</code>
<code>[:space:]</code>	All whitespace characters, including line breaks	<code>[\t\r\n\v\f]</code>	<code>[\p{Z}\t\r\n\v\f]</code>	<code>\s</code> <code>\p{Space}</code>
<code>[:upper:]</code>	Uppercase letters	<code>[A-Z]</code>	<code>\p{Lu}</code>	<code>\u</code> <code>\p{Upper}</code>
<code>[:word:]</code>	Word characters (letters, numbers and underscores)	<code>[A-Za-z0-9_]</code>	<code>[\p{L}\p{Nl}\p{Nd}\p{Pc}]</code>	<code>\w</code> <code>\p{IsWord}</code>
<code>[:xdigit:]</code>	Hexadecimal digits	<code>[A-Fa-f0-9]</code>	<code>[A-Fa-f0-9]</code>	<code>\p{XDigit}</code>
POSIX	Description	ASCII	Unicode	Shorthand Java

Collating Sequences

A POSIX locale can have collating sequences to describe how certain characters or groups of characters should be ordered. In Czech, for example, `ch` as in `chemie` (“chemistry” in Czech) is a digraph. This means it should be treated as if it were one character. It is ordered between `h` and `i` in the Czech alphabet. You can use the collating sequence element `[.ch.]` inside a bracket expression to match `ch` when the Czech locale (cs-CZ) is active. The regex `[[.ch.]]emie` matches `chemie`. Notice the double square brackets. One pair for the bracket expression, and one pair for the collating sequence.

Other than POSIX-compliant engines part of a POSIX-compliant system, none of the regex flavors discussed in this tutorial support collating sequences.

Note that a fully POSIX-compliant regex engine treats `ch` as a single character when the locale is set to Czech. This means that `[^x]emie` also matches `chemie`. `[^x]` matches a single character that is not an `x`, which includes `ch` in the Czech POSIX locale.

In any other regular expression engine, or in a POSIX engine using a locale that does not treat `ch` as a digraph, `[^x]emie` matches the misspelled word `cemie` but not `chemie`, as `[^x]` cannot match the two characters `ch`.

Finally, note that not all regex engines claiming to implement POSIX regular expressions actually have full support for collating sequences. Sometimes, these engines use the regular expression syntax defined by POSIX, but don't have full locale support. You may want to try the above matches to see if the engine you're using does. Tcl's `regexp` command, for example, supports the syntax for collating sequences. But Tcl only supports the Unicode locale, which does not define any collating sequences. The result is that in Tcl, a collating sequence specifying a single character matches just that character. All other collating sequences result in an error.

Character Equivalents

A POSIX locale can define character equivalents that indicate that certain characters should be considered as identical for sorting. In French, for example, accents are ignored when ordering words. `élève` comes before `être` which comes before `événement`. `é` and `ê` are all the same as `e`, but `l` comes before `t` which comes before `v`. With the locale set to French, a POSIX-compliant regular expression engine matches `è`, `é`, `ê` and `ë` when you use the collating sequence `[=e=]` in the bracket expression `[[=e=]]`.

If a character does not have any equivalents, the character equivalence token simply reverts to the character itself. `[[=x=] [=z=]]`, for example, is the same as `[xz]` in the French locale.

Like collating sequences, POSIX character equivalents are not available in any regex engine discussed in this tutorial, other than those following the POSIX standard. And those that do may not have the necessary POSIX locale support. Here too Tcl's `regexp` command supports the syntax for character equivalents. But the Unicode locale, the only one Tcl supports, does not define any character equivalents. This effectively means that `[[=e=]]` and `[e]` are exactly the same in Tcl, and only match `e`, for any character you may try instead of "e".

39. Zero-Length Regex Matches

We saw that anchors, word boundaries, and lookaround match at a position, rather than matching a character. This means that when a regex only consists of one or more anchors, word boundaries, or lookarounds, it can result in a zero-length match. Depending on the situation, this can be very useful or undesirable.

In email, for example, it is common to prepend a “greater than” symbol and a space to each line of the quoted message. 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 removes the regex match from the string, and inserts 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. The replacement string is inserted there, just like we want it.

Using `^\d*$` to test if the user entered a number would give undesirable results. It causes the script to accept an empty string as a valid input. Let’s see why.

There is only one “character” position in an empty string: the void after the string. The first token in the regex is `^`. It matches the position before the void after the string, because it is preceded by the void before the string. The next token is `\d*`. One of the star’s effects is that it makes the `\d`, in this case, optional. The engine tries to match `\d` with the void after the string. That fails. But the star turns the failure of the `\d` into a zero-length success. The engine proceeds with the next regex token, without advancing the position in the string. So the engine arrives at `$`, and the void after the string. These match. At this point, the entire regex has matched the empty string, and the engine reports success.

The solution is to use the regex `^\d+$` with the proper quantifier to require at least one digit to be entered. If you always make sure that your regexes cannot find zero-length matches, other than special cases such as matching the start or end of each line, then you can save yourself the headache you’ll get from reading the remainder of this topic.

Skipping Zero-Length Matches

Not all flavors support zero-length matches. The `TRegEx` class in Delphi XE5 and prior always skips zero-length matches. The `TPerlRegEx` class does too by default in XE5 and prior, but allows you to change this via the `State` property. In Delphi XE6 and later, `TRegEx` never skips zero-length matches while `TPerlRegEx` does not skip them by default but still allows you to skip them via the `State` property. PCRE finds zero-length matches by default, but can skip them if you set `PCRE_NOTEMPTY`.

Advancing After a Zero-Length Regex Match

If a regex can find zero-length matches at any position in the string, then it will. The regex `\d*` matches zero or more digits. If the subject string does not contain any digits, then this regex finds a zero-length match at every position in the string. It finds 4 matches in the string `abc`, one before each of the three letters, and one at the end of the string.

Things get tricky when a regex can find zero-length matches at any position as well as certain non-zero-length matches. Say we have the regex `\d*|x`, the subject string `x1`, and a regex engine allows zero-length matches.

Which and how many matches do we get when iterating over all matches? The answer depends on how the regex engine advances after zero-length matches. The answer is tricky either way.

The first match attempt begins at the start of the string. `\d` fails to match `x`. But the `*` makes `\d` optional. The first alternative finds a zero-length match at the start of the string. Until here, all regex engines that allow zero-length matches do the same.

Now the regex engine is in a tricky situation. We're asking it to go through the entire string to find all non-overlapping regex matches. The first match ended at the start of the string, where the first match attempt began. The regex engine needs a way to avoid getting stuck in an infinite loop that forever finds the same zero-length match at the start of the string.

The simplest solution, which is used by most regex engines, is to start the next match attempt one character after the end of the previous match, if the previous match was zero-length. In this case, the second match attempt begins at the position between the `x` and the `1` in the string. `\d` matches `1`. The end of the string is reached. The quantifier `*` is satisfied with a single repetition. `1` is returned as the overall match.

The other solution, which is used by Perl, is to always start the next match attempt at the end of the previous match, regardless of whether it was zero-length or not. If it was zero-length, the engine makes note of that, as it must not allow a zero-length match at the same position. Thus Perl begins the second match attempt also at the start of the string. The first alternative again finds a zero-length match. But this is not a valid match, so the engine backtracks through the regular expression. `\d*` is forced to give up its zero-length match. Now the second alternative in the regex is attempted. `x` matches `x` and the second match is found. The third match attempt begins at the position after the `x` in the string. The first alternative matches `1` and the third match is found.

But the regex engine isn't done yet. After `x` is matched, it makes one more match attempt starting at the end of the string. Here too `\d*` finds a zero-length match. So depending on how the engine advances after zero-length matches, it finds either three or four matches.

One exception is the JGsoft engine. The JGsoft engine advances one character after a zero-length match, like most engines do. But it has an extra rule to skip zero-length matches at the position where the previous match ended, so you can never have a zero-length match immediately adjacent to a non-zero-length match. In our example the JGsoft engine only finds two matches: the zero-length match at the start of the string, and `1`.

Python 3.6 and prior advance after zero-length matches. The `gsub()` function to search-and-replace skips zero-length matches at the position where the previous non-zero-length match ended, but the `finder()` function returns those matches. So a search-and-replace in Python gives the same results as the Just Great Software applications, but listing all matches adds the zero-length match at the end of the string.

Python 3.7 changed all this. It handles zero-length matches like Perl. `gsub()` does now replace zero-length matches that are adjacent to another match. This means regular expressions that can find zero-length matches are not compatible between Python 3.7 and prior versions of Python.

PCRE 8.00 and later and PCRE2 handle zero-length matches like Perl by backtracking. They no longer advance one character after a zero-length match like PCRE 7.9 used to do.

The `regxp` functions in R and PHP are based on PCRE, so they avoid getting stuck on a zero-length match by backtracking like PCRE does. But the `gsub()` function to search-and-replace in R also skips zero-length matches at the position where the previous non-zero-length match ended, like `gsub()` in Python 3.6 and

prior does. The other regexp functions in R and all the functions in PHP do allow zero-length matches immediately adjacent to non-zero-length matches, just like PCRE itself.

Caution for Programmers

A regular expression such as `$` all by itself can find a zero-length match at the end of the string. If you would query the engine for the character position, it would return the length of the string if string indexes are zero-based, or the length+1 if string indexes 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 `^$` in multi-line mode if the last character in the string is a newline.

40. 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. This is the case with Ruby and the Just Great Software applications. In EditPad Pro `\G` matches at the position of the text cursor. 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.

The distinction between the end of the previous match and the start of the match attempt is also important if your regular expression can find zero-length matches. Most regex engines advance through the string after a zero-length match. In that case, the start of the match attempt is one character further in the string than the end of the previous match attempt. .NET, Java, and Boost advance this way and also match `\G` at the end of the previous match attempt. Thus `\G` fails to match when .NET, Java, and Boost have advanced after a zero-length match.

`\G` Magic with Perl

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

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

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

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

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

\G in Other Programming Languages

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

Start of Match Attempt

Normally, `\A` is a start-of-string anchor. But in Tcl, the anchor `\A` matches at the start of the match attempt rather than at the start of the string. With the GNU flavors, `\<` (backslash backtick) does the same. This makes no difference if you're only making one call to `regexp` in Tcl or `regexec()` in the GNU library. It can make a difference if you make a second call to find another match in the remainder of the string after the first match. `\A` or `\<` then matches at the end of the first match, instead of failing to match as start-of-string anchors normally do. Strangely enough, the caret does not have this issue in either Tcl or GNU's library.

41. Replacement Strings Tutorial

A replacement string, also known as the replacement text, is the text that each regular expression match is replaced with during a search-and-replace. In most applications, the replacement text supports special syntax that allows you to reuse the text matched by the regular expression or parts thereof in the replacement. This tutorial explains this syntax. While replacement strings are fairly simple compared with regular expressions, there is still great variety between the syntax used by various applications and their actual behavior.

In this book, replacement strings are shown as `replace` like you would enter them in the Replace box of an application. Literal text in the replacement is highlighted in yellow. As `$&\$` shows, special tokens are highlighted in blue and escaped characters in gray.

42. Special Characters

The most basic replacement string consists only of literal characters. The replacement `replacement` simply replaces each regex match with the text `replacement`.

Because we want to be able to do more than simply replace each regex match with the exact same text, we need to reserve certain characters for special use. In most replacement text flavors, two characters tend to have special meanings: the backslash `\` and the dollar sign `$`. Whether and how to escape them depends on the application you're using. In some applications, you always need to escape them when you want to use them as literal characters. In other applications, you only need to escape them if they would form a replacement text token with the character that follows.

In the JGsoft flavor and Delphi, you can use a backslash to escape the backslash and the dollar, and you can use a dollar to escape the dollar. `\\` replaces with a literal backslash, while `\$` and `$$` replace with a literal dollar sign. You only need to escape them to suppress their special meaning in combination with other characters. In `\!` and `$!`, the backslash and dollar are literal characters because they don't have a special meaning in combination with the exclamation point. You can't and needn't escape the exclamation point or any other character except the backslash and dollar, because they have no special meaning in JGsoft and Delphi replacement strings.

In .NET, JavaScript, VBScript, XRegExp, PCRE2, and `std::regex` you can escape the dollar sign with another dollar sign. `$$` replaces with a literal dollar sign. XRegExp and PCRE2 require you to escape all literal dollar signs. They treat unescaped dollar signs that don't form valid replacement text tokens as errors. In .NET, JavaScript (without XRegExp), and VBScript you only need to escape the dollar sign to suppress its special meaning in combination with other characters. In `$\` and `$!`, the dollar is a literal character because it doesn't have a special meaning in combination with the backslash or exclamation point. You can't and needn't escape the backslash, exclamation point, or any other character except dollar, because they have no special meaning in .NET, JavaScript, VBScript, and PCRE2 replacement strings.

In Java, an unescaped dollar sign that doesn't form a token is an error. You must escape the dollar sign with a backslash or another dollar sign to use it as a literal character. `$!` is an error because the dollar sign is not escaped and has no special meaning in combination with the exclamation point. A backslash always escapes the character that follows. `\!` replaces with a literal exclamation point, and `\\` replaces with a single backslash. A single backslash at the end of the replacement text is an error.

In Python and Ruby, the dollar sign has no special meaning. You can use a backslash to escape the backslash. You only need to escape the backslash to suppress its special meaning in combination with other characters. In `\!`, the backslash is a literal character because it doesn't have a special meaning in combination with the exclamation point. You can't and needn't escape the exclamation point or any other character except the backslash, because they have no special meaning in Python and Ruby replacement strings. An unescaped backslash at the end of the replacement text, however, is an error in Python but a literal backslash in Ruby.

In PHP's `preg_replace`, you can use a backslash to escape the backslash and the dollar. `\\` replaces with a literal backslash, while `\$` replaces with a literal dollar sign. You only need to escape them to suppress their special meaning in combination with other characters. In `\!`, the backslash is a literal character because it doesn't have a special meaning in combination with the exclamation point. You can't and needn't escape the exclamation point or any other character except the backslash and dollar, because they have no special meaning in PHP replacement strings.

In Boost, a backslash always escapes the character that follows. `\!` replaces with a literal exclamation point, and `\\` replaces with a single backslash. A single backslash at the end of the replacement text is ignored. An unescaped dollar sign is a literal dollar sign if it doesn't form a replacement string token. You can escape dollar signs with a backslash or with another dollar sign. So `$$`, `\\$`, and `\\$` all replace with a single dollar sign.

In R, the dollar sign has no special meaning. A backslash always escapes the character that follows. `\!` replaces with a literal exclamation point, and `\\` replaces with a single backslash. A single backslash at the end of the replacement text is ignored.

In Tcl, the ampersand `&` has a special meaning, and must be escaped with a backslash if you want a literal ampersand in your replacement text. You can use a backslash to escape the backslash. You only need to escape the backslash to suppress its special meaning in combination with other characters. In `\!`, the backslash is a literal character because it doesn't have a special meaning in combination with the exclamation point. You can't and needn't escape the exclamation point or any other character except the backslash and ampersand, because they have no special meaning in Tcl replacement strings. An unescaped backslash at the end of the replacement text is a literal backslash.

In XPath, an unescaped backslash is an error. An unescaped dollar sign that doesn't form a token is also an error. You must escape backslashes and dollars with a backslash to use them as literal characters. The backslash has no special meaning other than to escape another backslash or a dollar sign.

Perl is a special case. Perl doesn't really have a replacement text syntax. So it doesn't have escape rules for replacement texts either. In Perl source code, replacement strings are simply double-quoted strings. What looks like backreferences in replacement text are really interpolated variables. You could interpolate them in any other double-quoted string after a regex match, even when not doing a search-and-replace.

Special Characters and Programming Languages

The rules in the previous section explain how the search-and-replace functions in these programming languages parse the replacement text. If your application receives the replacement text from user input, then the user of your application would have to follow these escape rules, and only these rules. 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-and-replace function of a text editor like EditPad Pro, you should not escape or repeat the quote characters like you do in a programming language.

If you specify the replacement text as a string constant in your source code, then you have to keep in mind which characters are given special treatment inside string constants by your programming language. That is because those characters are processed by the compiler, before the replacement text function sees the string. So Java, for example, to replace all regex matches with a single dollar sign, you need to use the replacement text `\$`, which you need to enter in your source code as `\"\\$"`. The Java compiler turns the escaped backslash in the source code into a single backslash in the string that is passed on to the `replaceAll()` function. That function then sees the single backslash and the dollar sign as an escaped dollar sign.

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

43. Non-Printable Characters

Most applications and programming languages do not support any special syntax in the replacement text to make it easier to enter non-printable characters. If you are the end user of an application, that means you'll have to use an application such as the Windows Character Map to help you enter characters that you cannot type on your keyboard. If you are programming, you can specify the replacement text as a string constant in your source code. Then you can use the syntax for string constants in your programming language to specify non-printable characters.

The Just Great Software applications are an exception. They allow you to use special escape sequences to enter a few common control characters. Use `\t` to replace with a tab character (ASCII 0x09), `\r` for carriage return (0x0D), and `\n` for line feed (0x0A). Remember that Windows text files use `\r\n` to terminate lines, while UNIX text files use `\n`.

Python also supports the above escape sequences in replacement text, in addition to supporting them in string constants. Python and Boost also support these more exotic non-printables: `\a` (bell, 0x07), `\f` (form feed, 0x0C) and `\v` (vertical tab, 0x0B).

The Just Great Software applications and Boost also support hexadecimal escapes. You can use `\x{FFFF}` to insert a Unicode character. The euro currency sign occupies Unicode code point U+20AC. If you cannot type it on your keyboard, you can insert it into the replacement text with `\x{20AC}`. The JGsoft apps also support `\u20AC`. For the 127 ASCII characters, you can use `\x00` through `\x7F`. If you are working with files using 8-bit code pages in EditPad or PowerGREP, or are using Boost with 8-bit character strings, you can also use `\x80` through `\xFF` to insert characters from those 8-bit code pages.

Python does not support hexadecimal escapes in the replacement text syntax, even though it supports `\xFF` and `\uFFFF` in string constants.

Regex Syntax versus String Syntax

Many programming languages support escapes for non-printable characters in their syntax for literal strings in source code. Then such escapes are translated by the compiler into their actual characters before the string is passed to the search-and-replace function. If the search-and-replace function does not support the same escapes, this can cause an apparent difference in behavior when a regex is specified as a literal string in source code compared with a regex that is read from a file or received from user input. For example, JavaScript's `string.replace()` function does not support any of these escapes. But the JavaScript language does support escapes like `\n`, `\x0A`, and `\u000A` in string literals. So when developing an application in JavaScript, `\n` is only interpreted as a newline when you add the replacement text as a string literal to your source code. Then the JavaScript interpreter then translates `\n` and the `string.replace()` function sees an actual newline character. If your code reads the same replacement text from a file, then `string.replace()` function sees `\n`, which it treats as a literal backslash and a literal `n`.

44. Matched Text

Reinserting the entire regex match into the replacement text allows a search-and-replace to insert text before and after regular expression matches without really replacing anything. It also allows you to replace the regex match with something that contains the regex match. For example, you could replace all matches of the regex `https?://\S+` with `$&` to turn all URLs in a file into HTML anchors that link to those URLs.

`$&` is substituted with the whole regex match in the replacement text in the JGsoft applications, Delphi, .NET, JavaScript, and VBScript. It is also the variable that holds the whole regex match in Perl. `\&` works in the JGsoft applications, Delphi, and Ruby. In Tcl, `&` all by itself represents the whole regex match, while `$&` is a literal dollar sign followed by the whole regex match, and `\&` is a literal ampersand.

In Boost and `std::regex` your choice of replacement format changes the meaning of `&` and `$&`. When using the `sed` replacement format, `&` represents the whole regex match and `$&` is a literal dollar sign followed by the whole regex match. When using the default (ECMAScript) or “all” replacement format, `&` is a literal ampersand and `$&` represents the whole regex match.

The overall regex match is usually treated as an implied capturing group number zero. In many applications you can use the syntax for backreferences in the replacement text to reference group number zero and thus insert the whole regex match in the replacement text. You can do this with `$0` in the JGsoft applications, Delphi, .NET, Java, XRegExp, PCRE2, PHP, and XPath. `\0` works with the JGsoft applications, Delphi, Ruby, PHP, and Tcl.

Python does not support any of the above. In Python you can reinsert the whole regex match by using the syntax for named backreferences with group number zero: `\g<0>`. This also works in the JGsoft applications. Delphi does not support this, even though it supports named backreferences using this syntax.

45. Numbered and Named Backreferences

If your regular expression has named or numbered capturing groups, then you can reinsert the text matched by any of those capturing groups in the replacement text. Your replacement text can reference as many groups as you like, and can even reference the same group more than once. This makes it possible to rearrange the text matched by a regular expression in many different ways. As a simple example, the regex `*(\w+)*` matches a single word between asterisks, storing the word in the first (and only) capturing group. The replacement text `\1` replaces each regex match with the text stored by the capturing group between bold tags. Effectively, this search-and-replace replaces the asterisks with bold tags, leaving the word between the asterisks in place. This technique using backreferences is important to understand. Replacing `*word*` as a whole with `word` is far easier and far more efficient than trying to come up with a way to correctly replace the asterisks separately.

The `\1` syntax for backreferences in the replacement text is borrowed from the syntax for backreferences in the regular expression. `\1` through `\9` are supported by the JGsoft applications, Delphi, Perl (though deprecated), Python, Ruby, PHP, R, Boost, and Tcl. Double-digit backreferences `\10` through `\99` are supported by the JGsoft applications, Delphi, Python, and Boost. If there are not enough capturing groups in the regex for the double-digit backreference to be valid, then all these flavors treat `\10` through `\99` as a single-digit backreference followed by a literal digit. The flavors that support single-digit backreferences but not double-digit backreferences also do this.

`$1` through `$99` for single-digit and double-digit backreferences are supported by the JGsoft applications, Delphi, .NET, Java, JavaScript, VBScript, PCRE2, PHP, Boost, `std::regex`, and XPath. These are also the variables that hold text matched by capturing groups in Perl. If there are not enough capturing groups in the regex for a double-digit backreference to be valid, then `$10` through `$99` are treated as a single-digit backreference followed by a literal digit by all these flavors except .NET, Perl, PCRE2, and `std::regex`.

Putting curly braces around the digit `${1}` isolates the digit from any literal digits that follow. This works in the JGsoft applications, Delphi, .NET, Perl, PCRE2, PHP, Boost, and XRegExp.

If your regular expression has named capturing groups, then you should use named backreferences to them in the replacement text. The regex `(?'name'group)` has one group called “name”. You can reference this group with `${name}` in the JGsoft applications, Delphi, .NET, PCRE2, Java 7, and XRegExp. PCRE2 also supports `$name` without the curly braces. In Perl 5.10 and later you can interpolate the variable `${+name}`. Boost too uses `${+name}` in replacement strings. `${name}` does not work in any version of Perl. `$name` is unique to PCRE2.

In Python, if you have the regex `(?P<name>group)` then you can use its match in the replacement text with `\g<name>`. This syntax also works in the JGsoft applications and Delphi. Python and the JGsoft applications, but not Delphi, also support numbered backreferences using this syntax. In Python this is the only way to have a numbered backreference immediately followed by a literal digit.

PHP and R support named capturing groups and named backreferences in regular expressions. But they do not support named backreferences in replacement texts. You’ll have to use numbered backreferences in the replacement text to reinsert text matched by named groups. To determine the numbers, count the opening parentheses of all capturing groups (named and unnamed) in the regex from left to right.

Backreferences to Non-Existent Capturing Groups

An invalid backreference is a reference to a number greater than the number of capturing groups in the regex or a reference to a name that does not exist in the regex. Such a backreference can be treated in three different ways. Delphi, Perl, Ruby, PHP, R, Boost, std::regex, XPath, and Tcl substitute the empty string for invalid backreferences. Java, XRegExp, PCRE2, and Python treat them as a syntax error. JavaScript (without XRegExp) and .NET treat them as literal text.

The original JGsoft flavor replaced invalid backreferences with the empty string. But JGsoft V2 treats them as a syntax error. Applications using the V2 flavor all apply syntax coloring to replacement strings, highlighting invalid backreferences in red.

Backreferences to Non-Participating Capturing Groups

A non-participating capturing group is a group that did not participate in the match attempt at all. This is different from a group that matched an empty string. The group in `a(b?)c` always participates in the match. Its contents are optional but the group itself is not optional. The group in `a(b)?c` is optional. It participates when the regex matches `abc`, but not when the regex matches `ac`.

In most applications, there is no difference between a backreference in the replacement string to a group that matched the empty string or a group that did not participate. Both are replaced with an empty string. Two exceptions are Python and PCRE2. They do allow backreferences in the replacement string to optional capturing groups. But the search-and-replace will return an error code in PCRE2 if the capturing group happens not to participate in one of the regex matches. The same situation raises an exception in Python 3.4 and prior. Python 3.5 no longer raises the exception.

Backreference to The Highest-Numbered Group

In the JGsoft applications and Delphi, `$+` inserts the text matched by the highest-numbered group that actually participated in the match. In Perl 5.18, the variable `$+` holds the same text. When `(a)(b)|(c)(d)` matches `ab`, `$+` is substituted with `b`. When the same regex matches `cd`, `$+` inserts `d`. `\+` does the same in the JGsoft applications, Delphi, and Ruby.

In .NET, VBScript, and Boost `$+` inserts the text matched by the highest-numbered group, regardless of whether it participated in the match or not. If it didn't, nothing is inserted. In Perl 5.16 and prior, the variable `$+` holds the same text. When `(a)(b)|(c)(d)` matches `ab`, `$+` is substituted with the empty string. When the same regex matches `cd`, `$+` inserts `d`.

Boost 1.42 added additional syntax of its own invention for either meaning of highest-numbered group. `$$N`, `$_LAST_SUBMATCH_RESULT`, and `$$LAST_SUBMATCH_RESULT` all insert the text matched by the highest-numbered group that actually participated in the match. `$_LAST_PAREN_MATCH` and `$$LAST_PAREN_MATCH` both insert the text matched by the highest-numbered group regardless of whether participated in the match.

46. Match Context

Some applications support special tokens in replacement strings that allow you to insert the subject string or the part of the subject string before or after the regex match. This can be useful when the replacement text syntax is used to collect search matches and their context instead of making replacements in the subject string.

In the replacement text, `$<` (dollar backtick) is substituted with the part of the subject string to the left of the regex match in the JGsoft applications, Delphi, .NET, JavaScript, VBScript, Boost, and `std::regex`. It is also the variable that holds the part of the subject string to the left of the regex match in Perl. `\<` (backslash backtick) works in the JGsoft applications, Delphi, and Ruby.

In the same applications, you can use `$'` (dollar quote) or `\'` (backslash quote) to insert the part of the subject string to the right of the regex match.

In the replacement text, `$_` is substituted with the entire subject string in the JGsoft applications, Delphi, and .NET. In Perl, `$_` is the default variable that the regex is applied to if you use a regular expression without the matching operator `=~`. `_` is just an escaped underscore. It has no special meaning in any application.

Boost 1.42 added some alternative syntax of its own invention. `$PREMATCH` and `${^PREMATCH}` are synonyms for `$^`. `$POSTMATCH` and `${^POSTMATCH}` are synonyms for `$'`.

47. Replacement Text Case Conversion

Some applications can insert the text matched by the regex or by capturing groups converted to uppercase or lowercase. The Just Great Software applications allow you to prefix the matched text token `\0` and the backreferences `\1` through `\99` with a letter that changes the case of the inserted text. `U` is for uppercase, `L` for lowercase, `I` for initial capitals (first letter of each word is uppercase, rest is lowercase), and `F` for first capital (first letter in the inserted text is uppercase, rest is lowercase). The letter only affects the case of the backreference that it is part of.

When the regex `(?i)(Hello) (World)` matches `HeLLó WóRlD` the replacement text `\U1 \L2 \I0 \F0` becomes `HELLó wóRlD Helló WóRlD Helló wóRlD`.

Perl String Features in Regular Expressions and Replacement Texts

The double-slashed and triple-slashed notations for regular expressions and replacement texts in Perl support all the features of double-quoted strings. Most obvious is variable interpolation. You can insert the text matched by the regex or capturing groups simply by using the regex-related variables in your replacement text.

Perl's case conversion escapes also work in replacement texts. The most common use is to change the case of an interpolated variable. `\U` converts everything up to the next `\L` or `\E` to uppercase. `\L` converts everything up to the next `\U` or `\E` to lowercase. `\u` converts the next character to uppercase. `\l` converts the next character to lowercase. You can combine these into `\l\U` to make the first character lowercase and the remainder uppercase, or `\u\L` to make the first character uppercase and the remainder lowercase. `\E` turns off case conversion. You cannot use `\u` or `\l` after `\U` or `\L` unless you first stop the sequence with `\E`.

When the regex `(?i)(hello) (world)` matches `HeLLó WóRlD` the replacement text `\U\1\E \L\u$2` becomes `hELLó wóRlD`. Literal text is also affected. `\U$1 Dear $2` becomes `HELLó DEAR WóRlD`.

Perl's case conversion works in regular expressions too. But it doesn't work the way you might expect. Perl applies case conversion when it parses a string in your script and interpolates variables. That works great with backreferences in replacement texts, because those are really interpolated variables in Perl. But backreferences in the regular expression are regular expression tokens rather than variables. `(?-i)(a)\U1` matches `aa` but not `aA`. `\1` is converted to uppercase while the regex is parsed, not during the matching process. Since `\1` does not include any letters, this has no effect. In the regex `\U\w`, `\w` is converted to uppercase while the regex is parsed. This means that `\U\w` is the same as `\W`, which matches any character that is not a word character.

Boost's Replacement String Case Conversion

Boost supports case conversion in replacement strings when using the default replacement format or the "all" replacement format. `\U` converts everything up to the next `\L` or `\E` to uppercase. `\L` converts everything up to the next `\U` or `\E` to lowercase. `\u` converts the next character to uppercase. `\l` converts the next character to lowercase. `\E` turns off case conversion. As in Perl, the case conversion affects both literal text in your replacement string and the text inserted by backreferences.

where Boost differs from Perl is that combining these needs to be done the other way around. `\U\l` makes the first character lowercase and the remainder uppercase. `\L\u` makes the first character uppercase and the remainder lowercase. Boost also allows `\l` inside a `\U` sequence and a `\u` inside a `\L` sequence. So when `(?i)(helló)(wórlđ)` matches `HeLLó WóRlD` you can use `\L\u\1 \u\2` to replace the match with `HeLLó Wórlđ`.

PCRE2's Replacement String Case Conversion

PCRE2 supports case conversion in replacement strings when using `PCRE2_SUBSTITUTE_EXTENDED`. `\U` converts everything that follows to uppercase. `\L` converts everything that follows to lowercase. `\u` converts the next character to uppercase. `\l` converts the next character to lowercase. `\E` turns off case conversion. As in Perl, the case conversion affects both literal text in your replacement string and the text inserted by backreferences.

Unlike in Perl, in PCRE2 `\U`, `\L`, `\u`, and `\l` all stop any preceding case conversion. So you cannot combine `\L` and `\u`, for example, to make the first character uppercase and the remainder lowercase. `\L\u` makes the first character uppercase and leaves the rest unchanged, just like `\u`. `\u\L` makes all characters lowercase, just like `\L`.

In PCRE2, case conversion runs through conditionals. Any case conversion in effect before the conditional also applies to the conditional. If the conditional contains its own case conversion escapes in the part of the conditional that is actually used, then those remain in effect after the conditional. So you could use `${1:+\U:\L}${2}` to insert the text matched by the second capturing group in uppercase if the first group participated, and in lowercase if it didn't.

R's Backreference Case Conversion

The `sub()` and `gsub()` functions in R support case conversion escapes that are inspired by Perl strings. `\U` converts all backreferences up to the next `\L` or `\E` to uppercase. `\L` converts all backreferences up to the next `\U` or `\E` to lowercase. `\E` turns off case conversion.

When the regex `(?i)(Helló)(wórlđ)` matches `HeLLó WóRlD` the replacement string `\U$1 \L$2` becomes `HeLLó wórlđ`. Literal text is not affected. `\U$1 Dear $2` becomes `HeLLó Dear WóRlD`.

48. Replacement String Conditionals

Replacement string conditionals allow you to use one replacement when a particular capturing group participated in the match and another replacement when that capturing group did not participate in the match. They are supported by JGsoft V2, Boost, and PCRE2. Boost and PCRE2 each invented their own syntax. JGsoft V2 supports both.

For conditionals to work in Boost, you need to pass `regex_constants::format_all` to `regex_replace`. For them to work in PCRE2, you need to pass `PCRE2_SUBSTITUTE_EXTENDED` to `pcre2_substitute`.

Boost Replacement String Conditionals

Boost's syntax is `(?1matched:unmatched)` where `1` is a number between 1 and 99 referencing a numbered capturing group. `matched` is used as the replacement for matches in which the capturing group participated. `unmatched` is used for matches in which the group did not participate. The colon `:` delimits the two parts. If you want a literal colon in the `matched` part, then you need to escape it with a backslash. If you want a literal closing parenthesis anywhere in the conditional, then you need to escape that with a backslash too.

The parentheses delimit the conditional from the remainder of the replacement string. `start(?1matched:unmatched)finish` replaces with `startmatchedfinish` when the group participates and with `startunmatchedfinish` when it doesn't. JGsoft V2 requires the parentheses. Boost allows you to omit the parentheses if nothing comes after the conditional in the replacement. So `?1matched:unmatched` is the same as `(?1matched:unmatched)`.

The `matched` and `unmatched` parts can be blank. You can omit the colon if the `unmatched` part is blank. So `(?1matched:)` and `(?1matched)` replace with `matched` when the group participates. They replace the match with nothing when the group does not participate.

You can use the full replacement string syntax in `matched` and `unmatched`. This means you can nest conditionals inside other conditionals. So `(?1one(?2two):(?2two:none))` replaces with `onetwo` when both groups participate, with `one` or `two` when group 1 or 2 participates and the other doesn't, and with `none` when neither group participates. With Boost `?1one(?2two):?2two:none` does exactly the same but omits parentheses that aren't needed.

The JGsoft V2 regex flavor treats conditionals that reference non-existing capturing groups as an error. If there are two digits after the question mark but not enough capturing groups for a two-digit conditional to be valid, then only the first digit is used for the conditional and the second digit is a literal. So when there are less than 12 capturing groups in the regex, `(?12matched)` replaces with `2matched` when capturing group 1 participates in the match.

Boost treats conditionals that reference a non-existing group number as conditionals to a group that never participates in the match. So `(?12twelve:not twelve)` always replaces with `not twelve` when there are fewer than 12 capturing groups in the regex.

You can avoid the ambiguity between single digit and double digit conditionals by placing curly braces around the number. `(?{1}1:0)` replaces with `1` when group 1 participates and with `0` when it doesn't, even if there are 11 or more capturing groups in the regex. `(?{12}twelve:not twelve)` is always a conditional that

references group 12, even if there are fewer than 12 groups in the regex (which may make the conditional invalid).

The syntax with curly braces also allows you to reference named capturing groups by their names. `{?name}matched:unmatched` replaces with `matched` when the group “name” participates in the match and with `unmatched` when it doesn’t. If the group does not exist, the JGsoft V2 regex flavor treats the conditionals as an error. Boost, however, treats conditionals that reference a non-existing group name as literals. So `{?nonexisting}matched:unmatched` uses `{nonexisting}matched:unmatched` as a literal replacement.

PCRE2 Replacement String Conditional

PCRE2’s syntax is `{1:+matched:unmatched}` where `1` is a number between 1 and 99 referencing a numbered capturing group. If your regex contains named capturing groups then you can reference them in a conditional by their name: `{name:+matched:unmatched}`.

`matched` is used as the replacement for matches in which the capturing group participated. `unmatched` is used for matches in which the group did not participate. `:+` delimits the group number or name from the first part of the conditional. The second colon delimits the two parts. If you want a literal colon in the `matched` part, then you need to escape it with a backslash. If you want a literal closing curly brace anywhere in the conditional, then you need to escape that with a backslash too. Plus signs have no special meaning beyond the `:+` that starts the conditional, so they don’t need to be escaped.

You can use the full replacement string syntax in `matched` and `unmatched`. This means you can nest conditionals inside other conditionals. So `{1:+one{2:+two}{2:+two:none}}` replaces with `onetwo` when both groups participate, with `one` or `two` when group 1 or 2 participates and the other doesn’t, and with `none` when neither group participates.

`{1:-unmatched}` and `{name:-unmatched}` are shorthands for `{1:+{1}:unmatched}` and `{name:+{name}:unmatched}`. They insert the text captured by the group if it participated in the match. They insert `unmatched` if the group did not participate. When using this syntax, `:-` delimits the group number or name from the contents of the conditional. The conditional has only one part in which colons and minus signs have no special meaning.

Both PCRE2 and JGsoft V2 treat conditionals that reference non-existing capturing groups as an error.

Escaping Question Marks, Colons, Parentheses, and Curly Braces

As explained above, you need to use backslashes to escape colons that you want to use as literals when used in the `matched` part of the conditional. You also need to escape literal closing parentheses (Boost) or curly braces (PCRE2) with backslashes inside conditionals.

In replacement string flavors that support conditionals, you can escape colons, parentheses, curly braces, and even question marks with backslashes to make sure they are interpreted as literals anywhere in the replacement string. But generally there is no need to.

The colon does not have any special meaning in the `unmatched` part or outside conditionals. So you don't need to escape it there. The question mark does not have any special meaning if it is not followed by a digit or a curly brace. In PCRE2 it never has a special meaning. So you only need to escape question marks with backslashes if you want to use a literal question mark followed by a literal digit or curly brace as the replacement in Boost or JGsoftV2.

In the JGsoft V2 flavor, opening parentheses are part of the syntax for conditionals. The first unescaped closing parenthesis that follows it then ends the conditional. All other unescaped opening and closing parentheses are literals.

Boost always uses parentheses for grouping. An unescaped opening parenthesis always opens a group. Groups can be nested. An unescaped closing parenthesis always closes a group. An unescaped closing parenthesis that does not have a matching opening parenthesis effectively truncates the replacement string. So Boost requires you to always escape literal parentheses with backslashes.

Part 3

Regular Expressions 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.

Grabbing HTML Tags

`<TAG\b[^>]*>(.*?)</TAG>` matches the opening and closing pair of a specific HTML tag. Anything between the tags is captured into the first backreference. The question mark in the regex makes the star lazy, to make sure it stops before the first closing tag rather than before the last, like a greedy star would do. This regex will not properly match tags nested inside themselves, like in `<TAG>one<TAG>two</TAG>one</TAG>`.

`<([A-Z][A-Z0-9]*)\b[^>]*>(.*?)</\1>` will match the opening and closing pair of any HTML tag. Be sure to turn off case sensitivity. The key in this solution is the use of the backreference `\1` in the regex. Anything between the tags is captured into the second backreference. This solution will also not match tags nested in themselves.

Trimming Whitespace

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

More Detailed Examples

Numeric Ranges. Since regular expressions work with text rather than numbers, matching specific numeric ranges requires a bit of extra care.

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

Matching an Email Address. There's a lot of controversy about what is a proper regex to match email addresses. It's a perfect example showing that you need to know exactly what you're trying to match (and what not), and that there's always a trade-off between regex complexity and accuracy.

Matching an IP Address.

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

Finding or Verifying Credit Card Numbers. Validate credit card numbers entered on your order form. Find credit card numbers in documents for a security audit.

Matching Complete Lines. Shows how to match complete lines in a text file rather than just the part of the line that satisfies a certain requirement. Also shows how to match lines in which a particular regex does *not* match.

Removing Duplicate Lines or Items. Illustrates simple yet clever use of capturing parentheses or backreferences.

Regex Examples for Processing Source Code. How to match common programming language syntax such as comments, strings, numbers, etc.

Two Words Near Each Other. Shows how to use a regular expression to emulate the “near” operator that some tools have.

Common Pitfalls

Catastrophic Backtracking. If your regular expression seems to take forever, or simply crashes your application, it has likely contracted a case of catastrophic backtracking. The solution is usually to be more specific about what you want to match, so the number of matches the engine has to try doesn't rise exponentially.

Making Everything Optional. If all the parts in your regex are optional, it will match a zero-length string anywhere. Your regex will need to express the facts that different parts are optional depending on which parts are present.

Repeating a Capturing Group vs. Capturing a Repeated Group. Repeating a capturing group will capture only the last iteration of the group. Capture a repeated group if you want to capture all iterations.

Mixing Unicode and 8-bit Character Codes. Using 8-bit character codes like `\x80` with a Unicode engine and subject string may give unexpected results.

2. Matching Numeric Ranges with a Regular Expression

Since regular expressions deal with text rather than with numbers, matching a number in a given range takes a little extra care. You can't just write `[0-255]` to match a number between 0 and 255. Though a valid regex, it matches something entirely different. `[0-255]` is a character class with three elements: the character range 0-2, the character 5 and the character 5 (again). This character class matches a single digit 0, 1, 2 or 5, just like `[0125]`.

Since regular expressions work with text, a regular expression engine treats `0` as a single character, and `255` as three characters. To match all characters from 0 to 255, we'll need a regex that matches between one and three characters.

The regex `[0-9]` matches single-digit numbers 0 to 9. `[1-9][0-9]` matches double-digit numbers 10 to 99. That's the easy part.

Matching the three-digit numbers is a little more complicated, since we need to exclude numbers 256 through 999. `1[0-9][0-9]` takes care of 100 to 199. `2[0-4][0-9]` matches 200 through 249. Finally, `25[0-5]` adds 250 till 255.

As you can see, you need to split up the numeric range in ranges with the same number of digits, and each of those ranges that allow the same variation for each digit. In the 3-digit range in our example, numbers starting with 1 allow all 10 digits for the following two digits, while numbers starting with 2 restrict the digits that are allowed to follow.

Putting this all together using alternation we get: `[0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5]`. This matches the numbers we want, with one caveat: regular expression searches usually allow partial matches, so our regex would match `123` in `12345`. There are two solutions to this.

If you're searching for these numbers in a larger document or input string, use word boundaries to require a non-word character (or no character at all) to precede and to follow any valid match. The regex then becomes `\b([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\b`. Since the alternation operator has the lowest precedence of all, the parentheses are required to group the alternatives together. This way the regex engine will try to match the first word boundary, then try all the alternatives, and then try to match the second word boundary after the numbers it matched. Regular expression engines consider all alphanumeric characters, as well as the underscore, as word characters.

If you're using the regular expression to validate input, you'll probably want to check that the entire input consists of a valid number. To do this, replace the word boundaries with anchors to match the start and end of the string: `^([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])$`.

Here are a few more common ranges that you may want to match:

- 000..255: `^([01][0-9][0-9]|2[0-4][0-9]|25[0-5])$`
- 0 or 000..255: `^([01]?[0-9]?[0-9]|2[0-4][0-9]|25[0-5])$`
- 0 or 000..127: `^(0?[0-9]?[0-9]|1[01][0-9]|12[0-7])$`
- 0..999: `^([0-9]|[1-9][0-9]|1[0-9][0-9])$`
- 000..999: `^[0-9]{3}$`
- 0 or 000..999: `^[0-9]{1,3}$`
- 1..999: `^([1-9]|[1-9][0-9]|1[0-9][0-9])$`

- 001..999: `^(00[1-9]|0[1-9][0-9]|[1-9][0-9][0-9])$`
- 1 or 001..999: `^(0{0,2}[1-9]|0?[1-9][0-9]|[1-9][0-9][0-9])$`
- 0 or 00..59: `^[0-5]?[0-9]$`
- 0 or 000..366: `^([012]?[0-9]?[0-9]|3[0-5][0-9]|36[0-6])$`

3. Matching Floating Point Numbers with a Regular Expression

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

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

Spelling out the regex in words makes it obvious: everything in this regular expression is optional. This regular expression considers a sign by itself or a dot by itself as a valid floating point number. In fact, it even considers an empty string as a valid floating point number. If you tried to use this regex to find floating point numbers in a file, you'd get a zero-length match at every position in the string where no floating point number occurs.

Not escaping the dot is also a common mistake. A dot that is not escaped matches any character, including a dot. If we had not escaped the dot, both `4.4` and `4X4` would be considered floating point numbers.

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

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

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

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

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

Finally, if you want to validate if a particular string holds a floating point number, rather than finding a floating point number within longer text, you'll have to anchor your regex: `^[-+]? [0-9]* \. ? [0-9]+ $` or `^[-+]? [0-9]* \. ? [0-9]+ ([eE] [-+]? [0-9]+)? $`. You can find additional variations of these regexes in RegxBuddy's library.

4. How to Find or Validate an Email Address

The regular expression I receive the most feedback, not to mention “bug” reports on, is the one you’ll find right in the tutorial’s introduction: `\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}\b`. This regular expression, I claim, matches any email address. Most of the feedback I get refutes that claim by showing one email address that this regex doesn’t match. Usually, the “bug” report also includes a suggestion to make the regex “perfect”.

As I explain below, my claim only holds true when one accepts my definition of what a valid email address really is, and what it’s not. If you want to use a different definition, you’ll have to adapt the regex. Matching a valid email address is a perfect example showing that (1) before writing a regex, you have to know exactly what you’re trying to match, and what not; and (2) there’s often a trade-off between what’s exact, and what’s practical.

The virtue of my regular expression above is that it matches 99% of the email addresses in use today. All the email addresses it matches can be handled by 99% of all email software out there. If you’re looking for a quick solution, you only need to read the next paragraph. If you want to know all the trade-offs and get plenty of alternatives to choose from, read on.

If you want to use the regular expression above, there are two things you need to understand. First, long regexes make it difficult to nicely format paragraphs. So I didn’t include `a-z` in any of the three character classes. This regex is intended to be used with your regex engine’s “case insensitive” option turned on. (You’d be surprised how many “bug” reports I get about that.) Second, the above regex is delimited with word boundaries, which makes it suitable for extracting email addresses from files or larger blocks of text. If you want to check whether the user typed in a valid email address, replace the word boundaries with start-of-string and end-of-string anchors, like this: `^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}$`.

The previous paragraph also applies to all of the following examples. You may need to change word boundaries into start/end-of-string anchors, or vice versa. And you have to turn on the case insensitive matching option.

Trade-Offs in Validating Email Addresses

Before ICANN made it possible for any well-funded company to create their own top-level domains, the longest top-level domains were the rarely used `.museum` and `.travel` which are 6 letters long. The most common top-level domains were 2 letters long for country-specific domains, and 3 or 4 letters long for general-purpose domains like `.com` and `.info`. A lot of regexes for validating email addresses you’ll find in various regex tutorials and references still assume the top-level domain to be fairly short. Older editions of this regex tutorial mentioned `\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b` as the regex for email addresses in its introduction. There’s only one little difference between this regex and the one at the top of this page. The `4` at the end of the regex restricts the top-level domain to 4 characters. If you use this regex with anchors to validate the email address entered on your order form, `fabio@disapproved.solutions` has to do his shopping elsewhere. Yes, the `.solutions` TLD exists and when I write this, `disapproved.solutions` can be yours for \$16.88 per year.

If you want to be more strict than `[A-Z]{2,}` for the top-level domain, `^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,63}$` is as far as you can practically go. Each part of a domain name can be no longer than 63 characters. There are no single-digit top-level domains and none contain digits. It doesn’t look like ICANN will approve such domains either.

Email addresses can be on servers on a subdomain as in `john@server.department.company.com`. All of the above regexes match this email address, because I included a dot in the character class after the `@` symbol. But the above regexes also match `john@aol...com` which is not valid due to the consecutive dots. You can exclude such matches by replacing `[A-Z0-9.-]+\.` with `(?:[A-Z0-9-]+\.)+` in any of the above regexes. I removed the dot from the character class and instead repeated the character class and the following literal dot. E.g. `^[A-Z0-9._%+]+@(?:[A-Z0-9-]+\.)+[A-Z]{2,}$` matches `john@server.department.company.com` but not `john@aol...com`.

If you want to avoid your system choking on arbitrarily large input, you can replace the infinite quantifiers with finite ones. `^[A-Z0-9._%+]{1,64}@(?:[A-Z0-9-]{1,63}\.){1,125}[A-Z]{2,63}$` takes into account that the local part (before the `@`) is limited to 64 characters and that each part of the domain name is limited to 63 characters. There's no direct limit on the number of subdomains. But the maximum length of an email address that can be handled by SMTP is 254 characters. So with a single-character local part, a two-letter top-level domain and single-character sub-domains, 125 is the maximum number of sub-domains.

The previous regex does not actually limit email addresses to 254 characters. If each part is at its maximum length, the regex can match strings up to 8129 characters in length. You can reduce that by lowering the number of allowed sub-domains from 125 to something more realistic like 8. I've never seen an email address with more than 4 subdomains. If you want to enforce the 254 character limit, the best solution is to check the length of the input string before you even use a regex. Though this requires a few lines of procedural code, checking the length of a string is near-instantaneous. If you can only use regexes, `^[A-Z0-9@._%+]{6,254}$` can be used as a first pass to make sure the string doesn't contain invalid characters and isn't too short or too long. If you need to do everything with one regex, you'll need a regex flavor that supports lookahead. The regular expression `^(?=[A-Z0-9@._%+]{6,254}$)[A-Z0-9._%+]{1,64}@(?:[A-Z0-9-]{1,63}\.){1,8}[A-Z]{2,63}$` uses a lookahead to first check that the string doesn't contain invalid characters and isn't too short or too long. When the lookahead succeeds, the remainder of the regex makes a second pass over the string to check for proper placement of the `@` sign and the dots.

All of these regexes allow the characters `._%+` anywhere in the local part. You can force the local part to begin with a letter by using `^[A-Z0-9][A-Z0-9._%+]{0,63}` instead of `^[A-Z0-9._%+]{1,64}` for the local part: `^[A-Z0-9][A-Z0-9._%+]{0,63}@(?:[A-Z0-9-]{1,63}\.){1,125}[A-Z]{2,63}$`. When using lookahead to check the overall length of the address, the first character can be checked in the lookahead. We don't need to repeat the initial character check when checking the length of the local part. This regex is too long to fit the width of the page, so let's turn on free-spacing mode:

```
^(?=[A-Z0-9][A-Z0-9@._%+]{5,253}$)
[A-Z0-9._%+]{1,64}@(?:[A-Z0-9-]{1,63}\.){1,8}[A-Z]{2,63}$
```

Domain names can contain hyphens. But they cannot begin or end with a hyphen. `[A-Z0-9](?:[A-Z0-9-]{0,61}[A-Z0-9])?` matches a domain name between 1 and 63 characters long that starts and ends with a letter or digit. The non-capturing group makes the middle of the domain and the final letter or digit optional as a whole to ensure that we allow single-character domains while at the same time ensuring that domains with two or more characters do not end with a hyphen. The overall regex starts to get quite complicated:

```
^[A-Z0-9][A-Z0-9._%+]{0,63}@
(?:[A-Z0-9](?:[A-Z0-9-]{0,61}[A-Z0-9])?\.){1,8}[A-Z]{2,63}$
```

Domain names cannot contain consecutive hyphens. `[A-Z0-9]+(?:-[A-Z0-9]+)*` matches a domain name that starts and ends with a letter or digit and that contains any number of non-consecutive hyphens. This is the most efficient way. This regex does not do any backtracking to match a valid domain name. It matches all letters and digits at the start of the domain name. If there are no hyphens, the optional group that

follows fails immediately. If there are hyphens, the group matches each hyphen followed by all letters and digits up to the next hyphen or the end of the domain name. We can't enforce the maximum length when hyphens must be paired with a letter or digit, but letters and digits can stand on their own. But we can use the lookahead technique that we used to enforce the overall length of the email address to enforce the length of the domain name while disallowing consecutive hyphens: `(?=[A-Z0-9-]{1,63}\.)[A-Z0-9]+(?:-[A-Z0-9-]+)*`. Notice that the lookahead also checks for the dot that must appear after the domain name when it is fully qualified in an email address. This is important. Without checking for the dot, the lookahead would accept longer domain names. Since the lookahead does not consume the text it matches, the dot is not included in the overall match of this regex. When we put this regex into the overall regex for email addresses, the dot will be matched as it was in the previous regexes:

```
^[A-Z0-9][A-Z0-9. %+-]{0,63}@
(?: (?:[A-Z0-9-]{1,63}\. ) [A-Z0-9]+(?:-[A-Z0-9-]+)*\.) {1,8} [A-Z]{2,63}$
```

If we include the lookahead to check the overall length, our regex makes two passes over the local part, and three passes over the domain names to validate everything:

```
^(?=[A-Z0-9][A-Z0-9@. %+-]{5,253}$)[A-Z0-9. %+-]{1,64}@
(?: (?:[A-Z0-9-]{1,63}\. ) [A-Z0-9]+(?:-[A-Z0-9-]+)*\.) {1,8} [A-Z]{2,63}$
```

On a modern PC or server this regex will perform just fine when validating a single 254-character email address. Rejecting longer input would even be faster because the regex will fail when the lookahead fails during first pass. But I wouldn't recommend using a regex as complex as this to search for email addresses through a large archive of documents or correspondence. You're better off using the simple regex at the top of this page to quickly gather everything that looks like an email address. Deduplicate the results and then use a stricter regex if you want to further filter out invalid addresses.

And speaking of backtracking, none of the regexes on this page do any backtracking to match valid email addresses. But particularly the latter ones may do a fair bit of backtracking on something that's not quite a valid email address. If your regex flavor supports possessive quantifiers, you can eliminate all backtracking by making all quantifiers possessive. Because no backtracking is needed to find matches, doing this does not change what is matched by these regexes. It only allows them to fail faster when the input is not a valid email address. The simplest regex that correctly handles subdomains then becomes `^[A-Z0-9. %+-]++@(?:[A-Z0-9-]++\.)++[A-Z]{2,}+$` with an extra + after each quantifier. We can do the same with our most complex regex:

```
^(?=[A-Z0-9][A-Z0-9@. %+-]{5,253}++$)[A-Z0-9. %+-]{1,64}++@
(?: (?:[A-Z0-9-]{1,63}\. ) [A-Z0-9]++(?:-[A-Z0-9-]++)*\.) {1,8}+[A-Z]{2,63}++$
```

An important trade-off in all these regexes is that they only allow English letters, digits, and the most commonly used special symbols. The main reason is that I don't trust all my email software to be able to handle much else. Even though `John.O'Hara@theoharas.com` is a syntactically valid email address, there's a risk that some software will misinterpret the apostrophe as a delimiting quote. Blindly inserting this email address into an SQL query, for example, will at best cause it to fail when strings are delimited with single quotes and at worst open your site up to SQL injection attacks.

And of course, it's been many years already that domain names can include non-English characters. But most software still sticks to the 37 characters Western programmers are used to. Supporting internationalized domains opens up a whole can of worms of how the non-ASCII characters should be encoded. So if you use any of the regexes on this page, anyone with an `@ทีเอชบี.ไทย` address will be out of luck. But perhaps it is

telling that `http://ทีเอชเน็ต.ไทย` simply redirects to `http://thnic.co.th` even though they're in the business of selling `.ไทย` domains.

The conclusion is that to decide which regular expression to use, whether you're trying to match an email address or something else that's vaguely defined, you need to start with considering all the trade-offs. How bad is it to match something that's not valid? How bad is it not to match something that is valid? How complex can your regular expression be? How expensive would it be if you had to change the regular expression later because it turned out to be too broad or too narrow? Different answers to these questions will require a different regular expression as the solution. My email regex does what I want, but it may not do what you want.

Regexes Don't Send Email

Don't go overboard in trying to eliminate invalid email addresses with your regular expression. The reason is that you don't really know whether an address is valid until you try to send an email to it. And even that might not be enough. Even if the email arrives in a mailbox, that doesn't mean somebody still reads that mailbox. If you really need to be sure an email address is valid, you'll need to send an email to it that contains a code or link for the recipient to perform a second authentication step. And if you're doing that, then there is little point in using a regex that may reject valid email addresses.

The same principle applies in many situations. When trying to match a valid date, it's often easier to use a bit of arithmetic to check for leap years, rather than trying to do it in a regex. Use a regular expression to find potential matches or check if the input uses the proper syntax, and do the actual validation on the potential matches returned by the regular expression. Regular expressions are a powerful tool, but they're far from a panacea.

The Official Standard: RFC 5322

Maybe you're wondering why there's no "official" fool-proof regex to match email addresses. Well, there is an official definition, but it's hardly fool-proof.

The official standard is known as RFC 5322. It describes the syntax that valid email addresses must adhere to. You can (but you shouldn't—read on) implement it with the following regular expression. RFC 5322 leaves the domain name part open to implementation-specific choices that won't work on the Internet today. The regex implements the "preferred" syntax from RFC 1035 which is one of the recommendations in RFC 5322:

```
\A(?:[a-z0-9!#$%&'*/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*/=?^_`{|}~-]+)*)
| "(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]
| \\[\x01-\x09\x0b\x0c\x0e-\x7f])*"
@ (?: (?: [a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?
| \[(?: (?: 25[0-5]|2[0-4][0-9]|01?[0-9])[0-9]?[0-9]?\.){3}
| (?: 25[0-5]|2[0-4][0-9]|01?[0-9])[0-9]?[0-9]?\.[a-z0-9-]*[a-z0-9]:
| (?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]
| \\[\x01-\x09\x0b\x0c\x0e-\x7f])+)
\])\z
```

This regex has two parts: the part before the `@`, and the part after the `@`. There are two alternatives for the part before the `@`. The first alternative allows it to consist of a series of letters, digits and certain symbols,

including one or more dots. However, dots may not appear consecutively or at the start or end of the email address. The other alternative requires the part before the @ to be enclosed in double quotes, allowing any string of ASCII characters between the quotes. Whitespace characters, double quotes and backslashes must be escaped with backslashes.

The part after the @ also has two alternatives. It can either be a fully qualified domain name (e.g. regular-expressions.info), or it can be a literal Internet address between square brackets. The literal Internet address can either be an IP address, or a domain-specific routing address.

The reason you shouldn't use this regex is that it is overly broad. Your application may not be able to handle all email addresses this regex allows. Domain-specific routing addresses can contain non-printable ASCII control characters, which can cause trouble if your application needs to display addresses. Not all applications support the syntax for the local part using double quotes or square brackets. In fact, RFC 5322 itself marks the notation using square brackets as obsolete.

We get a more practical implementation of RFC 5322 if we omit IP addresses, domain-specific addresses, the syntax using double quotes and square brackets. It will still match 99.99% of all email addresses in actual use today.

```
\A[a-z0-9!#$%&'*/+=?^_`{|}~-]+(?:\. [a-z0-9!#$%&'*/+=?^_`{|}~-]+)*@
(?: [a-z0-9](?: [a-z0-9-]* [a-z0-9])?\.)+[a-z0-9](?: [a-z0-9-]* [a-z0-9])?\z
```

Neither of these regexes enforce length limits on the overall email address or the local part or the domain names. RFC 5322 does not specify any length limitations. Those stem from limitations in other protocols like the SMTP protocol for actually sending email. RFC 1035 does state that domains must be 63 characters or less, but does not include that in its syntax specification. The reason is that a true regular language cannot enforce a length limit and disallow consecutive hyphens at the same time. But modern regex flavors aren't truly regular, so we can add length limit checks using lookahead like we did before:

```
\A(?:[a-z0-9@. !#$%&'*/+=?^_`{|}~-]{6,254}\z)
(?:[a-z0-9. !#$%&'*/+=?^_`{|}~-]{1,64}@)
[a-z0-9!#$%&'*/+=?^_`{|}~-]+(?:\. [a-z0-9!#$%&'*/+=?^_`{|}~-]+)*
@ (?: (?: [a-z0-9-]{1,63}\. ) [a-z0-9] (?: [a-z0-9-]* [a-z0-9])?\.\. )+
(?: [a-z0-9-]{1,63}\z) [a-z0-9] (?: [a-z0-9-]* [a-z0-9])?\z
```

So even when following official standards, there are still trade-offs to be made. Don't blindly copy regular expressions from online libraries or discussion forums. Always test them on your own data and with your own applications.

5. How to Find or Validate an IP Address

Matching an IP address is another good example of a trade-off between regex complexity and exactness. `\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b` will match any IP address just fine. But will also match `999.999.999.999` as if it were a valid IP address. If your regex flavor supports Unicode, it may even match `١٢٣.٩٢٣.٠٩٩.٠٩٩`. Whether this is a problem depends on the files or data you intend to apply the regex to.

Restricting and Capturing The Four IP Address Numbers

To restrict all 4 numbers in the IP address to 0..255, you can use the following regex. It stores each of the 4 numbers of the IP address into a capturing group. You can use these groups to further process the IP number. Free-spacing mode allows this to fit the width of the page.

```
\b(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b
```

The above regex allows one leading zero for numbers 10 to 99 and up to two leading zeros for numbers 0 to 9. Strictly speaking, IP addresses with leading zeros imply octal notation. So you may want to disallow leading zeros. This requires a slightly longer regex:

```
\b(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\b
```

Restricting The Four IP Address Numbers Without Capturing Them

If you don't need access to the individual numbers, you can shorten above 3 regexes with a quantifier to:

```
\b(?:\d{1,3}\.){3}\d{1,3}\b
```

```
\b(?: (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\. ){3}
```

```
\b(?: (?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\. ){3}
```

Checking User Input

The above regexes use word boundaries to make sure the first and last number in the IP address aren't part of a longer sequence of alphanumeric characters. These regexes are appropriate for finding IP addresses in longer strings.

If you want to validate user input by making sure a string consists of nothing but an IP address then you need to replace the word boundaries with start-of-string and end-of-string anchors. You can use the dedicated anchors `\A` and `\Z` if your regex flavor supports them:

```
\A(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\Z
```

If not, you'll have to use `^` and `$` and make sure that the option for them to match at line breaks is off:

```
^(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$
```

6. Matching a Valid Date

`^(19|20)\d\d[- /.](0[1-9]|1[012])[- /.](0[1-9]|12|[0-9]|3[01])$` matches a date in yyyy-mm-dd format from 1900-01-01 through 2099-12-31, with a choice of four separators. The anchors make sure the entire variable is a date, and not a piece of text containing a date. The year is matched by `(19|20)\d\d`. I used alternation to allow the first two digits to be 19 or 20. The parentheses are mandatory. Had I omitted them, the regex engine would go looking for 19 or the remainder of the regular expression, which matches a date between 2000-01-01 and 2099-12-31. Parentheses are the only way to stop the vertical bar from splitting up the entire regular expression into two options.

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

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

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

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

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

Here is how you could check a valid date in Perl. I also added parentheses to capture the year into a backreference.

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

To match a date in mm/dd/yyyy format, rearrange the regular expression to `^(0[1-9]|1[012])[- /.](0[1-9]|1[12][0-9]|3[01])[- /.](19|20)\d\d$`. For dd-mm-yyyy format, use `^(0[1-9]|1[12][0-9]|3[01])[- /.](0[1-9]|1[012])[- /.](19|20)\d\d$`. You can find additional variations of these regexes in RegexBuddy's library.

7. Replacing Numerical Dates with Textual Dates

This example shows how you can replace numerical dates from 1/1/50 or 01/01/50 through 12/31/49 with their textual equivalents from January 1st, 1950 through December 31st, 2049. This is only possible with a single regular expression if you can vary the replacement based on what was matched. One way to do this is to build each replacement in procedural code. This example shows how you can do it using replacement string conditionals. This example works can be used with PowerGREP 5, the Boost C++ library, and the PCRE2 C library.

To be able to use replacement string conditionals, the regular expression needs a separate capturing group for each part of the match that needs a different replacement. Each month needs to be replaced with its own name, so we need a separate capturing group to match each month number. Cardinal numbers ending with 1, 2, and 3 have unique suffixes. So we need four groups to match day numbers ending with 1, 2, 3, or another digit. We're assuming year numbers 50 to 99 to be 1950 to 1999, and year numbers 00 to 49 to be 2000 to 2049. So we need two more groups to match each half century.

Putting this all together results in a rather long regular expression. Free-spacing helps to keep it readable. The structure of the regex is the same as what you would use for matching valid dates. It's only more long-winded because we need 12 alternatives to match the month, 4 alternatives to match the day, and 2 alternatives to match the year.

```
\b
(?: # Month
  ('jan'|'0?1')|('feb'|'0?2')|('mar'|'0?3')|('apr'|'0?4')|('may'|'0?5')|('jun'|'0?6')
  |('jul'|'0?7')|('aug'|'0?8')|('sep'|'0?9')|('oct'|'10')|('nov'|'11')|('dec'|'12')
) /
0?(?: # Day
  ('1st'|'[23]?1')|('2nd'|'2?2')|('3rd'|'2?3')|('nth'|'30|1[123]|'[12]?[4-9]')
) /
(?: # Year
  ('19xx'|'[5-9][0-9]')|('20xx'|'[0-4][0-9]')
)
\b
```

The replacement string will use backreferences to reinsert the date numbers. Since we want to omit leading zeros from the replacements, we placed `0?` outside the capturing groups for date numbers. This means that our regex also allows leading zeros for days 10 to 31. Since our goal is to replace dates rather than validate them, we can live with this. Otherwise, we would need two sets of four alternatives to match the day of the month. One set for single digit days, and one set for double digit days.

Unfortunately, free-spacing does not work with replacement strings. So the replacement consists of one very long line. It is broken into multiple lines here to fit the width of the page. This is the replacement using Boost syntax:

```
(?{jan}January)(?{feb}February)(?{mar}March)(?{apr}April)(?{may}May)(?{jun}June)
(?{jul}July)(?{aug}August)(?{sep}September)(?{oct}October)(?{nov}November)
(?{dec}December) (?{1st}${1st}st)(?{2nd}${2nd}nd)(?{3rd}${3rd}rd)(?{nth}${nth}th)
, (?{19xx}19${19xx})(?{20xx}20${20xx})
```

This is the replacement using PCRE2 syntax:

```

${jan:+January}${feb:+February}${mar:+March}${apr:+April}${may:+May}${jun:+June}
${jul:+July}${aug:+August}${sep:+September}${oct:+October}${nov:+November}
${dec:+December} ${1st:+${1st}st}${2nd:+${2nd}nd}${3rd:+${3rd}rd}${nth:+${nth}th}
, ${19xx:+19${19xx}}${20xx:+20${20xx}}

```

First we have 12 conditionals that reference the 12 capturing groups for the months. Each conditional inserts the month's name when their group participates. They insert nothing when their group does not participate. Since only one of these groups participates in any match, only one of these conditionals actually inserts anything into the replacement.

Then we have a literal space and 4 more conditionals that reference the 4 capturing groups for the days. When the group participates, the conditional uses a backreference to the same group to reinsert the day number matched by the group. The backreference is followed by a literal suffix.

Finally, we have a literal comma, a literal space, and 2 more conditionals for the year. The conditionals again use literal text and a backreference to expand the year from 2 to 4 digits.

8. Finding or Verifying Credit Card Numbers

With a few simple regular expressions, you can easily verify whether your customer entered a valid credit card number on your order form. You can even determine the type of credit card being used. Each card issuer has its own range of card numbers, identified by the first 4 digits.

You can use a slightly different regular expression to find credit card numbers, or number sequences that might be credit card numbers, within larger documents. This can be very useful to prove in a security audit that you're not improperly exposing your clients' financial details.

We'll start with the order form.

Stripping Spaces and Dashes

The first step is to remove all non-digits from the card number entered by the customer. Physical credit cards have spaces within the card number to group the digits, making it easier for humans to read or type in. So your order form should accept card numbers with spaces or dashes in them.

To remove all non-digits from the card number, simply use the “replace all” function in your scripting language to search for the regex `[^0-9]+` and replace it with nothing. If you only want to replace spaces and dashes, you could use `[-]+`. If this regex looks odd, remember that in a character class, the hyphen is a literal when it occurs right before the closing bracket (or right after the opening bracket or negating caret).

If you're wondering what the plus is for: that's for performance. If the input has consecutive non-digits, such as `1===2`, then `[^0-9]+` matches the three equals signs at once and deletes them in one replacement. Without the plus, three replacements would be required. In this case, the savings are only a few microseconds. But it's a good habit to keep regex efficiency in the back of your mind. Though the savings are minimal here, so is the effort of typing the extra plus.

Validating Credit Card Numbers on Your Order Form

Validating credit card numbers is the ideal job for regular expressions. They're just a sequence of 13 to 16 digits, with a few specific digits at the start that identify the card issuer. You can use the specific regular expressions below to alert customers when they try to use a kind of card you don't accept, or to route orders using different cards to different processors. All these regexes were taken from RegexBuddy's library.

- Visa: `^4[0-9]{12}(?:[0-9]{3})?*$` All Visa card numbers start with a 4. New cards have 16 digits. Old cards have 13.
- MasterCard: `^(?:5[1-5][0-9]{2}||222[1-9]||22[3-9][0-9]||2[3-6][0-9]{2}||27[01][0-9]||2720)[0-9]{12}$` MasterCard numbers either start with the numbers 51 through 55 or with the numbers 2221 through 2720. All have 16 digits.
- American Express: `^3[47][0-9]{13}$` American Express card numbers start with 34 or 37 and have 15 digits.
- Diners Club: `^3(?:0[0-5]||[68][0-9])[0-9]{11}$` Diners Club card numbers begin with 300 through 305, 36 or 38. All have 14 digits. There are Diners Club cards that begin with 5 and have 16 digits. These are a joint venture between Diners Club and MasterCard, and should be processed like a MasterCard.

- Discover: `^6(?:011|5[0-9]{2})[0-9]{12}$` Discover card numbers begin with 6011 or 65. All have 16 digits.
- JCB: `^(?:2131|1800|35\d{3})\d{11}$` JCB cards beginning with 2131 or 1800 have 15 digits. JCB cards beginning with 35 have 16 digits.

If you just want to check whether the card number looks valid, without determining the brand, you can combine the above six regexes using alternation. A non-capturing group puts the anchors outside the alternation. Free-spacing allows for comments and for the regex to fit the width of this page.

```
^(?:4[0-9]{12}(?:[0-9]{3})?|
| (?:5[1-5][0-9]{2}
| 222[1-9]|22[3-9][0-9]|2[3-6][0-9]{2}|27[01][0-9]|2720)[0-9]{12}
| 3[47][0-9]{13}
| 3(?:0[0-5]|68)[0-9]{11}
| 6(?:011|5[0-9]{2})[0-9]{12}
| (?:2131|1800|35\d{3})\d{11}
)$
# Visa
# MasterCard
# American Express
# Diners Club
# Discover
# JCB
```

These regular expressions will easily catch numbers that are invalid because the customer entered too many or too few digits. They won't catch numbers with incorrect digits. For that, you need to follow the Luhn algorithm, which cannot be done with a regex. And of course, even if the number is mathematically valid, that doesn't mean a card with this number was issued or that there's money in the account. The benefit of the regular expression is that you can put it in a bit of JavaScript to instantly check for obvious errors, instead of making the customer wait 30 seconds for your credit card processor to fail the order. And if your card processor charges for failed transactions, you'll really want to implement both the regex and the Luhn validation.

Finding Credit Card Numbers in Documents

With two simple modifications, you could use any of the above regexes to find card numbers in larger documents. Simply replace the caret and dollar with a word boundary as in `\b4[0-9]{12}(?:[0-9]{3})?\b`.

If you're planning to search a large document server, a simpler regular expression will speed up the search. Unless your company uses 16-digit numbers for other purposes, you'll have few false positives. The regex `\b\d{13,16}\b` will find any sequence of 13 to 16 digits.

When searching a hard disk full of files, you can't strip out spaces and dashes first like you can when validating a single card number. To find card numbers with spaces or dashes in them, use `\b(?:\d[-]*?)\{13,16}\b`. This regex allows any amount of spaces and dashes anywhere in the number. This is really the only way. Visa and MasterCard put digits in sets of 4, while Amex and Discover use groups of 4, 5 and 6 digits. People entering the numbers may have different ideas yet.

9. Matching Whole Lines of Text

Often, you want to match complete lines in a text file rather than just the part of the line that satisfies a certain requirement. This is useful if you want to delete entire lines in a search-and-replace in a text editor, or collect entire lines in an information retrieval tool.

To keep this example simple, let's say we want to match lines containing the word "John". The regex `John` makes it easy enough to locate those lines. But the software will only indicate `John` as the match, not the entire line containing the word.

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

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

Finding Lines Containing or Not Containing Certain Words

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

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

If your condition is that a line should *not* contain something, use negative lookahead. `^(?!regex).*$` matches a complete line that does *not* match `regex`. 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 `regex` fails everywhere, not just somewhere.

Finally, you can combine multiple positive and negative requirements as follows: `^(?=.*?\bmust-have\b)(?=.*?\bmandatory\b)(?!avoid|illegal).*$`. When checking multiple positive

requirements, the `.*` at the end of the regular expression full of zero-length assertions made sure that we actually matched something. Since the negative requirement must match the entire line, it is easy to replace the `.*` with the negative test.

10. Deleting Duplicate Lines From a File

If you have a file in which all lines are sorted (alphabetically or otherwise), you can easily delete (consecutive) duplicate lines. Simply open the file in your favorite text editor, and do a search-and-replace searching for `^(.*)\r?\n\1+$` and replacing with `\1`. For this to work, the anchors need to match before and after line breaks (and not just at the start and the end of the file or string), and the dot must *not* match newlines.

Here is how this works. The caret will match only at the start of a line. So the regex engine will only attempt to match the remainder of the regex there. The dot and star combination simply matches an entire line, whatever its contents, if any. The parentheses store the matched line into the first backreference.

Next we will match the line separator. I put the question mark into `\r?\n` to make this regex work with both Windows (`\r\n`) and UNIX (`\n`) text files. So up to this point we matched a line and the following line break.

Now we need to check if this combination is followed by a duplicate of that same line. We do this simply with `\1`. This is the first backreference which holds the line we matched. The backreference will match that very same text.

If the backreference fails to match, the regex match and the backreference are discarded, and the regex engine tries again at the start of the next line. If the backreference succeeds, the plus symbol in the regular expression will try to match additional copies of the line. Finally, the dollar symbol forces the regex engine to check if the text matched by the backreference is a complete line. We already know the text matched by the backreference is preceded by a line break (matched by `\r?\n`). Therefore, we now check if it is also followed by a line break or if it is at the end of the file using the dollar sign.

The entire match becomes `line\nline` (or `line\nline\nline` etc.). Because we are doing a search and replace, the line, its duplicates, and the line breaks in between them, are all deleted from the file. Since we want to keep the original line, but not the duplicates, we use `\1` as the replacement text to put the original line back in.

Removing Duplicate Items From a String

We can generalize the above example to `afterseparator(item)(separator\1)+beforeseparator`, where `afterseparator` and `beforeseparator` are zero-length. So if you want to remove consecutive duplicates from a comma-delimited list, you could use `(?<=,|^)([^\,]*)(,\1)+(?=,|$)`.

The positive lookbehind `(?<=,|^)` forces the regex engine to start matching at the start of the string or after a comma. `([^\,]*)` captures the item. `(,\1)+` matches consecutive duplicate items. Finally, the positive lookahead `(?=,|$)` checks if the duplicate items are complete items by checking for a comma or the end of the string.

11. Example Regexes to Match Common Programming Language Constructs

Regular expressions are very useful to manipulate source code in a text editor or in a regex-based text processing tool. Most programming languages use similar constructs like keywords, comments and strings. But often there are subtle differences that make it tricky to use the correct regex. When picking a regex from the list of examples below, be sure to read the description with each regex to make sure you are picking the correct one.

Unless otherwise indicated, all examples below assume that the dot does *not* match newlines and that the caret and dollar *do* match at embedded line breaks. In many programming languages, this means that single-line mode must be off, and multi-line mode must be on.

When used by themselves, these regular expressions may not have the intended result. If a comment appears inside a string, the comment regex will consider the text inside the string as a comment. The string regex will also match strings inside comments. The solution is to use more than one regular expression and to combine those into a simple parser, like in this pseudo-code:

```
GlobalStartPosition := 0;
while GlobalStartPosition < LengthOfText do
  GlobalMatchPosition := LengthOfText;
  MatchedRegex := NULL;
  foreach Regex in RegexList do
    Regex.StartPosition := GlobalStartPosition;
    if Regex.Match and Regex.MatchPosition < GlobalMatchPosition then
      MatchedRegex := Regex;
      GlobalMatchPosition := Regex.MatchPosition;
    endif
  endforeach
  if MatchedRegex <> NULL then
    // At this point, MatchedRegex indicates which regex matched
    // and you can do whatever processing you want depending on
    // which regex actually matched.
  endif
  GlobalStartPosition := GlobalMatchPosition;
endwhile
```

If you put a regex matching a comment and a regex matching a string in `RegexList`, then you can be sure that the comment regex will not match comments inside strings, and vice versa. Inside the loop you can then process the match according to whether it is a comment or a string.

An alternative solution is to combine regexes: `(comment)|(string)`. The alternation has the same effect as the code snipped above. Iterate over all the matches of this regex. Inside the loop, check which capturing group found the regex match. If group 1 matched, you have a comment. If group two matched, you have a string. Then process the match according to that.

You can use this technique to build a full parser. Add regular expressions for all lexical elements in the language or file format you want to parse. Inside the loop, keep track of what was matched so that the following matches can be processed according to their context. For example, if curly braces need to be balanced, increment a counter when an opening brace is matched, and decrement it when a closing brace is matched. Raise an error if the counter goes negative at any point or if it is nonzero when the end of the file is reached.

Comments

`#.*$` matches a single-line comment starting with a `#` and continuing until the end of the line. Similarly, `//.*$` matches a single-line comment starting with `//`.

If the comment must appear at the start of the line, use `^#.*$`. If only whitespace is allowed between the start of the line and the comment, use `^\s*#.*$`. Compiler directives or pragmas in C can be matched this way. Note that in this last example, any leading whitespace will be part of the regex match. Use capturing parentheses to separate the whitespace and the comment.

`/*.*?*/` matches a C-style multi-line comment if you turn on the option for the dot to match newlines. The general syntax is `begin.*?end`. C-style comments do not allow nesting. If the “begin” part appears inside the comment, it is ignored. As soon as the “end” part is found, the comment is closed.

If your programming language allows nested comments, there is no straightforward way to match them using a regular expression, since regular expressions cannot count. Additional logic is required.

Strings

`"[^\r\n]*"` matches a single-line string that does not allow the quote character to appear inside the string. Using the negated character class is more efficient than using a lazy dot. `"[\""]*"` allows the string to span across multiple lines.

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

You can adapt the above regexes to match any sequence delimited by two (possibly different) characters. If we use `b` for the starting character, `e` and the end, and `x` as the escape character, the version without escape becomes `b[^\er\n]*e`, and the version with escape becomes `b[^\ex\r\n]*(?:x. [^\ex\r\n]*)*e`.

Numbers

`\b\d+\b` matches a positive integer number. Do not forget the word boundaries! `[-+]? \b\d+\b` allows for a sign.

`\b0[xX][0-9a-fA-F]+\b` matches a C-style hexadecimal number.

`((\b[0-9]+)?\.)?[0-9]+\b` matches an integer number as well as a floating point number with optional integer part. `(\b[0-9]+\.\. ([0-9]+\b)?|\.\. [0-9]+\b)` matches a floating point number with optional integer as well as optional fractional part, but does not match an integer number.

`((\b[0-9]+)?\.)? \b[0-9]+ ([eE] [-+]? [0-9]+)? \b` matches a number in scientific notation. The mantissa can be an integer or floating point number with optional integer part. The exponent is optional.

`\b[0-9]+(\.[0-9]+)?(e[+-]?[0-9]+)?\b` also matches a number in scientific notation. The difference with the previous example is that if the mantissa is a floating point number, the integer part is mandatory.

If you read through the floating point number example, you will notice that the above regexes are different from what is used there. The above regexes are more stringent. They use word boundaries to exclude numbers that are part of other things like identifiers. You can prepend `[-+]?` to all of the above regexes to include an optional sign in the regex. I did not do so above because in programming languages, the + and - are usually considered operators rather than signs.

Reserved Words or Keywords

Matching reserved words is easy. Simply use alternation to string them together: `\b(first|second|third|etc)\b` Again, do not forget the word boundaries.

12. Find Two Words Near Each Other

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

You can easily perform the same task with the proper regular expression.

Emulating “near” with a Regular Expression

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

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

If the words may also occur in reverse order, we need to specify the opposite pattern as well:

```
\b(?:word1\W+(?:\w+\W+){1,6}?word2|word2\W+(?:\w+\W+){1,6}?word1)\b
```

If you want to find any pair of two words out of a list of words, you can use:

```
\b(word1|word2|word3)(?:\W+\w+){1,6}?\W+(word1|word2|word3)\b
```

The final regex also finds a word near itself. It will match `word2 near word2`, for example.

13. Runaway Regular Expressions: Catastrophic Backtracking

Consider the regular expression `(x+x+)+y`. Before you scream in horror and say this contrived example should be written as `xx+y` or `x{2,}y` to match exactly the same without those terribly nested quantifiers: just assume that each “x” represents something more complex, with certain strings being matched by both “x”. See the section on HTML files below for a real example.

Let’s see what happens when you apply this regex to `xxxxxxxxxy`. The first `x+` will match all 10 `x` characters. The second `x+` fails. The first `x+` then backtracks to 9 matches, and the second one picks up the remaining `x`. The group has now matched once. The group repeats, but fails at the first `x+`. Since one repetition was sufficient, the group matches. `y` matches `y` and an overall match is found. The regex is declared functional, the code is shipped to the customer, and his computer explodes. Almost.

The above regex turns ugly when the `y` is missing from the subject string. When `y` fails, the regex engine backtracks. The group has one iteration it can backtrack into. The second `x+` matched only one `x`, so it can’t backtrack. But the first `x+` can give up one `x`. The second `x+` promptly matches `xx`. The group again has one iteration, fails the next one, and the `y` fails. Backtracking again, the second `x+` now has one backtracking position, reducing itself to match `x`. The group tries a second iteration. The first `x+` matches but the second is stuck at the end of the string. Backtracking again, the first `x+` in the group’s first iteration reduces itself to 7 characters. The second `x+` matches `xxx`. Failing `y`, the second `x+` is reduced to `xx` and then `x`. Now, the group can match a second iteration, with one `x` for each `x+`. But this (7,1),(1,1) combination fails too. So it goes to (6,4) and then (6,2)(1,1) and then (6,1),(2,1) and then (6,1),(1,2) and then I think you start to get the drift.

If you try this regex on a 10x string in RegexBuddy’s debugger, it’ll take 2558 steps to figure out the final `y` is missing. For an 11x string, it needs 5118 steps. For 12, it takes 10238 steps. Clearly we have an exponential complexity of $O(2^n)$ here. At 19x the debugger bows out because it won’t show more than one million steps.

RegexBuddy is forgiving in that it detects it’s going in circles and aborts the match attempt. Other regex engines (like .NET) will keep going forever, while others will crash with a stack overflow (like Perl, before version 5.10). Stack overflows are particularly nasty on Windows, since they tend to make your application vanish without a trace or explanation. Be very careful if you run a web service that allows users to supply their own regular expressions. People with little regex experience have surprising skill at coming up with exponentially complex regular expressions.

Possessive Quantifiers and Atomic Grouping to The Rescue

In the above example, the sane thing to do is obviously to rewrite it as `xx+y` which eliminates the nested quantifiers entirely. Nested quantifiers are repeated or alternated tokens inside a group that is itself repeated or alternated. These almost always lead to catastrophic backtracking. About the only situation where they don’t is when the start of each alternative inside the group is not optional, and mutually exclusive with the start of all the other alternatives, and mutually exclusive with the token that follows it (inside its alternative inside the group). E.g. `(a+b+|c+d+)+y` is safe. If anything fails, the regex engine will backtrack through the whole regex, but it will do so linearly. The reason is that all the tokens are mutually exclusive. None of them can match any characters matched by any of the others. So the match attempt at each backtracking position

will fail, causing the regex engine to backtrack linearly. If you test this on `aaaabbbbccccdddd`, `RegexBuddy` needs only 13 steps rather than millions of steps to figure it out.

However, it's not always possible or easy to rewrite your regex to make everything mutually exclusive. So we need a way to tell the regex engine not to backtrack. When we've grabbed all the x's, there's no need to backtrack. There couldn't possibly be a y in anything matched by either `x+`. Using a possessive quantifier, our regex becomes `(x+x+)+y`. This fails the `21x` string in merely 7 steps. That's 6 steps to match all the x's, and 1 step to figure out that y fails. Almost no backtracking is done. Using an atomic group, the regex becomes `(?>(x+x+)+)y` with the exact same results.

A Real Example: Matching CSV Records

Here's a real example from a technical support case I once handled. The customer was trying to find lines in a comma-delimited text file where the 12th item on a line started with a P. He was using the innocently-looking regex `^(.*?,){11}P`.

At first sight, this regex looks like it should do the job just fine. The lazy dot and comma match a single comma-delimited field, and the `{11}` skips the first 11 fields. Finally, the P checks if the 12th field indeed starts with P. In fact, this is exactly what will happen when the 12th field indeed starts with a P.

The problem rears its ugly head when the 12th field does not start with a P. Let's say the string is `1,2,3,4,5,6,7,8,9,10,11,12,13`. At that point, the regex engine will backtrack. It will backtrack to the point where `^(.*?,){11}` had consumed `1,2,3,4,5,6,7,8,9,10,11`, giving up the last match of the comma. The next token is again the dot. The dot matches a comma. *The dot matches the comma!* However, the comma does not match the 1 in the 12th field, so the dot continues until the 11th iteration of `.*?`, has consumed `11,12,`. You can already see the root of the problem: the part of the regex (the dot) matching the contents of the field also matches the delimiter (the comma). Because of the double repetition (star inside `{11}`), this leads to a catastrophic amount of backtracking.

The regex engine now checks whether the 13th field starts with a P. It does not. Since there is no comma after the 13th field, the regex engine can no longer match the 11th iteration of `.*?`. But it does not give up there. It backtracks to the 10th iteration, expanding the match of the 10th iteration to `10,11,`. Since there is still no P, the 10th iteration is expanded to `10,11,12,`. Reaching the end of the string again, the same story starts with the 9th iteration, subsequently expanding it to `9,10,`, `9,10,11,`, `9,10,11,12,`. But between each expansion, there are more possibilities to be tried. When the 9th iteration consumes `9,10,`, the 10th could match just `11`, as well as `11,12,`. Continuously failing, the engine backtracks to the 8th iteration, again trying all possible combinations for the 9th, 10th, and 11th iterations.

You get the idea: the possible number of combinations that the regex engine will try for each line where the 12th field does not start with a P is huge. All this would take a long time if you ran this regex on a large CSV file where most rows don't have a P at the start of the 12th field.

Preventing Catastrophic Backtracking

The solution is simple. When nesting repetition operators, make absolutely sure that there is only one way to match the same match. If repeating the inner loop 4 times and the outer loop 7 times results in the same overall match as repeating the inner loop 6 times and the outer loop 2 times, you can be sure that the regex engine will try all those combinations.

In our example, the solution is to be more exact about what we want to match. We want to match 11 comma-delimited fields. The fields must not contain comma's. So the regex becomes: `^([\^,\r\n]*,){11}P`. If the P cannot be found, the engine will still backtrack. But it will backtrack only 11 times, and each time the `[\^,\r\n]` is not able to expand beyond the comma, forcing the regex engine to the previous one of the 11 iterations immediately, without trying further options.

See the Difference with RegexBuddy

The screenshot shows the RegexBuddy application window. The main text area contains the regular expression `^([\^,\r\n]*,){11}P`. The History panel on the right shows three entries: "12th field starts with P (lazy dot)", "12th field starts with P (negated class)", and "12th field starts with P (atomic)". The Test panel shows the input string "1,2,3,4,5,6,7,8,9,10,11,12,13" and a message stating "The regular expression does not match the test subject". The Step list on the left shows a sequence of steps from 52121 to 52149, with most steps labeled "backtrack" and the final step labeled "Match attempt failed after 52149 steps".

If you try this example with RegexBuddy's debugger, you will see that the original regex `^([\^,\r\n]*,){11}P` needs 25,593 steps to conclude there regex cannot match `1,2,3,4,5,6,7,8,9,10,11,12`. If the string is `1,2,3,4,5,6,7,8,9,10,11,12,13`, just 3 characters more, the number of steps doubles to 52,149. It's

not too hard to imagine that at this kind of exponential rate, attempting this regex on a large file with long lines could easily take forever.

Our improved regex `^([\^,\r\n]*,){11}P`, however, needs just 52 steps to fail, whether the subject string has 12 numbers, 13 numbers, 16 numbers or a billion. While the complexity of the original regex was exponential, the complexity of the improved regex is constant with respect to whatever follows the 11th field. The reason is the regex fails almost immediately when it discovers the 12th field doesn't start with a P. It backtracks the 11 iterations of the group without expanding again.

The screenshot shows the RegxBuddy application interface. The main window displays the regex `^([\^,\r\n]*,){11}P` and the test subject `1,2,3,4,5,6,7,8,9,10,11,12,13`. The results pane shows that the regex fails on lines 25 through 39, with 'backtrack' labels indicating the failure point. The application interface includes a menu bar, a toolbar, and a main text area.

The complexity of the improved regex is linear to the length of the first 11 fields. 24 steps are needed in our example to match the 11 fields. It takes only 1 step to discover that P can't be matched. Another 27 steps are then needed to backtrack all the iterations of the two quantifiers. That's the best we can do, since the engine

does have to scan through all the characters of the first 11 fields to find out where the 12th one begins. Our improved regex is a perfect solution.

Alternative Solution Using Atomic Grouping

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

The screenshot shows the RegexBuddy application interface. The main window displays the regex `^(?>([^\r\n]+,){11})P` in the input field. The 'Test' tab is active, showing a list of test subjects. The first 25 test subjects are strings of 11 comma-separated numbers (e.g., '1,2,3,4,5,6,7,8,9,10,11'), and the 26th is '1,2,3,4,5,6,7,8,9,10,11,backtrack'. The 27th test subject is 'backtrack'. The application shows that the regex matches the first 25 test subjects but fails on the 26th and 27th. A message at the bottom of the test area states: 'The regular expression does not match the test subject'. The 'History' panel on the right shows three previous test attempts: '12th field starts with P (lazy dot)', '12th field starts with P (negated class)', and '12th field starts with P (atomic)'. The 'Test' panel also shows a message: 'Match attempt failed after 27 steps'.

Using atomic grouping, the above regex becomes `^(?>(. *?,){11})P`. Everything between `(?>)` is treated as one single token by the regex engine, once the regex engine leaves the group. Because the entire group is

one token, no backtracking can take place once the regex engine has found a match for the group. If backtracking is required, the engine has to backtrack to the regex token before the group (the caret in our example). If there is no token before the group, the regex must retry the entire regex at the next position in the string.

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

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

So far, everything happened just like in the original, troublesome regular expression. Now comes the difference. `P` fails to match, so the engine backtracks. But the atomic group made it forget all backtracking positions. The match attempt at the start of the string fails immediately.

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

If you test the final solution in RegxBuddy's debugger, you'll see that it needs the same 24 steps to match the 11 fields. Then it takes 1 step to exit the atomic group and throw away all the backtracking information. Discovering that `P` can't be matched still takes one step. But because of the atomic group, backtracking it all takes only a single step.

Quickly Matching a Complete HTML File

Another common situation where catastrophic backtracking occurs is when trying to match "something" followed by "anything" followed by "another something" followed by "anything", where the lazy dot `. *?` is used. The more "anything", the more backtracking. Sometimes, the lazy dot is simply a symptom of a lazy programmer. `". *?"` is not appropriate to match a double-quoted string, since you don't really want to allow anything between the quotes. A string can't have (unescaped) embedded quotes, so `"[^\r\n]*"` is more appropriate, and won't lead to catastrophic backtracking when combined in a larger regular expression. However, sometimes "anything" really is just that. The problem is that "another something" also qualifies as "anything", giving us a genuine `x+x+` situation.

Suppose you want to use a regular expression to match a complete HTML file, and extract the basic parts from the file. If you know the structure of HTML files, it's very straightforward to write the regular expression

```
<html>. *?<head>. *?<title>. *?</title>. *?</head>. *?<body [^>]*>. *?</body>. *?</html>.
```

With the "dot matches newlines" or "single line" matching mode turned on, it will work just fine on valid HTML files.

Unfortunately, this regular expression won't work nearly as well on an HTML file that misses some of the tags. The worst case is a missing `</html>` tag at the end of the file. When `</html>` fails to match, the regex engine backtracks, giving up the match for `</body>.*?`. It will then further expand the lazy dot before `</body>`, looking for a second closing `</body>` tag in the HTML file. When that fails, the engine gives up `<body[^>]*>.*?`, and starts looking for a second opening `<body[^>]*>` tag all the way to the end of the file. Since that also fails, the engine proceeds looking all the way to the end of the file for a second closing head tag, a second closing title tag, etc.

If you run this regex in RegexBuddy's debugger, the output will look like a sawtooth. The regex matches the whole file, backs up a little, matches the whole file again, backs up some more, backs up yet some more, matches everything again, etc. until each of the 7 `.*` tokens has reached the end of the file. The result is that this regular has a worst case complexity of N^7 . If you double the length of the HTML file with the missing `<html>` tag by appending text at the end, the regular expression will take 128 times (2^7) as long to figure out the HTML file isn't valid. This isn't quite as disastrous as the 2^N complexity of our first example, but will lead to very unacceptable performance on larger invalid files.

In this situation, we know that each of the literal text blocks in our regular expression (the HTML tags, which function as delimiters) will occur only once in a valid HTML file. That makes it very easy to package each of the lazy dots (the delimited content) in an atomic group.

`<html>(?!.*<head>)(?!.*<title>)(?!.*</title>)(?!.*</head>)(?!.*<body[^>]*>)(?!.*</body>).*</html>` will match a valid HTML file in the same number of steps as the original regex. The gain is that it will fail on an invalid HTML file almost as fast as it matches a valid one. When `</html>` fails to match, the regex engine backtracks, giving up the match for the last lazy dot. But then, there's nothing further to backtrack to. Since all of the lazy dots are in an atomic group, the regex engines has discarded their backtracking positions. The groups function as a "do not expand further" roadblock. The regex engine is forced to announce failure immediately.

You've no doubt noticed that each atomic group also contains an HTML tag after the lazy dot. This is critical. We do allow the lazy dot to backtrack until its matching HTML tag was found. E.g. when `.*/body>` is processing `Last paragraph</p></body>`, the `</` regex tokens will match `</` in `</p>`. However, `b` will fail `p`. At that point, the regex engine will backtrack and expand the lazy dot to include `</p>`. Since the regex engine hasn't left the atomic group yet, it is free to backtrack inside the group. Once `</body>` has matched, and the regex engine leaves the atomic group, it discards the lazy dot's backtracking positions. Then it can no longer be expanded.

Essentially, what we've done is to bind a repeated regex token (the lazy dot to match HTML content) to the non-repeated regex token that follows it (the literal HTML tag). Since anything, including HTML tags, can appear between the HTML tags in our regular expression, we cannot use a negated character class instead of the lazy dot to prevent the delimiting HTML tags from being matched as HTML content. But we can and did achieve the same result by combining each lazy dot and the HTML tag following it into an atomic group. As soon as the HTML tag is matched, the lazy dot's match is locked down. This ensures that the lazy dot will never match the HTML tag that should be matched by the literal HTML tag in the regular expression.

14. Runaway Regular Expressions: Too Many Repetitions

When a regular expression contains a repeated group such as `^(one|two)*done$` then it has two alternatives to try for each repetition of the group. In theory this regex should match a string with an arbitrarily long sequence of `oneoneone...done`. In practice a backtracking regex engines will have to give up at some point.

If the regex engine uses a recursive algorithm then each repetition of the group adds a call to the engine's call stack. The engine will give up or even crash when the number of repetitions it actually needs to make exceed the available stack space.

Even if the engine is non-recursive, it still has to keep track of where the group's match attempt started with each repetition. This is needed so the engine can backtrack if the remainder of the regex fails to match. When trying to match `oneoneone` the engine repeats the group 3 times. It stores a backtracking position for the quantifier before each repetition. The alternation operator also stores a backtracking position at each attempt. After the 3 repetitions `done` fails to match at the end of the string. Now the engine goes back through the 6 backtracking positions in reverse order. It attempts `two` at each backtracking position of the alternation operator. It attempts `done` at each backtracking position of the quantifier. The process is entirely linear. Even with a string that repeats `one` thousands of times the regex will match or fail to match instantly, depending on whether there's a `done` at the end of the string.

But if you use this regex on a string that repeats `one` millions of times then you may run into limitations of the regex engine designed to stop it from running out of memory trying to remember all those backtracking positions. This can prevent the engine from finding extremely long matches.

The above example makes matters worse by using a capturing group. At the end of a successful match, the capturing group will hold the last occurrence of `one` or `two` in the string. But during the match attempt, it also holds the most recent match of `one` or `two` with each iteration. This causes extra work for the regex engine with each iteration of the group and each time the group is backtracked into.

Optimize with Non-Capturing and Atomic Groups

We can optimize this regex to reduce the amount of work the regex engine has to do. These are good techniques to apply to all your regexes. Even if you'll only use your regexes on shorter strings where the performance difference is hardly measurable, you should treat them as good coding habits.

First of all, only use capturing groups if you really want to capture part of the regex match. Otherwise you can always use a non-capturing group. Turning a capturing group that doesn't have a backreference into a non-capturing group never changes what the regex is supposed to match. So `^(?:one|two)*done$` is our first optimization.

In this case, the two alternatives are mutually exclusive. `two` can never match at a position where `one` has already matched. So all that backtracking is unnecessary. We can tell the regex engine that by using an atomic group: `^(?>one|two)*done$`. Now the regex engine throws away the backtracking position of the alternation operator each time it repeats the group. It no longer attempts `two` at a position where `one` has already matched.

Optimize with Possessive Quantifiers

But the quantifier `*` still backtracks. `^(?>one|two)*done$` still attempts `done` at every position where `one` has matched as the quantifier backtracks. To stop this we can make the quantifier possessive: `^(?>one|two)*+done$`. This regex does not backtrack at all.

Whether this regex can really match an arbitrarily long sequence of `oneoneone...done` depends on how the regex engine implements possessive quantifiers. If the possessive quantifier does not store backtracking positions at all, then it can. But in some regex flavors the possessive quantifier is another way of writing `^(?>(?:one|two)*)done$`. In that case, the quantifier still stores all its backtracking positions, only to throw them away when the regex engine exits the outer atomic group. This does improve performance when `done` fails to match. But it doesn't allow longer successful matches if the regex engine is limited by the number of backtracking positions that the quantifier can store.

Eliminate Needless Groups

Even better than turning capturing groups into non-capturing or atomic groups is to eliminate unnecessary groups. People sometimes needlessly group regex tokens because they do not understand the precedence of operators in a regex. The alternation operator has the lowest precedence of all. It alternates between everything to the left of it and everything to the right of it within the regex or group that contains it. A quantifier has high precedence. It repeats just the token or group in front of it.

So `one|two*` would match `one`, `tw`, `two`, `twoo`, `twooo`, etc. We needed the group to repeat the alternation instead of just the final `o`. But we don't need extra groups around the alternatives. The two nested groups in `(?: (?:one|two))*` are unnecessary.

`(?:[ab])+` also has a needless group. The character class is treated as a single regex token. The quantifier can repeat it directly: `[ab]+`.

How much an impact these unnecessary groups have depends on the regex engine. If you followed the advice to use non-capturing groups then the engine may be able to optimize away the unnecessary groups. But it can't do that if you have unnecessary capturing groups. The regex engine can't know whether you'll need to retrieve the text matched by the capturing groups afterwards, so it can't remove them.

Repeat Single Tokens

In theory, `^(?:a|b|c)+$` and `^[abc]+$` are the same. In practice, most backtracking engines execute the latter much faster. The character class can attempt both characters at the same time. It doesn't need to backtrack at all to try the other characters like the alternation operator does. Each iteration of the character class matches exactly one character. While the quantifier may need to backtrack, it doesn't need backtracking positions to do so. It just needs to remember the number of iterations. It can backtrack simply by stepping backwards one character in the string and decrementing the number of iterations. This enables `^[abc]+$` to match a string of any length.

`"(?:[^\"]|\\.)*"` is a simplistic solution to match a double-quoted string that may contain double quotes if they are escaped by a backslash. We allow line breaks and assume the dot matches them.

The above regex is correct in the sense that it matches all double-quoted strings and nothing else. But it is simplistic because it performs poorly. At least the it uses a non-capturing group. We could use an atomic group if we flipped the two alternatives (remember the regex engine is eager). But unless the regex engine has possessive quantifiers that don't store backtracking positions at all, we're not going to be able make this regex match double-quoted strings that are millions of characters long as long as we're repeating the group for each character in the string.

To optimize this regex we need to repeat the negated character class: `"(?:[^\\"]+|\\".)*"`. This allows the regex to quickly match runs of non-escaped characters within the string. Since those are far more common than escaped characters, this significantly reduces the number of backtracking positions the regex engine needs to remember. The outer group only repeats once for each run of non-escaped characters and once for each escaped character. The two quantifiers will still backtrack if the closing quote fails to match. But most iterations will be of the inner quantifier which can backtrack much faster.

Note that the negated character class now includes the backslash. This ensures the two alternatives are mutually exclusive. This is essential. If you paid attention to the catastrophic backtracking topic then you'll notice a similar pattern of nested quantifiers. Though our regex will backtrack when the closing quote fails to match, it does so linearly because the second alternative can never match a character that was matched by the first alternative.

We can take this optimization one step further. We don't need to repeat the group for runs of non-escaped characters and we don't need it to alternate it between escaped and non-escaped characters. We only need the group to handle escaped characters. `"[^\\"]*(?:\\.[^\\"])*"` treats a double-quoted string as a series of zero or more non-escaped characters followed by zero or more escaped characters that are each followed by zero or more non-escaped characters. Now the group only remembers one backtracking position for each non-escaped character in the string. This enables the regex to match strings of pretty much any length. It'll only run into regex engine limitations if a string should contain millions of escaped characters. It will backtrack if the closing quote fails to match. But all the backtracking attempts will immediately fail because the group starts with `\\.` which is mutually exclusive with `[^\\"]*`.

If the regex engine does support atomic grouping or possessive quantifiers then we can put the icing on the cake with `"(?:>[^\\"]*(?:>\\.[^\\"]*+))"` or `"[^\\"]*+(?:\\.[^\\"]*+)*"`. Both these regexes throw away all backtracking positions when attempting to match the closing double quote. So they never backtrack at all.

15. Preventing Regular Expression Denial of Service (ReDoS)

The previous topic explains catastrophic backtracking with practical examples from the perspective of somebody trying to get their regular expressions to work and perform well on their own PC. You should understand those examples before reading this topic.

It's annoying when catastrophic backtracking happens on your PC. But when it happens in a server application with multiple concurrent users, it can really be catastrophic. Too many users running regexes that exhibit catastrophic backtracking will bring down the whole server. And "too many" need only be as few as the number of CPU cores in the server.

If the server accepts regexes from the user, then the user can easily provide one that exhibits catastrophic backtracking on any subject. If the server accepts subject data from the user, then the user may be able to provide subjects that trigger catastrophic backtracking in regexes used by the server, if those regexes are predisposed to catastrophic backtracking. When the user can do either of those things, the server is susceptible to regular expression denial of service (ReDoS). When enough users (or one actor masquerading as many users) provide malicious regexes and/or subjects to match against, the server will be spending nearly all its CPU cycles on trying to match those regexes.

Handling Regexes Provided by The User

If your application allows the user to provide their own regexes, then your only real defense is to use a text-directed regex engine. Those engines don't backtrack. Their performance depends on the length of the subject string, not the complexity of the regular expression. But they also don't support features like backreferences that depend on backtracking and that many users expect.

If your application uses a backtracking engine with user-provided regexes, then you can only mitigate the consequences of catastrophic backtracking. And you'll really need to do so. It's very easy for people with limited regex skills to accidentally craft one that degenerates into catastrophic backtracking.

You'll need to use a regex engine that aborts the match attempt when catastrophic backtracking occurs rather than running until the script crashes or the OS kills it. You can easily test this. When the regex `(x\w{1,10})+y` is attempted on an ever growing string of `x`'s there should be a reasonable limit on how long it takes for the regex engine to give up. Ideally your engine will allow you to configure this limit for your purposes. The .NET engine, for example, allows you to pass a timeout to the `Regex()` constructor. The PCRE engine allows you to set recursion limits. The lower your limits the better the protection against ReDoS, but higher the risk of aborting legitimate regexes that would find a valid match given slightly more time. Low recursion limits may prevent long regex matches. Low timeouts may abort searches through large files too early.

If your regex engine has no such features, you could implement your own timeout. Spawn a separate thread to execute the regular expression. Wait on the thread with a timeout. If the thread finishes before the wait times out, process its result. Otherwise, kill the thread and tell the user the regex is too complex. The `safe_regexp` package implements this for Ruby.

Reviewing Regexes in The Application

If the server only uses regexes that are hard-coded in your application, then you can prevent regex-based denial of service attacks entirely. You need to make sure that your regexes won't exhibit catastrophic backtracking regardless of the subjects they're used on. This isn't particularly difficult for somebody with a solid grasp of regular expressions. But it does require care and attention. It's not enough to just test that the regex matches valid subjects. You need to make sure, by looking at the regex independently of any subject data, that it is not possible for multiple permutations of the same regex to match the same thing.

Permutations occur when you give the regular expression a choice. You can do this with alternation and with quantifiers. So these are the regex tokens you need to inspect. Possessive quantifiers are excepted, because they never backtrack.

Alternation

Alternatives must be mutually exclusive. If multiple alternatives can match the same text then the engine will try both if the remainder of the regex fails. If the alternatives are in a group that is repeated, you have catastrophic backtracking.

A classic example is `(.|\\s)*` to match any amount of any text when the regex flavor does not have a “dot matches line breaks” mode. If this is part of a longer regex then a subject string with a sufficiently long run of spaces will break the regex. The engine will try every possible combination of the spaces being matched by `.` or `\\s`. For example, 3 spaces could be matched as `...`, `..\\s`, `.\\s.`, `.\\s\\s`, `\\s.`, `\\s.\\s`, `\\s\\s.`, or `\\s\\s\\s`. That's 2^N permutations. The fix is to use `(.|\\n)*` to make the alternatives mutually exclusive. Even better to be more specific about which characters are really allowed, such as `[\\r\\n\\t\\x20-\\x7E]*` for ASCII printables, tabs, and line breaks.

It is acceptable for two alternatives to partially match the same text. `[0-9]*\\. [0-9]+| [0-9]+` is perfectly fine to match a floating point number with optional integer part and optional fraction. Though a subject that consists of only digits is initially matched by `[0-9]*` and does cause some backtracking when `\\.` fails, this backtracking never becomes catastrophic. Even if you put this inside a group in a longer regex, the group only does a minimal amount of backtracking. (But the group mustn't have a quantifier or it will fall foul of the rule for nested quantifiers.)

Quantifiers in Sequence

Quantified tokens that are in sequence must either be mutually exclusive with each other or be mutually exclusive with what comes between them. Otherwise both can match the same text and all combinations of the two quantifiers will be tried when the remainder of the regex fails to match. A token inside a group with alternation is still in sequence with any token before or after the group.

A classic example is `a.*?b.*?c` to match 3 things with “anything” between them. When `c` can't be matched the first `.*` expands character by character until the end of the line or file. For each expansion the second `.*` expands character by character to match the remainder of the line or file. The fix is to realize that you can't have “anything” between them. The first run needs to stop at `b` and the second run needs to stop at `c`. With single characters `a[^b]*b[^c]*c` is an easy solution. The negated character classes guarantee the

repetition stops at the delimiter. If your regex flavor supports possessive quantifiers then you can use `a[b]*+b[c]*+c` to further increase performance.

For a more complex example and solution, see matching a complete HTML file in the previous topic. This explains how you can use atomic grouping to prevent backtracking in more complex situations.

Nested Quantifiers

A group that contains a token with a quantifier must not have a quantifier of its own unless the quantified token inside the group can only be matched with something else that is mutually exclusive with it. That ensures that there is no way that fewer iterations of the outer quantifier with more iterations of the inner quantifier can match the same text as more iterations of the outer quantifier with fewer iterations of the inner quantifier.

The regex `(x\w{1,10})+y` matches a sequence of one or more codes that start with an `x` followed by 1 to 10 word characters, all followed by a `y`. All is well as long as the `y` can be matched. When the `y` is missing, backtracking occurs. If the string doesn't have too many `x`'s then backtracking happens very quickly. Things only turn catastrophic when the subject contains a long sequence of `x`'s. `x` and `x` are not mutually exclusive. So the repeated group can match `xxxx` in one iteration as `x\w\w\w` or in two iterations as `x\wx\w`.

To solve this, you first need to consider whether `x` and `y` should be allowed in the 1 to 10 characters that follow it. Excluding the `x` eliminates most backtracking. What's left won't be catastrophic. You could exclude it with character class subtraction as in `(x[\w-[x]]{1,10})+y` or with character class intersection as in `(x[\w&&[^x]]{1,10})+y`. If you don't have those features you'll need to spell out the characters you want to allow: `(x[a-wyz0-9_]{1,10})+y`.

If the `x` should be allowed then your only solution is to disallow the `y` in the same way. Then you can make the group atomic or the quantifier possessive to eliminate the backtracking.

If both `x` and `y` should be allowed in the sequences of 1 to 10 characters, then there is no regex-only solution. You can't make the group atomic or the quantifier possessive as then `\w{1,10}` matches the final `y` which causes `y` to fail.

Other Defensive Techniques

In addition to preventing catastrophic backtracking as explained above, you should make your regular expressions as strict as possible. The stricter the regex, the less backtracking it does and thus the better it performs. Even if you can't measure the performance difference because the regex is used infrequently on short strings, proper technique is a habit. It also reduces the chance that a less experienced developer introduces catastrophic backtracking when they extend your regex later.

Make groups that contain alternatives atomic as much as you can. Use `\b(?:one|two|three)\b` to match a list of words.

Make quantifiers possessive as much as you can. If a repeated token is mutually exclusive with what follows, enforce that with a possessive quantifier.

Use (negated) character classes instead of the dot. It's rare that you really want to allow "anything". A double-quoted string, for example, can't contain "anything". It can't contain unescaped double quotes. So use `"[^\n]"*+` instead of `".*?"`. Though both find exactly the same matches when used on their own, the latter can lead to catastrophic backtracking when pasted into a longer regex. The former never backtracks regardless of anything else the regex needs to match.

Why Use Regexes at All?

Some would certainly argue that the above only shows that regexes are dangerous and that they should not be used. They'll then force developers to do the job with procedural code. Procedural code to match non-trivial patterns quickly becomes long and complicated, increasing the chance of bugs and the cost to develop and maintain the code. Many pattern matching problems are naturally solved with recursion. And when a large subject string can't be matched, runaway recursion leads to stack overflows that crash the application.

Developers need to learn to correctly use their tools. This is no different for regular expressions than for anything else.

16. Repeating a Capturing Group vs. Capturing a Repeated Group

When creating a regular expression that needs a capturing group to grab part of the text matched, a common mistake is to repeat the capturing group instead of capturing a repeated group. The difference is that the repeated capturing group will capture only the last iteration, while a group capturing another group that's repeated will capture all iterations. An example will make this clear.

Let's say you want to match a tag like `!abc!` or `!123!`. Only these two are possible, and you want to capture the `abc` or `123` to figure out which tag you got. That's easy enough: `!(abc|123)!` will do the trick.

Now let's say that the tag can contain multiple sequences of `abc` and `123`, like `!abc123!` or `!123abcabc!`. The quick and easy solution is `!(abc|123)+!`. This regular expression will indeed match these tags. However, it no longer meets our requirement to capture the tag's label into the capturing group. When this regex matches `!abc123!`, the capturing group stores only `123`. When it matches `!123abcabc!`, it only stores `abc`.

This is easy to understand if we look at how the regex engine applies `!(abc|123)+!` to `!abc123!`. First, `!` matches `!`. The engine then enters the capturing group. It makes note that capturing group #1 was entered when the engine reached the position between the first and second character in the subject string. The first token in the group is `abc`, which matches `abc`. A match is found, so the second alternative isn't tried. (The engine does store a backtracking position, but this won't be used in this example.) The engine now leaves the capturing group. It makes note that capturing group #1 was exited when the engine reached the position between the 4th and 5th characters in the string.

After having exited from the group, the engine notices the plus. The plus is greedy, so the group is tried again. The engine enters the group again, and takes note that capturing group #1 was entered between the 4th and 5th characters in the string. It also makes note that since the plus is not possessive, it may be backtracked. That is, if the group cannot be matched a second time, that's fine. In this backtracking note, the regex engine also saves the entrance and exit positions of the group during the previous iteration of the group.

`abc` fails to match `123`, but `123` succeeds. The group is exited again. The exit position between characters 7 and 8 is stored.

The plus allows for another iteration, so the engine tries again. Backtracking info is stored, and the new entrance position for the group is saved. But now, both `abc` and `123` fail to match `!`. The group fails, and the engine backtracks. While backtracking, the engine restores the capturing positions for the group. Namely, the group was entered between characters 4 and 5, and existed between characters 7 and 8.

The engine proceeds with `!`, which matches `!`. An overall match is found. The overall match spans the whole subject string. The capturing group spans characters 5, 6 and 7, or `123`. Backtracking information is discarded when a match is found, so there's no way to tell after the fact that the group had a previous iteration that matched `abc`. (The only exception to this is the .NET regex engine, which does preserve backtracking information for capturing groups after the match attempt.)

The solution to capturing `abc123` in this example should be obvious now: the regex engine should enter and leave the group only once. This means that the plus should be inside the capturing group rather than outside. Since we do need to group the two alternatives, we'll need to place a second capturing group around the repeated group: `!((abc|123)+)!`. When this regex matches `!abc123!`, capturing group #1 will store

abc123, and group #2 will store 123. Since we're not interested in the inner group's match, we can optimize this regular expression by making the inner group non-capturing: `!(?:abc|123)+!`.

17. Mixing Unicode and 8-bit Character Codes

Internally, computers deal with numbers, not with characters. When you save a text file, each character is mapped to a number, and the numbers are stored on disk. When you open a text file, the numbers are read and mapped back to characters. When processing text with a regular expression, the regular expression needs to use the same mapping as you used to create the file or string you want the regex to process.

When you simply type in all the characters in your regular expression, you normally don't have anything to worry about. The application or programming library that provides the regular expression functionality will know what text encodings your subject string uses, and process it accordingly. So if you want to search for the euro currency symbol, and you have a European keyboard, just press AltGr+E. Your regex € will find all euro symbols just fine.

But you can't press AltGr+E on a US keyboard. Or perhaps you like your source code to be 7-bit clean (i.e. plain ASCII). In those cases, you'll need to use a character escape in your regular expression.

If your regular expression engine supports Unicode, simply use the Unicode escape `\u20AC` (most Unicode flavors) or `\x{20AC}` (Perl and PCRE). U+20AC is the Unicode code point for the euro symbol. It will always match the euro symbol, whether your subject string is encoded in UTF-8, UTF-16, UCS-2 or whatever. Even when your subject string is encoded with a legacy 8-bit code page, there's no confusion. You may need to tell the application or regex engine what encoding your file uses. But `\u20AC` is always the euro symbol.

Most Unicode regex engines also support the 8-bit character escape `\xFF`. However, its use is not recommended. For characters `\x00` through `\x7F`, there's usually no trouble. The first 128 Unicode code points are identical to the ASCII table that most 8-bit code pages are based on.

But the interpretation of `\x80` and above may vary. A pure Unicode engine will treat this identical to `\u0080`, which represents a Latin-1 control code. But what most people expect is that `\x80` matches the euro symbol, as that occupies position 80h in all Windows code pages. And it will when using an 8-bit regex engine if your text file is encoded using a Windows code page.

Since most people expect `\x80` to be treated as an 8-bit character rather than the Unicode code point `\u0080`, some Unicode regex engines do exactly that. Some are hard-wired to use a particular code page, say Windows 1252 or your computer's default code page, to interpret 8-bit character codes.

Other engines will let it depend on the input string. Just Great Software applications treat `\x80` as `\u0080` when searching through a Unicode text file, but as `\u20AC` when searching through a Windows 1252 text file. There's no magic here. It matches the character with index 80h in the text file, regardless of the text file's encoding. Unicode code point U+0080 is a Latin-1 control code, while Windows 1252 character index 80h is the euro symbol. In reverse, if you type in the euro symbol in a text editor, saving it as UTF-16LE will save two bytes AC 20, while saving as Windows 1252 will give you one byte 80.

If you find the above confusing, simply don't use `\x80` through `\xFF` with a regex engine that supports Unicode.

8-bit Regex Engines

When working with a legacy (obsolete?) regular expression engine that works on 8-bit data only, you can't use Unicode escapes like `\u20AC`. `\x80` is all you have. Note that even modern engines have legacy modes. The popular regex library PCRE, for example, runs as an 8-bit engine by default. You need to explicitly enable UTF-8 support if you want to use Unicode features. When you do, PCRE also expects you to convert your subject strings to UTF-8.

When crafting a regular expression for an 8-bit engine, you'll have to take into account which character set or code page you'll be working with. 8-bit regex engines just don't care. If you type `\x80` into your regex, it will match any byte 80h, regardless of what that byte represents. That'll be the euro symbol in a Windows 1252 text file, a control code in a Latin-1 file, and the digit zero in an EBCDIC file.

Even for literal characters in your regex, you'll have to match up the encoding you're using in the regular expression with the subject encoding. If your application is using the Latin-1 code page, and you use the regex `Ä`, it'll match `Ë` when you search through a Latin-2 text file. The application would duly display this as `Ä` on the screen, because it's using the wrong code page. This problem is not really specific to regular expressions. You'll encounter it any time you're working with files and applications that use different 8-bit encodings.

So when working with 8-bit data, open the actual data you're working with in a hex editor. See the bytes being used, and specify those in your regular expression.

Where it gets really hairy is if you're processing Unicode files with an 8-bit engine. Let's go back to our text file with just a euro symbol. When saved as little endian UTF-16 (called "Unicode" on Windows), an 8-bit regex engine will see two bytes `AC 20` (remember that little endian reverses the bytes). When saved as UTF-8 (which has no endianness), our 8-bit engine will see three bytes `E2 82 AC`. You'd need `\xE2\x82\xAC` to match the euro symbol in an UTF-8 file with an 8-bit regex engine.

Part 4

Regular Expressions Tools & Programming Languages

1. Specialized Tools and Utilities for Working with Regular Expressions

These tools and utilities have regular expressions as the core of their functionality.

grep - The utility from the UNIX world that first made regular expressions popular

PowerGREP - Next generation grep for Microsoft Windows

RegexBuddy - Learn, create, understand, test, use and save regular expressions. RegexBuddy makes working with regular expressions easier than ever before.

RegexMagic - Generate regular expressions using RegexMagic's powerful patterns instead of the cryptic regular expression syntax.

General Applications with Notable Support for Regular Expressions

There are a lot of applications these days that support regular expressions in one way or another, enhancing certain part of their functionality. But certain applications stand out from the crowd by implementing a full-featured Perl-style regular expression flavor and allowing regular expressions to be used instead of literal search terms throughout the application.

EditPad Lite - Basic text editor that has all the essential features for text editing, including powerful regex-based search and replace.

EditPad Pro - Convenient text editor with a powerful regex-based search and replace feature, as well as regex-based customizable syntax coloring and file navigation.

Programming Languages and Libraries

If you are a programmer, you can save a lot of coding time by using regular expressions. With a regular expression, you can do powerful string parsing in only a handful lines of code, or maybe even just a single line. A regex is faster to write and easier to debug and maintain than dozens or hundreds of lines of code to achieve the same by hand.

Boost - Free C++ source libraries with comprehensive regex support that was later standardized by C++11. But there are significant differences in Boost's regex flavors and the flavors in std::regex implementations.

Delphi - Delphi XE and later ship with RegularExpressions and RegularExpressionsCore units that wrap the PCRE library. For older Delphi versions, you can use the TPerlRegEx component, which is the unit that the RegularExpressionsCore unit is based on.

Gnulib - Gnulib or the GNU Portability Library includes many modules, including a regex module. It implements both POSIX flavors, as well as these two flavors with added GNU extensions.

Groovy - Groovy uses Java's `java.util.regex` package for regular expressions support. Groovy adds only a few language enhancements that allow you to instantiate the `Pattern` and `Matcher` classes with far fewer keystrokes.

Java - Java 4 and later include an excellent regular expressions library in the `java.util.regex` package.

JavaScript - If you use JavaScript to validate user input on a web page at the client side, using JavaScript's built-in regular expression support will greatly reduce the amount of code you need to write.

.NET (dot net) - Any .NET-based programming language such as C# (C sharp) or VB.NET can use .NET's excellent support for regular expressions.

PCRE - Popular open source regular expression library written in ANSI C that you can link directly into your C and C++ applications, or use through an `.so` (UNIX/Linux) or a `.dll` (Windows).

Perl - The text-processing language that gave regular expressions a second life, and introduced many new features. Regular expressions are an essential part of Perl.

PHP - Popular language for creating dynamic web pages, with three sets of regex functions. Two implement POSIX ERE, while the third is based on PCRE.

POSIX - The POSIX standard defines two regular expression flavors that are implemented in many applications, programming languages and systems.

PowerShell - PowerShell is a programming language from Microsoft that is primarily designed for system administration. Since PowerShell is built on top of .NET, it's built-in regex operators `-match` and `-replace` use the .NET regex flavor. PowerShell can also access the .NET Regex classes directly.

Python - Popular high-level scripting language with a comprehensive built-in regular expression library

R - The R Language is the programming languages used in the R Project for statistical computing. It has built-in support for regular expressions based on POSIX and PCRE.

Ruby - Another popular high-level scripting language with comprehensive regular expression support as a language feature.

`std::regex` - Regex support part of the standard C++ library defined in C++11 and previously in TR1.

Tcl - Tcl, a popular "glue" language, offers three regex flavors. Two POSIX-compatible flavors, and an "advanced" Perl-style flavor.

VBScript - Microsoft scripting language used in ASP (Active Server Pages) and Windows scripting, with a built-in `RegExp` object implementing the regex flavor defined in the JavaScript standard.

Visual Basic 6 - Last version of Visual Basic for Win32 development. You can use the VBScript `RegExp` object in your VB6 applications.

wxWidgets - Popular open source windowing toolkit. The `wxRegEx` class encapsulates the "Advanced Regular Expression" engine originally developed for Tcl.

XML Schema - The W3C XML Schema standard defines its own regular expression flavor for validating simple types using pattern facets.

Xojo - Cross-platform development tool formerly known as REALbasic, with a built-in RegEx class based on PCRE.

XQuery and XPath - The W3C standard for XQuery 1.0 and XPath 2.0 Functions and Operators extends the XML Schema regex flavor to make it suitable for full text search.

XRegExp - Open source JavaScript library that enhances the regex syntax and eliminates many cross-browser inconsistencies and bugs.

Databases

Modern databases often offer built-in regular expression features that can be used in SQL statements to filter columns using a regular expression. With some databases you can also use regular expressions to extract the useful part of a column, or to modify columns using a search-and-replace.

MySQL - MySQL's REGEXP operator works just like the LIKE operator, except that it uses a POSIX Extended Regular Expression.

Oracle - Oracle Database 10g adds 4 regular expression functions that can be used in SQL and PL/SQL statements to filter rows and to extract and replace regex matches. Oracle implements POSIX Extended Regular Expressions.

PostgreSQL - PostgreSQL provides matching operators and extraction and substitution functions using the "Advanced Regular Expression" engine also used by Tcl.

2. C++ Regular Expressions with Boost

Boost is a free source code library for C++. After downloading and unzipping, you need to run the `bootstrap` batch file or script and then run `b2 --with-regex` to compile Boost's regex library. Then add the folder into which you unzipped Boost to the include path of your C++ compiler. Add the `stage\lib` subfolder of that folder to your linker's library path. Then you can add `#include <boost/regex.hpp>` to your C++ code to make use of Boost regular expressions.

If you use C++Builder, you should download the Boost libraries for your specific version of C++Builder from Embarcadero. The version of Boost you get depends on your version of C++Builder and whether you're targeting Win32 or Win64. The Win32 compiler in XE3 through XE8, and the classic Win32 compiler in C++Builder 10 Seattle through 10.1 Berlin are all stuck on Boost 1.39. The Win64 compiler in XE3 through XE6 uses Boost 1.50. The Win64 compiler in XE7 through 10.1 Berlin uses Boost 1.55. The new C++11 Win32 compiler in C++Builder 10 and later uses the same version of boost as the Win64 compiler.

This book covers Boost 1.38, 1.39, and 1.42 through the latest 1.73. Boost 1.40 introduced many new regex features borrowed from Perl 5.10. But it also introduced some serious bugs that weren't fixed until Boost 1.42. So we completely ignore Boost 1.40 and 1.41. We still cover Boost 1.38 and 1.39 (which have identical regex features) because the classic Win32 C++Builder compiler is stuck on this version. If you're using another compiler, you should definitely use Boost 1.42 or later to avoid what are now old bugs. You should preferably use Boost 1.47 or later as this version changes certain behaviors involving backreferences that may change how some of your regexes behave if you later upgrade from pre-1.47 to post-1.47.

In practice, you'll mostly use the Boost's ECMAScript grammar. It's the default grammar and offers far more features than the other grammars. Whenever the tutorial in this book mentions Boost without mentioning any grammars then what is written applies to the ECMAScript grammar and may or may not apply to any of the other grammars. You'll really only use the other grammars if you want to reuse existing regular expressions from old POSIX code or UNIX scripts.

Boost And Regex Standards

The Boost documentation likes to talk about being compatible with Perl and JavaScript and how `boost::regex` was standardized as `std::regex` in C++11. When we compare the Dinkumware implementation of `std::regex` (included with Visual Studio and C++Builder) with `boost::regex`, we find that the class and function templates are almost the same. Your C++ compiler will just as happily compile code using `boost::regex` as it does compiling the same code using `std::regex`. So all the code examples given in the `std::regex` topic in this book work just fine with Boost if you replace `std` with `boost`.

But when you run your C++ application then it can make a big difference whether it is Dinkumware or Boost that is interpreting your regular expressions. Though both offer the same six grammars, their syntax and behavior are not the same between the two libraries. Boost defines `regex_constants::perl` which is not part of the C++11 standard. This is not actually an additional grammar but simply a synonym to ECMAScript and JavaScript. There are major differences in the regex flavors used by actual JavaScript and actual Perl. So it's obvious that a library treating these as one flavor or grammar can't be compatible with either. Boost's ECMAScript grammar is a cross between the actual JavaScript and Perl flavors, with a bunch of Boost-specific features and peculiarities thrown in. Dinkumware's ECMAScript grammar is closer to actual JavaScript, but still has significant behavioral differences. Dinkumware didn't borrow any features from Perl that JavaScript doesn't have.

The table below highlights the most important differences between the ECMAScript grammars in `std::regex` and Boost and actual JavaScript and Perl. Some are obvious differences in feature sets. But others are subtle differences in behavior that may bite you unexpectedly.

Feature	std::regex	Boost	JavaScript	Perl
Dot matches line breaks	never	(default)	never	(option)
Anchors match at line breaks	always	(default)	(option)	(option)
Line break characters	CR, LF	CR, LF, FF, NEL, LS, PS	CR, LF, LS, PS	LF
Backreferences to participating groups	non-Match empty string	empty (fail since 1.47)	Match empty string	empty (fail)
Empty character class	Fails to match	(Not possible)	Fails to match	(Not possible)
Free-spacing mode	no	(YES)	no	(YES)
Mode modifiers	no	(YES)	no	(YES)
Possessive quantifiers	no	(YES)	no	(YES)
Named capture	no	(.NET syntax)	no	(.NET & Python syntax)
Recursion	no	(atomic)	no	(backtracking)
Subroutines	no	(backtracking)	no	(backtracking)
Conditionals	no	(YES)	no	(YES)
Atomic groups	no	(YES)	no	(YES)
Atomic groups backtrack capturing groups	n/a	(no)	n/a	(YES)
Start and end of word boundaries	no	(YES)	no	no
Standard POSIX classes	(YES)	(YES)	no	(YES)
Single letter POSIX classes	no	(YES)	no	no
Feature	std::regex	Boost	JavaScript	Perl

3. Delphi Regular Expressions Classes

Delphi XE is the first release of Delphi that has built-in support for regular expressions. In most cases you'll use the `RegularExpressions` unit. This unit defines a set of records that mimic the regular expression classes in the .NET framework. Just like in .NET, they allow you to use a regular expression in just one line of code without explicit memory management.

Internally the `RegularExpressions` unit uses the `RegularExpressionsCore` unit which defines the `TPerlRegEx` class. `TPerlRegEx` is a wrapper around the open source PCRE library. Thus both the `RegularExpressions` and `RegularExpressionsCore` units use the PCRE regex flavor.

Delphi's `RegularExpressions` unit

The `RegularExpressions` unit defines `TRegEx`, `TMatch`, `TMatchCollection`, `TGroup`, and `TGroupCollection` as records rather than as classes. That means you don't need to call `Create` and `Free` to allocate and deallocate memory.

`TRegEx` does have a `Create` constructor that you can call if you want to use the same regular expression more than once. That way `TRegEx` doesn't compile the same regex twice. If you call the constructor, you can then call any of the non-static methods that do not take the regular expression as a parameter. If you don't call the constructor, you can only call the static (class) methods that take the regular expression as a parameter. All `TRegEx` methods have static and non-static overloads. Which ones you use solely depends on whether you want to make more than one call to `TRegEx` using the same regular expression.

The `IsMatch` method takes a string and returns `True` or `False` indicating whether the regular expression matches (part of) the string.

The `Match` method takes a string and returns a `TMatch` record with the details of the first match. If the match fails, it returns a `TMatch` record with the `Success` property set to `nil`. The non-static overload of `Match()` takes an optional starting position and an optional length parameter that you can use to search through only part of the input string.

The `Matches` method takes a string and returns a `TMatchCollection` record. The default `Item[]` property of this record holds a `TMatch` for each match the regular expression found in the string. If there are no matches, the `Count` property of the returned `TMatchCollection` record is zero.

Use the `Replace` method to search-and-replace all matches in a string. You can pass the replacement text as a string using the JGsoft replacement text flavor. Or, you can pass a `TMatchEvaluator` which is nothing more than a method that takes one parameter called `Match` of type `TMatch` and returns a string. The string returned by your method is used as a literal replacement string. If you want backreferences in your string to be replaced when using the `TMatchEvaluator` overload, call the `Result` method on the provided `Match` parameter before returning the string.

Use the `Split` method to split a string along its regex matches. The result is returned as a dynamic array of strings. As in .NET, text matched by capturing groups in the regular expression are also included in the returned array. If you don't like this, remove all named capturing groups from your regex and pass the `roExplicitCapture` option to disable numbered capturing groups. The non-static overload of `Split()`

takes an optional `Count` parameter to indicate the maximum number of elements that the returned array may have. In other words, the string is split at most `Count - 1` times. Capturing group matches are not included in the count. So if your regex has capturing groups, the returned array may have more than `Count` elements. If you pass `Count`, you can pass a second optional parameter to indicate the position in the string at which to start splitting. The part of the string before the starting position is returned unsplit in the first element of the returned array.

The `TMatch` record provides several properties with details about the match. `Success` indicates if a match was found. If this is `False`, all other properties and methods are invalid. `Value` returns the matched string. `Index` and `Length` indicate the position in the input string and the length of the match. `Groups` returns a `TGroupCollection` record that stores a `TGroup` record in its default `Item[]` property for each capturing group. You can use a numeric index to `Item[]` for numbered capturing groups, and a string index for named capturing groups.

`TMatch` also provides two methods. `NextMatch` returns the next match of the regular expression after this one. If your `TMatch` is part of a `TMatchCollection` you should not use `NextMatch` to get the next match but use `TMatchCollection.Item[]` instead, in order to avoid repeating the search. `TMatch.Result` takes one parameter with the replacement text as a string using the JGsoft replacement text flavor. It returns the string that this match would have been replaced with if you had used this replacement text with `TRegex.Replace`.

The `TGroup` record has `Success`, `Value`, `Index` and `Length` properties that work just like those of the `TMatch`.

In Delphi XE5 and prior `TRegex` always skips zero-length matches. This was fixed in Delphi XE6. You can make the same fix in XE5 and prior by modifying `RegularExpressionsCore.pas` to remove the line `State := [preNotEmpty]` from `TPerlRegex.Create`. This change will also affect code that uses `TPerlRegex` directly without setting the `State` property.

Regular Expressions Classes for Older Versions of Delphi

`TPerlRegex` has been available long before Embarcadero licensed a copy for inclusion with Delphi XE. Depending on your needs, you can download one of two versions for use with Delphi 2010 and earlier.

- Download the latest class-based `TPerlRegex`
- Download the older component-based `TPerlRegex`

The latest release of `TPerlRegex` is fully compatible with the `RegularExpressionsCore` unit in Delphi XE. For new code written in Delphi 2010 or earlier, using the latest release of `TPerlRegex` is strongly recommended. If you later migrate your code to Delphi XE, all you have to do is replace `PerlRegex` with `RegularExpressionsCore` in the `uses` clause of your units.

The older versions of `TPerlRegex` are non-visual components. This means you can put `TPerlRegex` on the component palette and drop it on a form. The original `TPerlRegex` was developed when Borland's goal was to have a component for everything on the component palette.

If you want to migrate from an older version of `TPerlRegex` to the latest `TPerlRegex`, start with removing any `TPerlRegex` components you may have placed on forms or data modules and instantiate the objects at

runtime instead. When instantiating at runtime, you no longer need to pass an owner component to the `Create()` constructor. Simply remove the parameter.

Some of the property and method names in the original `TPerlRegEx` were a bit unwieldy. These have been renamed in the latest `TPerlRegEx`. Essentially, in all identifiers `SubExpression` was replaced with `Group` and `MatchedExpression` was replaced with `Matched`. Here is a complete list of the changed identifiers:

Old Identifier	New Identifier
<code>StoreSubExpression</code>	<code>StoreGroups</code>
<code>NamedSubExpression</code>	<code>NamedGroup</code>
<code>MatchedExpression</code>	<code>MatchedText</code>
<code>MatchedExpressionLength</code>	<code>MatchedLength</code>
<code>MatchedExpressionOffset</code>	<code>MatchedOffset</code>
<code>SubExpressionCount</code>	<code>GroupCount</code>
<code>SubExpressions</code>	<code>Groups</code>
<code>SubExpressionLengths</code>	<code>GroupLengths</code>
<code>SubExpressionOffsets</code>	<code>GroupOffsets</code>

If you're using `RegexBuddy` or `RegexMagic` to generate Delphi code snippets, set the language to "Delphi (`TPerlRegEx`)" to use the old identifiers, or to "Delphi XE (Core)" to use the new identifiers, regardless of which (older) version of Delphi you're actually using.

UTF-8 Versus UTF-16

One thing you need to watch out for is that the `TPerlRegEx` versions you can download here as well as those included with Delphi XE, XE2, and XE3 use `UTF8String` properties and all the `Offset` and `Length` properties are indexes to those UTF-8 strings. This is because at that time PCRE only supported UTF-8 and using `UTF8String` avoids repeated conversions. If performance is critical, you should use `TPerlRegEx` instead of `TRegEx` with these versions of Delphi. If your data is already UTF-8, you can pass the UTF-8 directly to `TPerlRegEx`. If your data uses another encoding, you can control when the conversion to UTF-8 happens to avoid repeated conversions of the same data.

In Delphi XE4 and XE5 `TPerlRegEx` has `UnicodeString` (UTF-16) properties but still returns UTF-8 offsets and lengths. In Delphi XE6 the `Offset` and `Length` properties were changed to UTF-16 offsets and lengths. This means that code that works with XE3 or XE6 that uses the `Offset` and `Length` properties will not work with XE4 and XE5 if your strings contain non-ASCII characters. Delphi XE4 through and Delphi 10 through 10.2 continued to use the UTF-8 version of PCRE even though PCRE already had native UTF-16 support. This combined with the use of `UnicodeString` means constant conversions between UTF-16 and UTF-8 which can significantly degrade regex performance, particularly with long subject strings.

Delphi 10.3 and later use the UTF-16 version of PCRE on the Windows platform. `TRegEx` and `TPerlRegEx` now use `UnicodeString` for everything, without any conversion to UTF-8. Upgrading from Delphi XE4 or later to 10.3 or later will definitely improve the performance of any code that uses `TRegEx` or `TPerlRegEx`. Upgrading from Delphi XE3 or prior will improve performance unless you were doing everything with UTF-8.

Use System.Text.RegularExpressions with Delphi Prism

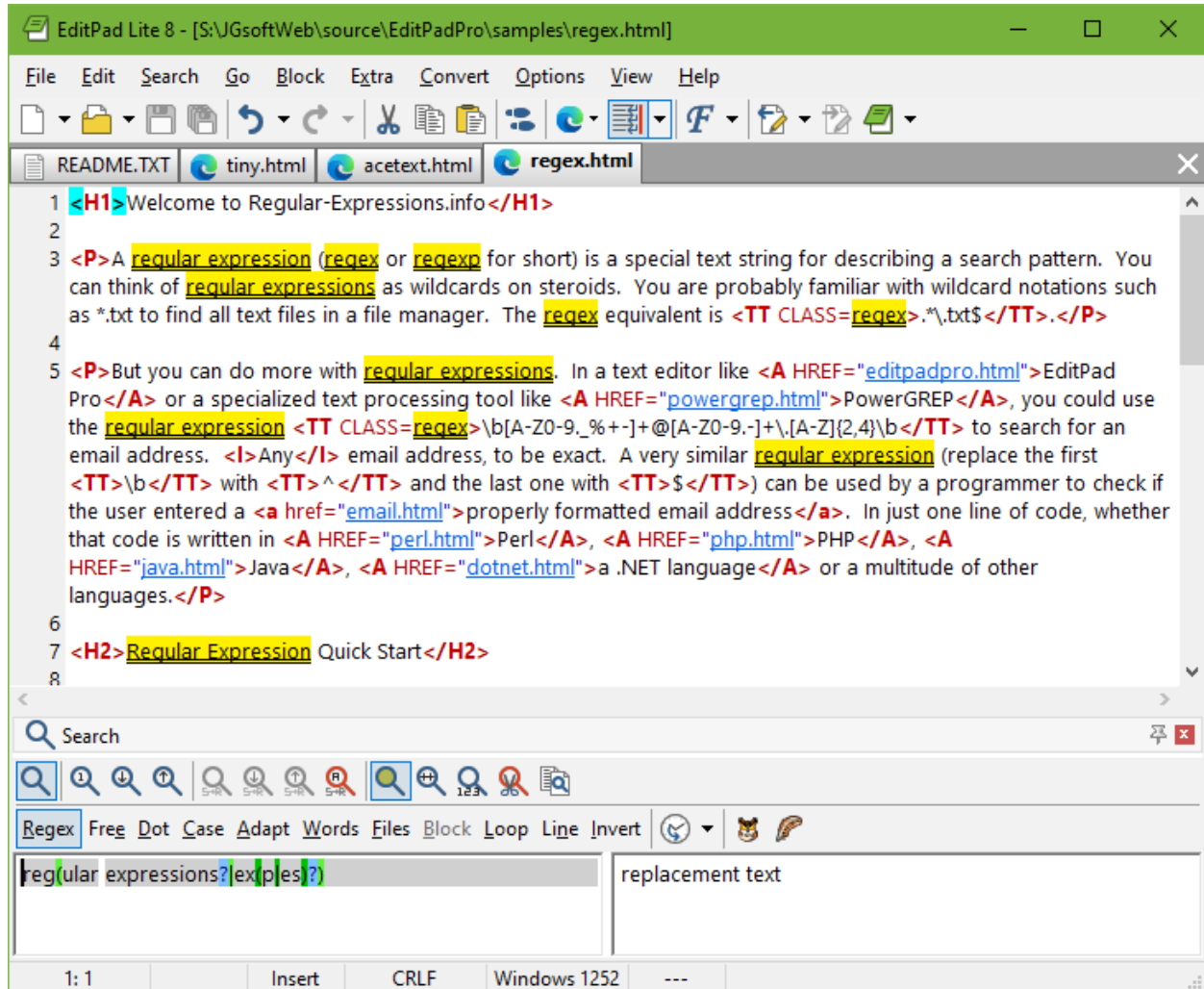
Delphi Prism was Embarcadero's variant of the Delphi language specifically developed to target the .NET framework. Delphi Prism lived inside the Visual Studio IDE. It was based entirely on the .NET framework. In Delphi Prism you could simply add the System.Text.RegularExpressions namespace to the uses clause of your units. Then you could access the .NET regex classes such as Regex, Match, and Group. You could then use them with Delphi Prism just as they can be used by C# and VB developers.

Use System.Text.RegularExpressions with Delphi for .NET

Delphi 8, 2005, 2006, and 2007 included a Delphi for .NET compiler for developing WinForms and VCL.NET applications. Though Delphi for .NET only supported .NET 1.1 or 2.0, depending on your Delphi version, you could still use .NET's full regular expression support. You only needed to add the System.Text.RegularExpressions namespace to the uses clause of your units to be able to access all the .NET regex classes.

4. EditPad Lite: Basic Text Editor with Full Regular Expression Support

EditPad Lite is a basic yet handy text editor for the Microsoft Windows platform. It has all the essential features for editing plain text files, including a complete set of search-and-replace features using a powerful regular expression engine.



EditPad Lite's Regular Expression Support

EditPad Lite doesn't use a limited and outdated regular expression engine like so many other text editors do. EditPad Lite uses the same full-featured regular expression engine used by PowerGREP. EditPad Lite's regex flavor is highly compatible with the flavors used by Perl, Java, .NET and many other modern Perl-style regular expression flavors.

EditPad Lite integrates with RegexBuddy and RegexMagic. You can instantly fire up RegexBuddy to edit the regex you want to use in EditPad Lite, select one from a RegexBuddy library, or generate one using RegexMagic.

Search and Replace Using Regular Expressions

Pressing Ctrl+F in EditPad Lite will make the search and replace pane appear. Mark the box labeled “regular expressions” to enable regex mode. Type in the regex you want to search for, and hit the Find First or Find Next button. EditPad Lite will then highlight search match. If the search pane takes up too much space, simply close it after entering the regular expression. Press Ctrl+F3 to find the first match, or F3 to find the next one.

When there are no further regex matches, EditPad Lite doesn't interrupt you with a popup message that you have to OK. The text cursor and selection will simply stay where they were, and the find button that you clicked will flash briefly. This may seem a little subtle at first, but you'll quickly appreciate EditPad Lite staying out of your way and keeping you productive.

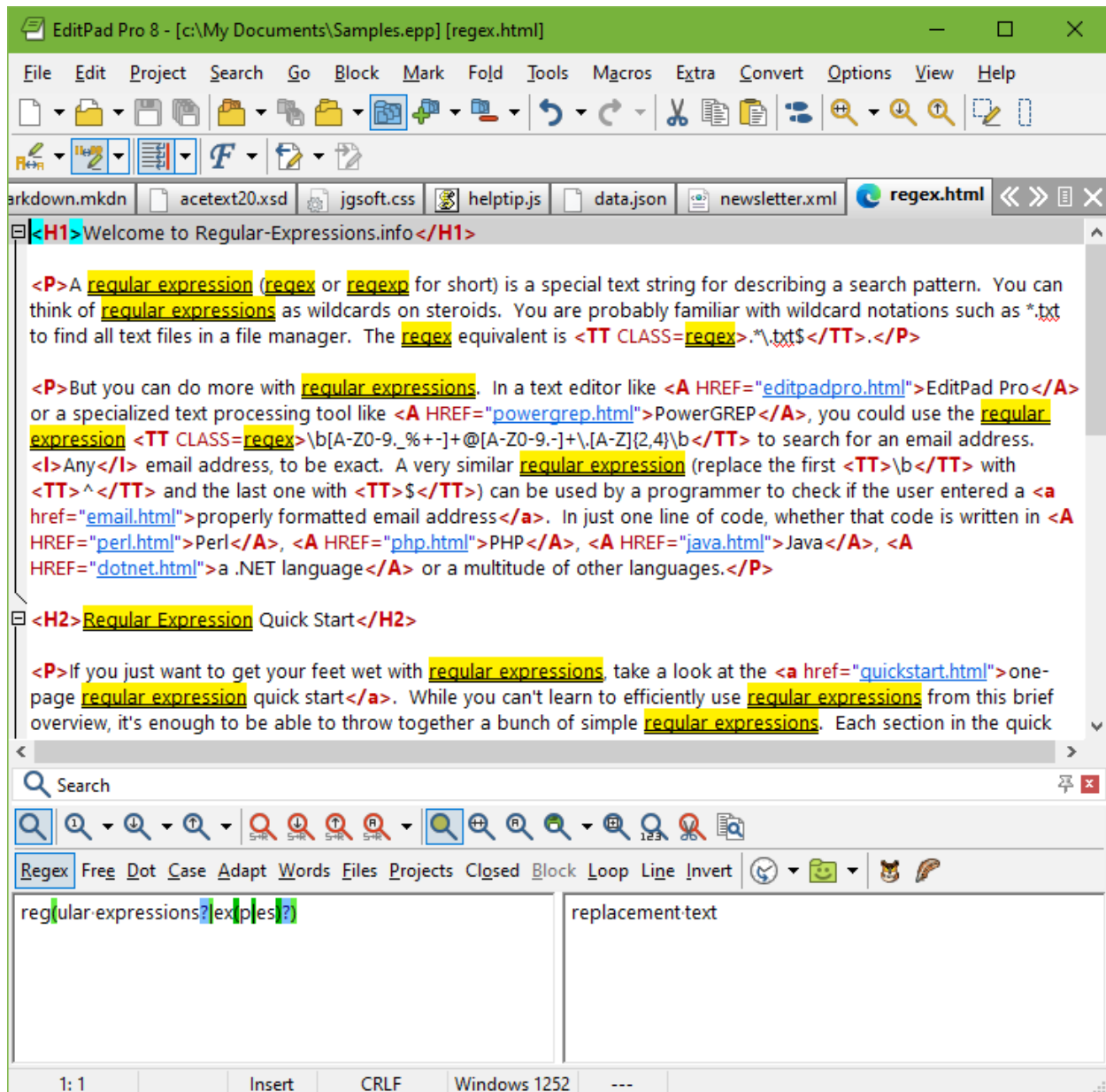
Replacing text is just as easy. First, type the replacement text, using backreferences if you want, in the Replace box. Search for the match you want to replace as above. To replace the current match, click the Replace button. To replace it and immediately search for the next match, click the Replace Next button. Or, click Replace All to get it over with.

More Information on EditPad Lite and Free Download

EditPad Lite works under Windows XP, Vista, 7, 8, 8.1, 10, and 11. For more information on EditPad Lite, please visit www.editpadlite.com. EditPad Lite is free for personal use. Business and government users can purchase a license.

5. EditPad Pro: Convenient Text Editor with Full Regular Expression Support

EditPad Pro is one of the most powerful and convenient text editors available on the Microsoft Windows platform. You can use EditPad Pro all day long without it getting into the way of what you are trying to do. When you use search & replace and the spell checker functionality, for example, you do not get a nasty popup window blocking your view of the document you are working on, but a small, extra pane just below the text. If you often work with many files at the same time, you will save time with the tabbed interface and the Project functionality for opening and saving sets of related files.



EditPad Pro's Regular Expression Support

EditPad Pro doesn't use a limited and outdated regular expression engine like so many other text editors do. EditPad Pro uses the same full-featured regular expression engine used by PowerGREP. EditPad Pro's regex flavor is highly compatible with the flavors used by Perl, Java, .NET and many other modern Perl-style regular expression flavors.

EditPad Pro integrates with RegxBuddy. You can instantly fire up RegxBuddy to edit the regex you want to use in EditPad Pro, select one from a RegxBuddy library, or generate one using RegxMagic.

Search and Replace Using Regular Expressions

Pressing Ctrl+F in EditPad Pro will make the search and replace pane appear. Mark the box labeled "regular expressions" to enable regex mode. Type in the regex you want to search for, and hit the Find First or Find Next button. EditPad Pro will then highlight search match. If the search pane takes up too much space, simply close it after entering the regular expression. Press Ctrl+F3 to find the first match, or F3 to find the next one.

When there are no further regex matches, EditPad Pro doesn't interrupt you with a popup message that you have to OK. The text cursor and selection will simply stay where they were, and the find button that you clicked will flash briefly. This may seem a little subtle at first, but you'll quickly appreciate EditPad Pro staying out of your way and keeping you productive.

Replacing text is just as easy. First, type the replacement text, using backreferences if you want, in the Replace box. Search for the match you want to replace as above. To replace the current match, click the Replace button. To replace it and immediately search for the next match, click the Replace Next button. Or, click Replace All to get it over with.

Syntax Coloring or Highlighting Schemes

Like many modern text editors, EditPad Pro supports syntax coloring or syntax highlighting for various popular file formats and programming languages. What makes EditPad Pro unique, is that you can use regular expressions to define your own syntax coloring schemes for file types not supported by default.

To create your own coloring scheme, all you need to do is download the custom syntax coloring schemes editor (only available if you have purchased EditPad Pro), and use regular expressions to specify the different syntactic elements of the file format or programming language you want to support. The regex engine used by the syntax coloring is identical to the one used by EditPad Pro's search and replace feature, so everything you learned in the tutorial in this book applies. Syntax coloring schemes can be shared on the EditPad Pro website.

The advantage is that you do not need to learn yet another scripting language or use a specific development tool to create your own syntax coloring schemes for EditPad Pro. All you need is decent knowledge of regular expressions.

File Navigation Schemes for Text Folding and Navigation

Text editors catering to programmers often allow you to fold certain sections in source code files to get a better overview. Another common feature is a sidebar showing you the file's structure, enabling you to quickly jump to a particular class definition or method implementation.

EditPad Pro also offers both these features, with one key difference. Most text editors only support folding and navigation for a limited set of file types, usually the more popular programming languages. If you use a less common language or file format, not to mention a custom one, you're out of luck.

EditPad Pro, however, implements folding and navigation using file navigation schemes. A bunch of them are included with EditPad Pro. These schemes are fully editable, and you can even create your own. Many file navigation schemes have been shared by other EditPad Pro users.

You can create and edit these schemes with a special file navigation scheme editor, which you can download after buying EditPad Pro. Like the syntax coloring schemes, file navigation schemes are based entirely on regular expressions. Because file navigation schemes are extremely flexible, editing them will take some effort. But with a bit of practice, you can make EditPad Pro's code folding and file navigation to work just the way you want it, and support all the file types that you work with, even proprietary ones.

More Information on EditPad Pro and Free Trial Download

EditPad Pro works under Windows XP, Vista, 7, 8, 8.1, 10, and 11. For more information on EditPad Pro, please visit www.editpadpro.com.

6. What Is grep?

Grep is a tool that originated from the UNIX world during the 1970's. It can search through files and folders (directories in UNIX) and check which lines in those files match a given regular expression. Grep will output the filenames and the line numbers or the actual lines that matched the regular expression. All in all a very useful tool for locating information stored anywhere on your computer, even (or especially) if you do not really know where to look.

Using grep

If you type `grep regex *.txt` grep will search through all text files in the current folder. It will apply the regex to each line in the files, and print (i.e. display) each line on which a match was found. This means that grep is inherently line-based. Regex matches cannot span multiple lines.

If you like to work on the command line, the traditional grep tool will make a lot of tasks easier. All Linux distributions (except tiny floppy-based ones) install a version of grep by default, usually GNU grep. If you are using Microsoft Windows, you will need to download and install it separately. If you use Borland development tools, you already have Borland's Turbo GREP installed.

grep not only works with globbed files, but also with anything you supply on the standard input. When used with standard input, grep will print all lines it reads from standard input that match the regex. E.g.: the Linux `find` command will glob the current directory and print all file names it finds, so `find | grep regex` will print only the file names that match regex.

Grep's Regex Engine

Most versions of grep use a regex-directed engine, like the regex flavors discussed in the regex tutorial in this book . However, grep's regex flavor is very limited. On POSIX systems, it uses POSIX Basic Regular Expressions.

An enhanced version of grep is called egrep. It uses a text-directed engine. Since neither grep nor egrep support any of the special features such as lazy repetition or lookahead, and because grep and egrep only indicate whether a match was found on a particular line or not, this distinction does not matter, except that the text-directed engine is faster. On POSIX systems, egrep uses POSIX Extended Regular Expressions. Despite the name "extended", egrep is almost the same as grep. It just uses a slightly different regex syntax and adds support for alternation, but loses support for backreferences.

GNU grep, the most popular version of grep on Linux, uses both a text-directed and a regex-directed engine. If you use backreferences it uses the regex-directed engine. Otherwise, it uses the faster text-directed engine. Again, for the tasks that grep is designed for, this does not matter to you, the user. If you type the "grep" command, you'll use the GNU Basic Regular Expressions syntax. If you type the "egrep" command, you'll use the GNU Extended Regular Expressions syntax. The GNU versions of grep and egrep have exactly the same capabilities, including alternation for grep and backreferences for egrep. They only use a slightly different syntax.

Beyond The Command Line

If you like to work on the command line, then the traditional grep tool is for you. But if you like to use a graphical user interface, there are many grep-like tools available for Windows and other platforms. Simply search for “grep” on your favorite software download site. Unfortunately, many grep tools come with poor documentation, leaving it up to you to figure out exactly which regex flavor they use. It’s not because they claim to be Perl-compatible, that they actually are. Some are almost perfectly compatible (but never identical, though), but others fail miserably when you want to use advanced and very useful constructs like lookaround.

One Windows-based grep tool that stands out from the crowd is PowerGREP.

7. GNU Regular Expression Extensions

GNU, which is an acronym for “GNU’s Not Unix”, is a project that strives to provide the world with free and open implementations of all the tools that are commonly available on Unix systems. Most Linux systems come with the full suite of GNU applications. This obviously includes traditional regular expression utilities like `grep`, `sed` and `awk`.

GNU’s implementation of these tools follows the POSIX standard, with added GNU extensions. The effect of the GNU extensions is that both the Basic Regular Expressions flavor and the Extended Regular Expressions flavor provide exactly the same functionality. The only difference is that BRE’s will use backslashes to give various characters a special meaning, while ERE’s will use backslashes to take away the special meaning of the same characters.

GNU Basic Regular Expressions (`grep`, `ed`, `sed`)

The Basic Regular Expressions or BRE flavor is pretty much the oldest regular expression flavor still in use today. The GNU utilities `grep`, `ed` and `sed` use it. One thing that sets this flavor apart is that most metacharacters require a backslash to give the metacharacter its flavor. Most other flavors, including GNU ERE, use a backslash to suppress the meaning of metacharacters. Using a backslash to escape a character that is never a metacharacter is an error.

A BRE supports POSIX bracket expressions, which are similar to character classes in other regex flavors, with a few special features. Other features using the usual metacharacters are the dot to match any character except a line break, the caret and dollar to match the start and end of the string, and the star to repeat the token zero or more times. To match any of these characters literally, escape them with a backslash.

The other BRE metacharacters require a backslash to give them their special meaning. The reason is that the oldest versions of UNIX `grep` did not support these. The developers of `grep` wanted to keep it compatible with existing regular expressions, which may use these characters as literal characters. The BRE `a{1,2}` matches `a{1,2}` literally, while `a\{1,2\}` matches `a` or `aa`. Tokens can be grouped with `\(` and `\)`. Backreferences are the usual `\1` through `\9`. Only up to 9 groups are permitted. E.g. `\(ab\)\1` matches `abab`, while `(ab)\1` is invalid since there’s no capturing group corresponding to the backreference `\1`. Use `\\1` to match `\1` literally.

On top of what POSIX BRE provides as described above, the GNU extension provides `\?` and `\+` as an alternative syntax to `\{0,1\}` and `\{1,\}`. It adds alternation via `\|`, something sorely missed in POSIX BREs. These extensions in fact mean that GNU BREs have exactly the same features as GNU EREs, except that `+`, `?`, `|`, braces and parentheses need backslashes to give them a special meaning instead of take it away.

GNU Extended Regular Expressions (`egrep`, `awk`, `emacs`)

The Extended Regular Expressions or ERE flavor is used by the GNU utilities `egrep` and `awk` and the `emacs` editor. In this context, “extended” is purely a historic reference. The GNU extensions make the BRE and ERE flavors identical in functionality.

All metacharacters have their meaning without backslashes, just like in modern regex flavors. You can use backslashes to suppress the meaning of all metacharacters. Escaping a character that is not a metacharacter is an error.

The quantifiers `?`, `+`, `{n}`, `{n,m}` and `{n,}` repeat the preceding token zero or once, once or more, `n` times, between `n` and `m` times, and `n` or more times, respectively. Alternation is supported through the usual vertical bar `|`. Unadorned parentheses create a group, e.g. `(abc){2}` matches `abcabc`.

POSIX ERE does not support backreferences. The GNU Extension adds them, using the same `\1` through `\9` syntax.

Additional GNU Extensions

The GNU extensions not only make both flavors identical. They also adds some new syntax and several brand new features. The shorthand classes `\w`, `\W`, `\s` and `\S` can be used instead of `[:alnum:]_`, `[^[:alnum:]_]`, `[:space:]` and `[^[:space:]]`. You can use these directly in the regex, but not inside bracket expressions. A backslash inside a bracket expression is always a literal.

The new features are word boundaries and anchors. Like modern flavors, GNU supports `\b` to match at a position that is at a word boundary, and `\B` at a position that is not. `\<` matches at a position at the start of a word, and `\>` matches at the end of a word. The anchor `\`` (backtick) matches at the very start of the subject string, while `\'` (single quote) matches at the very end. These are useful with tools that can match a regex against multiple lines of text at once, as then `^` will match at the start of a line, and `$` at the end.

Gnulib

GNU wouldn't be GNU if you couldn't use their regular expression implementation in your own (open source) applications. To do so, you'll need to download Gnulib. Use the included `gnulib-tool` to copy the regex module to your application's source tree.

The regex module provides the standard POSIX functions `regcomp()` for compiling a regular expression, `regerror()` for handling compilation errors, `regexexec()` to run a search using a compiled regex, and `regfree()` to clean up a regex you're done with.

8. Using Regular Expressions in Groovy

Because Groovy is based on Java, you can use Java's regular expression package with Groovy. Simply put `import java.util.regex.*` at the top of your Groovy source code. Any Java code using regular expressions will then automatically work in your Groovy code too.

Using verbose Java code to work with regular expressions in Groovy wouldn't be very groovy. Groovy has a bunch of language features that make code using regular expressions a lot more concise. You can mix the Groovy-specific syntax with regular Java code. It's all based in the `java.util.regex` package, which you'll need to import regardless.

Groovy Strings

Java has only one string style. Strings are placed between double quotes. Double quotes and backslashes in strings must be escaped with backslashes. That yields a forest of backslashes in literal regular expressions.

Groovy has five string styles. Strings can be placed between single quotes, double quotes, triple single quotes, and triple double quotes. Using triple single or double quotes allows the string to span multiple lines, which is handy for free-spacing regular expressions. Unfortunately, all four of these string styles require backslashes to be escaped.

The fifth string style is provided specifically for regular expressions. The string is placed between forward slashes, and only forward slashes (not backslashes) in the string need to be escaped. This is indeed a string style. Both `/hello/` and `"hello"` are literal instances of `java.lang.String`. Unfortunately, strings delimited with forward slashes cannot span across lines, so you can't use them for free-spacing regular expressions.

Groovy Patterns and Matchers

To actually use a string as a regular expression, you need to instantiate the `java.util.regex.Pattern` class. To actually use that pattern on a string, you need to instantiate the `java.util.regex.Matcher` class. You use these classes in Groovy just like you do in Java. But Groovy does provide some special syntax that allows you to create those instances with much less typing.

To create a `Pattern` instance, simply place a tilde before the string with your regular expression. The string can use any of Groovy's five string styles. When assigning this pattern to a variable, make sure to leave a space between the assignment operator and the tilde.

```
Pattern myRegex = ~/regex/
```

You won't actually instantiate patterns this way very often. The only time you need the `Pattern` instance is to split a string, which requires you to call `Pattern.split()`. To find regex matches or to search-and-replace with a regular expression, you need a `Matcher` instance that binds the pattern to a string. In Groovy, you can create this instance directly from the literal string with your regular expression using the `=~` operator. No space between the `=` and `~` this time.

```
Matcher myMatcher = "subject" =~ /regex/
```


This short for:

```
Matcher myMatcher = Pattern.compile(/regex/).matcher("subject")
```

Finally, the `==~` operator is a quick way to test whether a regex can match a string entirely. `myString ==~ /regex/` is equivalent to `myString.matches(/regex/)`. To find partial matches, you need to use the `Matcher`.

9. Using Regular Expressions in Java

Java 4 (JDK 1.4) and later have comprehensive support for regular expressions through the standard `java.util.regex` package. Because Java lacked a regex package for so long, there are also many 3rd party regex packages available for Java. I will only discuss Sun's regex library that is now part of the JDK. Its quality is excellent, better than most of the 3rd party packages. Unless you need to support older versions of the JDK, the `java.util.regex` package is the way to go.

Java 5 fixes some bugs and adds support for Unicode blocks. Java 6 fixes a few more bugs but doesn't add any features. Java 7 adds named capture and Unicode scripts. Java 13 allows infinite quantifiers inside lookbehind.

Quick Regex Methods of The String Class

The Java String class has several methods that allow you to perform an operation using a regular expression on that string in a minimal amount of code. The downside is that you cannot specify options such as "case insensitive" or "dot matches newline". For performance reasons, you should also not use these methods if you will be using the same regular expression often.

`myString.matches("regex")` returns true or false depending whether the string can be matched entirely by the regular expression. It is important to remember that `String.matches()` only returns true if the entire string can be matched. In other words: "regex" is applied as if you had written "`^regex$`" with start and end of string anchors. This is different from most other regex libraries, where the "quick match test" method returns true if the regex can be matched anywhere in the string. If `myString` is `abc` then `myString.matches("bc")` returns false. `bc` matches `abc`, but `^bc$` (which is really being used here) does not.

`myString.replaceAll("regex", "replacement")` replaces all regex matches inside the string with the replacement string you specified. No surprises here. All parts of the string that match the regex are replaced. You can use the contents of capturing parentheses in the replacement text via `$1`, `$2`, `$3`, etc. `$0` (dollar zero) inserts the entire regex match. `$12` is replaced with the 12th backreference if it exists, or with the 1st backreference followed by the literal "2" if there are less than 12 backreferences. If there are 12 or more backreferences, it is not possible to insert the first backreference immediately followed by the literal "2" in the replacement text.

In the replacement text, a dollar sign not followed by a digit causes an `IllegalArgumentException` to be thrown. If there are less than 9 backreferences, a dollar sign followed by a digit greater than the number of backreferences throws an `IndexOutOfBoundsException`. So be careful if the replacement string is a user-specified string. To insert a dollar sign as literal text, use `\$` in the replacement text. When coding the replacement text as a literal string in your source code, remember that the backslash itself must be escaped too: `"\\$"`.

`myString.split("regex")` splits the string at each regex match. The method returns an array of strings where each element is a part of the original string between two regex matches. The matches themselves are not included in the array. Use `myString.split("regex", n)` to get an array containing at most `n` items. The result is that the string is split at most `n-1` times. The last item in the string is the unsplit remainder of the original string.

Using The Pattern Class

In Java, you compile a regular expression by using the `Pattern.compile()` class factory. This factory returns an object of type `Pattern`. E.g.: `Pattern myPattern = Pattern.compile("regex");` You can specify certain options as an optional second parameter. `Pattern.compile("regex", Pattern.CASE_INSENSITIVE | Pattern.DOTALL | Pattern.MULTILINE)` makes the regex case insensitive for US ASCII characters, causes the dot to match line breaks and causes the start and end of string anchors to match at embedded line breaks as well. When working with Unicode strings, specify `Pattern.UNICODE_CASE` if you want to make the regex case insensitive for all characters in all languages. You should always specify `Pattern.CANON_EQ` to ignore differences in Unicode encodings, unless you are sure your strings contain only US ASCII characters and you want to increase performance.

If you will be using the same regular expression often in your source code, you should create a `Pattern` object to increase performance. Creating a `Pattern` object also allows you to pass matching options as a second parameter to the `Pattern.compile()` class factory. If you use one of the `String` methods above, the only way to specify options is to embed mode modifier into the regex. Putting `(?i)` at the start of the regex makes it case insensitive. `(?m)` is the equivalent of `Pattern.MULTILINE`, `(?s)` equals `Pattern.DOTALL` and `(?u)` is the same as `Pattern.UNICODE_CASE`. Unfortunately, `Pattern.CANON_EQ` does not have an embedded mode modifier equivalent.

Use `myPattern.split("subject")` to split the subject string using the compiled regular expression. This call has exactly the same results as `myString.split("regex")`. The difference is that the former is faster since the regex was already compiled.

Using The Matcher Class

Except for splitting a string (see previous paragraph), you need to create a `Matcher` object from the `Pattern` object. The `Matcher` will do the actual work. The advantage of having two separate classes is that you can create many `Matcher` objects from a single `Pattern` object, and thus apply the regular expression to many subject strings simultaneously.

To create a `Matcher` object, simply call `Pattern.matcher()` like this: `myMatcher = Pattern.matcher("subject")`. If you already created a `Matcher` object from the same pattern, call `myMatcher.reset("newssubject")` instead of creating a new matcher object, for reduced garbage and increased performance. Either way, `myMatcher` is now ready for duty.

To find the first match of the regex in the subject string, call `myMatcher.find()`. To find the next match, call `myMatcher.find()` again. When `myMatcher.find()` returns false, indicating there are no further matches, the next call to `myMatcher.find()` will find the first match again. The `Matcher` is automatically reset to the start of the string when `find()` fails.

The `Matcher` object holds the results of the last match. Call its methods `start()`, `end()` and `group()` to get details about the entire regex match and the matches between capturing parentheses. Each of these methods accepts a single int parameter indicating the number of the backreference. Omit the parameter to get information about the entire regex match. `start()` is the index of the first character in the match. `end()` is the index of the first character after the match. Both are relative to the start of the subject string. So the length of the match is `end() - start()`. `group()` returns the string matched by the regular expression or pair of capturing parentheses.

`myMatcher.replaceAll("replacement")` has exactly the same results as `myString.replaceAll("regex", "replacement")`. Again, the difference is speed.

The `Matcher` class allows you to do a search-and-replace and compute the replacement text for each regex match in your own code. You can do this with the `appendReplacement()` and `appendTail()`. Here is how:

```
StringBuffer myStringBuffer = new StringBuffer();
myMatcher = myPattern.matcher("subject");
while (myMatcher.find()) {
    if (checkIfThisMatchShouldBeReplaced()) {
        myMatcher.appendReplacement(myStringBuffer, computeReplacementString());
    }
}
myMatcher.appendTail(myStringBuffer);
```

Obviously, `checkIfThisMatchShouldBeReplaced()` and `computeReplacementString()` are placeholders for methods that you supply. The first returns true or false indicating if a replacement should be made at all. Note that skipping replacements is way faster than replacing a match with exactly the same text as was matched. `computeReplacementString()` returns the actual replacement string.

Regular Expressions, Literal Strings and Backslashes

In literal Java strings the backslash is an escape character. The literal string `"\"` is a single backslash. In regular expressions, the backslash is also an escape character. The regular expression `\"` matches a single backslash. This regular expression as a Java string, becomes `\"`. That's right: 4 backslashes to match a single one.

The regex `\w` matches a word character. As a Java string, this is written as `\"`.

The same backslash-mess occurs when providing replacement strings for methods like `String.replaceAll()` as literal Java strings in your Java code. In the replacement text, a dollar sign must be encoded as `\"` and a backslash as `\"` when you want to replace the regex match with an actual dollar sign or backslash. However, backslashes must also be escaped in literal Java strings. So a single dollar sign in the replacement text becomes `\"` when written as a literal Java string. The single backslash becomes `\"`. Right again: 4 backslashes to insert a single one.

10. Using Regular Expressions with JavaScript

JavaScript’s regular expression flavor is part of the ECMA-262 standard for the language. This means your regular expressions should work exactly the same in all implementations of JavaScript. In the past there were many serious browser-specific issues. But modern browsers do a very good job of following the JavaScript standard for regular expressions. You only need to make sure your web pages have a doctype that requests the browser to use standards mode rather than quirks mode.

JavaScript’s Regular Expression Flavor

In JavaScript source code, a regular expression is written in the form of `/pattern/modifiers` where “pattern” is the regular expression itself, and “modifiers” are a series of characters indicating various options. The “modifiers” part is optional. This syntax is borrowed from Perl. JavaScript supports the following modifiers, a subset of those supported by Perl:

- `/g` enables “global” matching. When using the `replace()` method, specify this modifier to replace all matches, rather than only the first one.
- `/i` makes the regex match case insensitive.
- `/m` enables “multi-line mode”. In this mode, the caret and dollar match before and after line breaks in the subject string.
- `/s` enables “single-line mode”. In this mode, the dot matches line breaks. This modifier is new in ECMAScript 2018. Older browsers, including Internet Explorer and the original Edge, do not support it.

You can combine multiple modifiers by stringing them together as in `/regex/gim`. Notably absent is an option to make the dot match line break characters.

Since forward slashes delimit the regular expression, any forward slashes that appear in the regex need to be escaped. E.g. the regex `1/2` is written as `/1\/2/` in JavaScript.

To match absolutely any character without `/s`, you can use character class that contains a shorthand class and its negated version, such as `[\s\S]`.

JavaScript implements Perl-style regular expressions. However, it lacks quite a number of advanced features available in Perl and other modern regular expression flavors:

- No `\A` or `\Z` anchors to match the start or end of the string. Use a caret or dollar instead.
- No atomic grouping or possessive quantifiers.
- No Unicode support, except for matching single characters with `\uFFFF`.
- No named capturing groups. Use numbered capturing groups instead.
- No mode modifiers to set matching options within the regular expression.
- No conditionals.
- No regular expression comments. Describe your regular expression with JavaScript `//` comments instead, outside the regular expression string.

Many of these features are available in the XRegExp library for JavaScript.

- Lookbehind was a major omission in JavaScript’s regex syntax for the longest time. Lookbehind is part of the ECMAScript 2018 specification. It is supported by the latest versions of Chrome, Edge, and Firefox but not by older browsers such as Internet Explorer.

Regex Methods of The String Class

To test if a particular regex matches (part of) a string, you can call the string’s `match()` method: `if (myString.match(/regex/)) { /*Success!*/ }`. If you want to verify user input, you should use anchors to make sure that you are testing against the entire string. To test if the user entered a number, use: `myString.match(/^\d+$/)`. `/\d+/` matches any string containing one or more digits, but `/^\d+$/` matches only strings consisting entirely of digits.

To do a search and replace with regexes, use the string’s `replace()` method: `myString.replace(/replaceme/g, "replacement")`. Using the `/g` modifier makes sure that all occurrences of “replaceme” are replaced. The second parameter is a normal string with the replacement text.

Using a string’s `split()` method allows you to split the string into an array of strings using a regular expression to determine the positions at which the string is split. E.g. `myArray = myString.split(/,/)` splits a comma-delimited list into an array. The comma’s themselves are not included in the resulting array of strings.

How to Use The JavaScript RegExp Object

The easiest way to create a new `RegExp` object is to simply use the special regex syntax: `myregexp = /regex/`. If you have the regular expression in a string (e.g. because it was typed in by the user), you can use the `RegExp` constructor: `myregexp = new RegExp(regexstring)`. Modifiers can be specified as a second parameter: `myregexp = new RegExp(regexstring, "gim")`.

I recommend that you do not use the `RegExp` constructor with a literal string, because in literal strings, backslashes must be escaped. The regular expression `\w+` can be created as `re = /\w+/` or as `re = new RegExp("\\w+")`. The latter is definitely harder to read. The regular expression `\\` matches a single backslash. In JavaScript, this becomes `re = /\\/` or `re = new RegExp("\\\\")`.

Whichever way you create “myregexp”, you can pass it to the String methods explained above instead of a literal regular expression: `myString.replace(myregexp, "replacement")`.

If you want to retrieve the part of the string that was matched, call the `exec()` function of the `RegExp` object that you created, e.g.: `mymatch = myregexp.exec("subject")`. This function returns an array. The zeroth item in the array will hold the text that was matched by the regular expression. The following items contain the text matched by the capturing parentheses in the regex, if any. `mymatch.length` indicates the length of the `match[]` array, which is one more than the number of capturing groups in your regular expression. `mymatch.index` indicates the character position in the subject string at which the regular expression matched. `mymatch.input` keeps a copy of the subject string.

Calling the `exec()` function also changes the `lastIndex` property of the `RegExp` object. It stores the index in the subject string at which the next match attempt will begin. You can modify this value to change the starting position of the next call to `exec()`.

The `test()` function of the `RegExp` object is a shortcut to `exec() != null`. It takes the subject string as a parameter and returns `true` or `false` depending on whether the regex matches part of the string or not.

You can call these methods on literal regular expressions too. `/\d/.test(subject)` is a quick way to test whether there are any digits in the subject string.

Replacement Text Syntax

The `String.replace()` function interprets several placeholders in the replacement text string. If the regex contains capturing groups, you can use backreferences in the replacement text. `$1` in the replacement text inserts the text matched by the first capturing group, `$2` the second, and so on until `$99`. If your regex has more than 1 but fewer than 10 capturing groups, then `$10` is treated as a backreference to the first group followed by a literal zero. If your regex has fewer than 7 capturing groups, then `$7` is treated as the literal text `$7`. `&` reinserts the whole regex match. ``` (backtick) inserts the text to the left of the regex match, `'` (single quote) inserts the text to the right of the regex match. `$$` inserts a single dollar sign, as does any `$` that does not form one of the placeholders described here.

`$_` and `$+` are not part of the standard but are supported by some browsers nonetheless. In Internet Explorer, `$_` inserts the entire subject string. In Internet Explorer and Firefox, `$+` inserts the text matched by the highest-numbered capturing group in the regex. If the highest-numbered group did not participate in the match, `$+` is replaced with nothing. This is not the same as `$+` in Perl, which inserts the text matched by the highest-numbered capturing group that actually participated in the match.

While things like `&` are actually variables in Perl that work anywhere, in JavaScript these only exist as placeholders in the replacement string passed to the `replace()` function.

11. MySQL Regular Expressions with The REGEXP Operator

MySQL's support for regular expressions is rather limited, but still very useful. MySQL only has one operator that allows you to work with regular expressions. This is the REGEXP operator, which works just like the LIKE operator, except that instead of using the `_` and `%` wildcards, it uses a POSIX Extended Regular Expression (ERE). Despite the "extended" in the name of the standard, the POSIX ERE flavor is a fairly basic regex flavor by modern standards. Still, it makes the REGEXP operator far more powerful and flexible than the simple LIKE operator.

One important difference between the LIKE and REGEXP operators is that the LIKE operator only returns True if the pattern matches the whole string. E.g. `WHERE testcolumn LIKE 'jg'` will return only rows where testcolumn is identical to `jg`, except for differences in case perhaps. On the other hand, `WHERE testcolumn REGEXP 'jg'` will return all rows where testcolumn has `jg` anywhere in the string. Use `WHERE testcolumn REGEXP '^jg$'` to get only columns identical to `jg`. The equivalent of `WHERE testcolumn LIKE 'jg%'` would be `WHERE testcolumn REGEXP '^jg'`. There's no need to put a `.` at the end of the regex (the REGEXP equivalent of LIKE's `%`), since partial matches are accepted.

MySQL does not offer any matching modes. POSIX EREs don't support mode modifiers inside the regular expression, and MySQL's REGEXP operator does not provide a way to specify modes outside the regular expression. The dot matches all characters including newlines, and the caret and dollar only match at the very start and end of the string. In other words: MySQL treats newline characters like ordinary characters. The REGEXP operator applies regular expressions case insensitively if the collation of the table is case insensitive, which is the default. If you change the collation to be case sensitive, the REGEXP operator becomes case sensitive.

Remember that MySQL supports C-style escape sequences in strings. While POSIX ERE does not support tokens like `\n` to match non-printable characters like line breaks, MySQL does support this escape in its strings. So `WHERE testcolumn REGEXP '\n'` returns all rows where testcolumn contains a line break. MySQL converts the `\n` in the string into a single line break character before parsing the regular expression. This also means that backslashes need to be escaped. The regex `\\` to match a single backslash becomes `'\\\\'` as a MySQL string, and the regex `\\$` to match a dollar symbol becomes `'\\$'` as a MySQL string. All this is unlike other databases like Oracle, which don't support `\n` and don't require backslashes to be escaped.

To return rows where the column doesn't match the regular expression, use `WHERE testcolumn NOT REGEXP 'pattern'`. The RLIKE operator is a synonym of the REGEXP operator. `WHERE testcolumn RLIKE 'pattern'` and `WHERE testcolumn NOT RLIKE 'pattern'` are identical to `WHERE testcolumn REGEXP 'pattern'` and `WHERE testcolumn NOT REGEXP 'pattern'`. I recommend you use REGEXP instead of RLIKE, to avoid confusion with the LIKE operator.

LIB_MYSQLUDF_PREG

If you want more regular expression power in your database, you can consider using LIB_MYSQLUDF_PREG. This is an open source library of MySQL user functions that imports the PCRE library. LIB_MYSQLUDF_PREG is delivered in source code form only. To use it, you'll need to be able to compile it and install it into your MySQL server. Installing this library does not change MySQL's built-in regex support in any way. It merely makes the following additional functions available:

PREG_CAPTURE extracts a regex match from a string. PREG_POSITION returns the position at which a regular expression matches a string. PREG_REPLACE performs a search-and-replace on a string. PREG_RLIKE tests whether a regex matches a string.

All these functions take a regular expression as their first parameter. This regular expression must be formatted like a Perl regular expression operator. E.g. to test if `regex` matches the subject case insensitively, you'd use the MySQL code `PREG_RLIKE('/regex/i', subject)`. This is similar to PHP's preg functions, which also require the extra `//` delimiters for regular expressions inside the PHP string.

12. Using Regular Expressions with Microsoft .NET

Microsoft .NET, which you can use with any .NET programming language such as C# (C sharp) or Visual Basic.NET, has solid support for regular expressions. .NET's regex flavor is very feature-rich. The only noteworthy features that are lacking are possessive quantifiers and subroutine calls.

There are no differences in the regex flavor supported by the .NET Framework versions 2.0 through 4.8. There are no differences between this flavor and the flavor supported by any version of .NET Core either. That includes the original .NET Core 1.0.0 and the latest .NET 5.0.

There are a few differences between the regex flavor in the .NET Framework 1.x compared with later versions. The .NET Framework 2.0 fixes a few bugs. The Unicode categories `\p{Pi}` and `\p{Pf}` are no longer reversed. Unicode blocks with hyphens in their names are now handled correctly. One feature was added in .NET 2.0: character class subtraction. It works exactly the way it does in XML Schema regular expressions. The XML Schema standard first defined this feature and its syntax.

System.Text.RegularExpressions Overview (Using VB.NET Syntax)

The regex classes are located in the namespace `System.Text.RegularExpressions`. To make them available, place `Imports System.Text.RegularExpressions` at the start of your source code.

The `Regex` class is the one you use to compile a regular expression. For efficiency, regular expressions are compiled into an internal format. If you plan to use the same regular expression repeatedly, construct a `Regex` object as follows: `Dim RegexObj as Regex = New Regex("regexexpression")`. You can then call `RegexObj.IsMatch("subject")` to check whether the regular expression matches the subject string. The `Regex` allows an optional second parameter of type `RegexOptions`. You could specify `RegexOptions.IgnoreCase` as the final parameter to make the regex case insensitive. Other options are `RegexOptions.IgnorePatternWhitespace` which makes the regex free-spacing, `RegexOptions.Singleline` which makes the dot to match newlines, `RegexOptions.Multiline` which makes the caret and dollar to match at embedded newlines in the subject string, and `RegexOptions.ExplicitCapture` which turns all unnamed groups into non-capturing groups.

Call `RegexObj.Replace("subject", "replacement")` to perform a search-and-replace using the regex on the subject string, replacing all matches with the replacement string. In the replacement string, you can use `&` to insert the entire regex match into the replacement text. You can use `$1`, `$2`, `$3`, etc. to insert the text matched between capturing parentheses into the replacement text. Use `$$` to insert a single dollar sign into the replacement text. To replace with the first backreference immediately followed by the digit 9, use `${1}9`. If you type `$19`, and there are less than 19 backreferences, then `$19` will be interpreted as literal text, and appear in the result string as such. To insert the text from a named capturing group, use `{name}`. Improper use of the `$` sign may produce an undesirable result string, but will never cause an exception to be raised.

`RegexObj.Split("Subject")` splits the subject string along regex matches, returning an array of strings. The array contains the text between the regex matches. If the regex contains capturing parentheses, the text matched by them is also included in the array. If you want the entire regex matches to be included in the array, simply place parentheses around the entire regular expression when instantiating `RegexObj`.

The `Regex` class also contains several static methods that allow you to use regular expressions without instantiating a `Regex` object. This reduces the amount of code you have to write, and is appropriate if the

same regular expression is used only once or reused seldomly. Note that member overloading is used a lot in the `Regex` class. All the static methods have the same names (but different parameter lists) as other non-static methods.

`Regex.IsMatch("subject", "regex")` checks if the regular expression matches the subject string. `Regex.Replace("subject", "regex", "replacement")` performs a search-and-replace. `Regex.Split("subject", "regex")` splits the subject string into an array of strings as described above. All these methods accept an optional additional parameter of type `RegexOptions`, like the constructor.

The System.Text.RegularExpressions.Match Class

If you want more information about the regex match, call `Regex.Match()` to construct a `Match` object. If you instantiated a `Regex` object, use `Dim MatchObj as Match = RegexObj.Match("subject")`. If not, use the static version: `Dim MatchObj as Match = Regex.Match("subject", "regex")`.

Either way, you will get an object of class `Match` that holds the details about the first regex match in the subject string. `MatchObj.Success` indicates if there actually was a match. If so, use `MatchObj.Value` to get the contents of the match, `MatchObj.Length` for the length of the match, and `MatchObj.Index` for the start of the match in the subject string. The start of the match is zero-based, so it effectively counts the number of characters in the subject string to the left of the match.

If the regular expression contains capturing parentheses, use the `MatchObj.Groups` collection. `MatchObj.Groups.Count` indicates the number of capturing parentheses. The count includes the zeroth group, which is the entire regex match. `MatchObj.Groups(3).Value` gets the text matched by the third pair of parentheses. `MatchObj.Groups(3).Length` and `MatchObj.Groups(3).Index` get the length of the text matched by the group and its index in the subject string, relative to the start of the subject string. `MatchObj.Groups("name")` gets the details of the named group "name".

To find the next match of the regular expression in the same subject string, call `MatchObj.NextMatch()` which returns a new `Match` object containing the results for the second match attempt. You can continue calling `MatchObj.NextMatch()` until `MatchObj.Success` is `False`.

Note that after calling `RegexObj.Match()`, the resulting `Match` object is independent from `RegexObj`. This means you can work with several `Match` objects created by the same `Regex` object simultaneously.

Regular Expressions, Literal Strings and Backslashes

In literal `C#` strings, as well as in `C++` and many other .NET languages, the backslash is an escape character. The literal string `"\"` is a single backslash. In regular expressions, the backslash is also an escape character. The regular expression `\"` matches a single backslash. This regular expression as a `C#` string, becomes `"\\\"`. That's right: 4 backslashes to match a single one.

The regex `\w` matches a word character. As a `C#` string, this is written as `"\\w"`.

To make your code more readable, you should use `C#` verbatim strings. In a verbatim string, a backslash is an ordinary character. This allows you to write the regular expression in your `C#` code as you would write it a tool like `RegexBuddy` or `PowerGREP`, or as the user would type it into your application. The regex to match

a backslash is written as @"\" when using C# verbatim strings. The backslash is still an escape character in the regular expression, so you still need to double it. But doubling is better than quadrupling. To match a word character, use the verbatim string @"\w".

RegexOptions.ECMAScript

Passing `RegexOptions.ECMAScript` to the `Regex()` constructor changes the behavior of certain regex features to follow the behavior prescribed in the ECMA-262 standard. This standard defines the ECMAScript language, which is better known as JavaScript. The table below compares the differences between canonical .NET (without the ECMAScript option) and .NET in ECMAScript mode. For reference the table also compares how JavaScript in modern browsers behaves in these areas.

Feature or Syntax	Canonical .NET	.NET in ECMAScript mode	JavaScript
<code>RegexOptions.FreeSpacing</code>	Supported	Only via <code>(?x)</code>	Not supported
<code>RegexOptions.SingleLine</code>	Supported	Only via <code>(?s)</code>	Not supported
<code>RegexOptions.ExplicitCapture</code>	Supported	Only via <code>(?n)</code>	Not supported
Escaped letter or underscore that does not form a regex token	Error	Literal letter or underscore	
Escaped digit that is not a valid backreference	Error	Octal escape or literal 8 or 9	
Escaped double digits that do not form a valid backreference	Error	Single digit backreference and literal digit if the single digit backreference is valid; otherwise single or double digit octal escape and/or literal 8 and 9	
Backreference to non-participating group	Fails to match	Zero-length match	
Forward reference	Supported	Error	Zero-length match
Backreference to group 0	Fails to match	Zero-length match	Syntactically not possible
<code>\s</code>	Unicode	ASCII	Unicode
<code>\d</code>	Unicode	ASCII	
<code>\w</code>	Unicode	ASCII	
<code>\b</code>	Unicode	ASCII	

Though `RegexOptions.ECMAScript` brings the .NET regex engine a little bit closer to JavaScript's, there are still significant differences between the .NET regex flavor and the JavaScript regex flavor. When creating web pages using ASP.NET on the server and JavaScript on the client, you cannot assume the same regex to work in the same way both on the client side and the server side even when setting `RegexOptions.ECMAScript`. The next table lists the more important differences between .NET and JavaScript. `RegexOptions.ECMAScript` has no impact on any of these.

The table also compares the `XRegExp` library for JavaScript. You can use this library to bring JavaScript's regex flavor a little bit closer to .NET's.

Feature or syntax	.NET	XRegExp	JavaScript
Dot	<code>[^\n]</code>	<code>[^\n\r\u2028\u2029]</code>	

Anchors in multi-line mode	Treat only <code>\n</code> as a line break	Treat <code>\n</code> , <code>\r</code> , <code>\u2028</code> , and <code>\u2029</code> as line breaks	
<code>\$</code> without multi-line mode	Matches at very end of string	Matches before final line break and at very end of string	
Permanent start and end of string anchors	Supported	Not supported	
Empty character class	Syntactically not possible	fails to match	
Lookbehind	Supported without restrictions	Supported (without ECMAScript 2018 restrictions)	since
Mode modifiers	Anywhere	At start of regex only	Not supported
Comments	Supported		Not supported
Unicode properties	Categories and blocks		Not supported
Named capture backreferences	Supported		Not supported
Balancing groups	Supported	Not supported	
Conditionals	Supported	Not supported	

13. Oracle Database Regular Expressions

With version 10g Release 1, Oracle Database offers 4 regexp functions that you can use in SQL and PL/SQL statements. These functions implement the POSIX Extended Regular Expressions (ERE) standard. Oracle fully supports collating sequences and equivalence classes in bracket expressions. The NLS_SORT setting determines the POSIX locale used, which determines the available collating sequences and equivalence classes.

Oracle does not implement the POSIX ERE standard exactly, however. It deviates in three areas. First, Oracle supports the backreferences \1 through \9 in the regular expression. The POSIX ERE standard does not support these, even though POSIX BRE does. In a fully compliant engine, \1 through \9 would be illegal. The POSIX standard states it is illegal to escape a character that is not a metacharacter with a backslash. Oracle allows this, and simply ignores the backslash. E.g. \q is identical to q in Oracle. The result is that all POSIX ERE regular expressions can be used with Oracle, but some regular expressions that work in Oracle may cause an error in a fully POSIX-compliant engine. Obviously, if you only work with Oracle, these differences are irrelevant.

The third difference is more subtle. It won't cause any errors, but may result in different matches. As I explained in the topic about the POSIX standard, it requires the regex engine to return the longest match in case of alternation. Oracle's engine does not do this. It is a traditional NFA engine, like all non-POSIX regex flavors discussed in this book.

If you've worked with regular expressions in other programming languages, be aware that POSIX does not support non-printable character escapes like \t for a tab or \n for a newline. You can use these with a POSIX engine in a programming language like C++, because the C++ compiler will interpret the \t and \n in string constants. In SQL statements, you'll need to type an actual tab or line break in the string with your regular expression to make it match a tab or line break. Oracle's regex engine will interpret the string '\t' as the regex t when passed as the regexp parameter.

Oracle 10g R2 further extends the regex syntax by adding a free-spacing mode (without support for comments), shorthand character classes, lazy quantifiers, and the anchors \A, \Z, and \z. Oracle 11g and 12c use the same regex flavor as 10g R2.

Oracle's REGEXP Functions

Oracle Database 10g offers four regular expression functions. You can use these equally in your SQL and PL/SQL statements.

REGEXP_LIKE(source, regexp, modes) is probably the one you'll use most. You can use it in the WHERE and HAVING clauses of a SELECT statement. In a PL/SQL script, it returns a Boolean value. You can also use it in a CHECK constraint. The source parameter is the string or column the regex should be matched against. The regexp parameter is a string with your regular expression. The modes parameter is optional. It sets the matching modes.

```
SELECT * FROM mytable WHERE REGEXP_LIKE(mycolumn, 'regexp', 'i');
IF REGEXP_LIKE('subject', 'regexp') THEN /* Match */ ELSE /* No match */ END IF;
ALTER TABLE mytable ADD (CONSTRAINT mycolumn_regexp CHECK (REGEXP_LIKE(mycolumn, '^regexp$')));
```

REGEXP_SUBSTR(source, regexp, position, occurrence, modes) returns a string with the part of source matched by the regular expression. If the match attempt fails, NULL is returned. You can use REGEXP_SUBSTR with a single string or with a column. You can use it in SELECT clauses to retrieve only a certain part of a column. The position parameter specifies the character position in the source string at which the match attempt should start. The first character has position 1. The occurrence parameter specifies which match to get. Set it to 1 to get the first match. If you specify a higher number, Oracle will continue to attempt to match the regex starting at the end of the previous match, until it found as many matches as you specified. The last match is then returned. If there are fewer matches, NULL is returned. Do not confuse this parameter with backreferences. Oracle does not provide a function to return the part of the string matched by a capturing group. The last three parameters are optional.

```
SELECT REGEXP_SUBSTR(mycolumn, 'regexp') FROM mytable;
match := REGEXP_SUBSTR('subject', 'regexp', 1, 1, 'i')
```

REGEXP_REPLACE(source, regexp, replacement, position, occurrence, modes) returns the source string with one or all regex matches replaced. If no matches can be found, the original string is replaced. If you specify a positive number for occurrence (see the above paragraph) only that match is replaced. If you specify zero or omit the parameter, all matches are replaced. The last three parameters are optional. The replacement parameter is a string that each regex match will be replaced with. You can use the backreferences \1 through \9 in the replacement text to re-insert text matched by a capturing group. You can reference the same group more than once. There's no replacement text token to re-insert the whole regex match. To do that, put parentheses around the whole regexp, and use \1 in the replacement. If you want to insert \1 literally, use the string '\\1'. Backslashes only need to be escaped if they're followed by a digit or another backslash. To insert \\ literally, use the string '\\\\'. While SQL does not require backslashes to be escaped in strings, the REGEXP_REPLACE function does.

```
SELECT REGEXP_REPLACE(mycolumn, 'regexp', 'replacement') FROM mytable;
result := REGEXP_REPLACE('subject', 'regexp', 'replacement', 1, 0, 'i');
```

REGEXP_INSTR(source, regexp, position, occurrence, return_option, modes) returns the beginning or ending position of a regex match in the source string. This function takes the same parameters as REGEXP_SUBSTR, plus one more. Set return_option to zero or omit the parameter to get the position of the first character in match. Set it to one to get the position of the first character after the match. The first character in the string has position 1. REGEXP_INSTR returns zero if the match cannot be found. The last 4 parameters are optional.

```
SELECT REGEXP_INSTR(mycolumn, 'regexp', 1, 1, 0, 'i') FROM mytable;
```

REGEXP_COUNT(source, regexp, position, modes) returns the number of times the regex can be matched in the source string. It returns zero if the regex finds no matches at all. This function is only available in Oracle 11g and later.

```
SELECT REGEXP_COUNT(mycolumn, 'regexp', 1, 'i') FROM mytable;
```

Oracle's Matching Modes

The modes parameter that each of the four regexp functions accepts should be a string of up to three characters, out of four possible characters. E.g. 'i' turns on case insensitive matching, while 'inm' turns on those three options. 'i' and 'c' are mutually exclusive. If you omit this parameter or pass an empty string, the default matching modes are used.

- 'i': Turn on case insensitive matching. The default depends on the NLS_SORT setting.
- 'c': Turn on case sensitive matching. The default depends on the NLS_SORT setting.
- 'n': Make the dot match any character, including newlines. By default, the dot matches any character except newlines.
- 'm': Make the caret and dollar match at the start and end of each line (i.e. after and before line breaks embedded in the source string). By default, these only match at the very start and the very end of the string.
- 'x': Turn on free-spacing mode which ignores any unescaped whitespace outside character classes (10gR2 and later).

14. The PCRE Open Source Regex Library

PCRE is short for Perl Compatible Regular Expressions. It is the name of an open source library written in C by Philip Hazel. The library is compatible with a great number of C compilers and operating systems. Many people have derived libraries from PCRE to make it compatible with other programming languages. The regex features included with PHP (prior to 7.3.0), Delphi, and R (prior to 4.0.0), and Xojo (REALbasic) are all based on PCRE. The library is also included with many Linux distributions as a shared .so library and a .h header file.

Though PCRE claims to be Perl-compatible, there are more than enough differences between contemporary versions of Perl and PCRE to consider them distinct regex flavors. Recent versions of Perl have even copied features from PCRE that PCRE had copied from other programming languages before Perl had them, in an attempt to make Perl more PCRE-compatible. Today PCRE is used more widely than Perl because PCRE is part of so many libraries and applications.

Philip Hazel has recently released a new library called PCRE2. The first PCRE2 release was given version number 10.00 to make a clear break with the previous PCRE 8.36. Future PCRE releases will be limited to bug fixes. New features will go into PCRE2 only. If you're taking on a new development project, you should consider using PCRE2 instead of PCRE. But for existing projects that already use PCRE, it's probably best to stick with PCRE. Moving from PCRE to PCRE2 requires significant changes to your source code (but not to your regular expressions).

You can find more information about PCRE and PCRE2 at <https://www.pcre.org/>.

Using PCRE

Using PCRE is very straightforward. Before you can use a regular expression, it needs to be converted into a binary format for improved efficiency. To do this, simply call `pcre_compile()` passing your regular expression as a null-terminated string. The function returns a pointer to the binary format. You cannot do anything with the result except pass it to the other pcre functions.

To use the regular expression, call `pcre_exec()` passing the pointer returned by `pcre_compile()`, the character array you want to search through, and the number of characters in the array (which need not be null-terminated). You also need to pass a pointer to an array of integers where `pcre_exec()` stores the results, as well as the length of the array expressed in integers. The length of the array should equal the number of capturing groups you want to support, plus one (for the entire regex match), multiplied by three (!). The function returns -1 if no match could be found. Otherwise, it returns the number of capturing groups filled plus one. If there are more groups than fit into the array, it returns 0. The first two integers in the array with results contain the start of the regex match (counting bytes from the start of the array) and the number of bytes in the regex match, respectively. The following pairs of integers contain the start and length of the backreferences. So `array[n*2]` is the start of capturing group `n`, and `array[n*2+1]` is the length of capturing group `n`, with capturing group 0 being the entire regex match.

When you are done with a regular expression, call `pcre_dispose()` with the pointer returned by `pcre_compile()` to prevent memory leaks.

The original PCRE library only supports regex matching, a job it does rather well. It provides no support for search-and-replace, splitting of strings, etc. This may not seem as a major issue because you can easily do

these things in your own code. The unfortunate consequence, however, is that all the programming languages and libraries that use PCRE for regex matching have their own replacement text syntax and their own idiosyncrasies when splitting strings. The new PCRE2 library does support search-and-replace.

Compiling PCRE with Unicode Support

By default, PCRE compiles without Unicode support. If you try to use `\p`, `\P` or `\X` in your regular expressions, PCRE will complain it was compiled without Unicode support.

To compile PCRE with Unicode support, you need to define the `SUPPORT_UTF8` and `SUPPORT_UCP` conditional defines. If PCRE's configuration script works on your system, you can easily do this by running `./configure --enable-unicode-properties` before running `make`. The regular expressions tutorial in this manual assumes that you've compiled PCRE with these options and that all other options are set to their defaults.

PCRE Callout

A feature unique to PCRE is the “callout”. If you put `(?C1)` through `(?C255)` anywhere in your regex, PCRE calls the `pcre_callout` function when it reaches the callout during the match attempt.

UTF-8, UTF-16, and UTF-32

By default, PCRE works with 8-bit strings, where each character is one byte. You can pass the `PCRE_UTF8` as the second parameter to `pcre_compile()` (possibly combined with other flavors using binary or) to tell PCRE to interpret your regular expression as a UTF-8 string. When you do this, `pcre_match()` automatically interprets the subject string using UTF-8 as well.

If you have PCRE 8.30 or later, you can enable UTF-16 support by passing `--enable-pcre16` to the `configure` script before running `make`. Then you can pass `PCRE_UTF16` to `pcre16_compile()` and then do the matching with `pcre16_match()` if your regular expression and subject strings are stored as UTF-16. UTF-16 uses two bytes for code points up to U+FFFF, and four bytes for higher code points. In Visual C++, `wchar_t` strings use UTF-16. It's important to make sure that you do not mix the `pcre_` and `pcre16_` functions. The `PCRE_UTF8` and `PCRE_UTF16` constants are actually the same. You need to use the `pcre16_` functions to get the UTF-16 version.

If you have PCRE 8.32 or later, you can enable UTF-32 support by passing `--enable-pcre32` to the `configure` script before running `make`. Then you can pass `PCRE_UTF32` to `pcre32_compile()` and then do the matching with `pcre32_match()` if your regular expression and subject strings are stored as UTF-32. UTF-32 uses four bytes per character and is common for in-memory Unicode strings on Linux. It's important to make sure that you do not mix the `pcre32_` functions with the `pcre16_` or `pcre_` sets. Again, the `PCRE_UTF8` and `PCRE_UTF32` constants are the same. You need to use the `pcre32_` functions to get the UTF-32 version.

15. The PCRE2 Open Source Regex Library

PCRE2 is short for Perl Compatible Regular Expressions, version 2. It is the successor to the widely popular PCRE library. Both are open source libraries written in C by Philip Hazel.

The first PCRE2 release was given version number 10.00 to make a clear break with the preceding PCRE 8.36. PCRE 8.37 through 8.44 and any future PCRE releases are limited to bug fixes. New features are added to PCRE2 only. If you're taking on a new development project, you should consider using PCRE2 instead of PCRE. But for existing projects that already use PCRE, it's probably best to stick with PCRE. Moving from PCRE to PCRE2 requires significant changes to your source code. The only real reason to do so would be to use the new search-and-replace feature.

PHP 7.3.0 moved from PCRE to PCRE2 but does not use the new PCRE2 search-and-replace. It continues to use PHP's own replacement text syntax.

R 4.0.0 also moved its `grep` and related functions from PCRE to PCRE2. Its `sub` and `gsub` functions continue to use R's own replacement text syntax.

The regex syntax supported by PCRE2 10.00 through 10.34 and PCRE 8.36 through 8.44 is pretty much the same. Because of this, everything the regex tutorial in this manual says about PCRE (or PCRE versions 8.36 through 8.44 in particular) also applies to PCRE2. PCRE2 is only mentioned in the few specific areas where it differs from PCRE.

The regex syntax has a new `versioncheck` conditional. The syntax looks much like a conditional that checks a named backreference, but the inclusion of the equals sign (and other symbols not allowed in capturing group names) make it a syntax error in the original PCRE. In all versions of PCRE2, `(?(VERSION>=10.00)yes|no)` matches `yes` in the string `yesno`. You can use any valid regex for the "yes" and "no" parts. If the version check succeeds the "yes" part is attempted. Otherwise the "no" part is attempted. This is exactly like a normal conditional that evaluates the part before or after the vertical bar depending on whether a capturing group participated in the match or not.

You can use `>=` to check for a minimum version, or `=` to check for a specific version. `(?(VERSION=10.00)yes|no)` matches `yes` in PCRE2 10.00. It matches `no` in PCRE2 10.10 and all later versions. Omitting the minor version number is the same as specifying `.00`. So `(?(VERSION>=10)yes|no)` matches `yes` in all versions of PCRE2, but `(?(VERSION=10)yes|no)` only matches `yes` in PCRE2 10.00. If you specify the minor version number you should use two digits after the decimal point. Three or more digits are an error as of version 10.21. Version 10.21 also changes the interpretation of single digit numbers, including those specified with a leading zero. Since the first release was 10.00 and the second release was 10.10 there should be no need to check for single digit numbers. You cannot omit the dot if you specify the minor version number. `(?(VERSION>=1000)yes|no)` checks for version 1000.00 or greater.

This version check conditional is mainly intended for people who use PCRE2 indirectly, via an application that provides regex support based on PCRE2 or a programming language that embeds PCRE2 but does not expose all its function calls. It allows them to find out which version of PCRE2 the application uses. If you are developing an application with the PCRE2 C library then you should use a function call to determine the PCRE2 version:

```
char version[255];
pcre2_config(PCRE2_CONFIG_VERSION, version);
```

UTF-8, UTF-16, or UTF-32

In the original PCRE library, UTF-16 and UTF-32 support were added in later versions through additional functions prefixed with `pcre16_` and `pcre32_`. In PCRE2, all functions are prefixed with `pcre2_` and suffixed with `_8`, `_16`, or `_32` to select 8-bit, 16-bit, or 32-bit code units. If you compile PCRE2 from source, you need to pass `--enable-pcre2-16` and `--enable-pcre2-32` to the `configure` script to make sure the `_16` and `_32` functions are available.

8-bit, 16-bit, or 32-bit code units means that PCRE2 interprets your string as consisting of single byte characters, double byte characters, or quad byte characters. To work with UTF-8, UTF-16, or UTF-32, you need to use the functions with the corresponding code unit size, and pass the `PCRE2_UTF` option to `pcre2_compile` to allow characters to consist of multiple code units. UTF-8 characters consist of 1 to 4 bytes. UTF-16 characters consist of 1 or 2 words.

If you want to call the PCRE2 functions without any suffix, as they are shown below, then you need to define `PCRE2_CODE_UNIT_WIDTH` as 8, 16, or 32 to make the functions without a suffix use 8-bit, 16-bit, or 32-bit code units. Do so before including the library, like this:

```
#define PCRE2_CODE_UNIT_WIDTH 8
#include "pcre2.h"
```

The functions without a suffix always use the code unit size you've defined. The functions with suffixes remain available. So your application can use regular expressions with all three code unit sizes. But it is important not to mix them up. If the same regex needs to be matched against UTF-8 and UTF-16 strings, then you need to compile it twice using `pcre_compile_8` and `pcre_compile_16` and then use the compiled regexes with the corresponding `pcre_match_8` and `pcre_match_16` functions.

Using PCRE2

Using PCRE2 is a bit more complicated than using PCRE. With PCRE2 you have to use various types of context to pass certain compilation or matching options, such as line break handling. In PCRE these options could be passed directly as option bits when compiling or matching.

Before you can use a regular expression, it needs to be converted into a binary format for improved efficiency. To do this, simply call `pcre2_compile()` passing your regular expression as a string. If the string is null-terminated, you can pass `PCRE2_ZERO_TERMINATED` as the second parameter. Otherwise, pass the length in code units as the second parameter. For UTF-8 this is the length in bytes, while for UTF-16 or UTF-32 this is the length in bytes divided by 2 or 4. The 3rd parameter is a set of options combined with `binary` or `or`. You should include `PCRE2_UTF` for proper UTF-8, UTF-16, or UTF-32 support. If you omit it, you get pure 8-bit, or UCS-2, or UCS-4 character handling. Other common options are `PCRE2_CASELESS`, `PCRE2_DOTALL`, `PCRE2_MULTILINE`, and so on. The 4th and 5th parameters receive error conditions. The final parameter is a context. Pass `NULL` unless you need special line break handling. The function returns a pointer to memory it allocated. You must free this with `pcre2_code_free()` when you're done with the regular expression.

If you need non-default line break handling, you need to call `pcre2_compile_context_create(NULL)` to create a new compile context. Then call `pcre2_set_newline()` passing that context and one of the options like `PCRE2_NEWLINE_LF` or `PCRE2_NEWLINE_CRLF`. Then pass this context as the final parameter to `pcre2_compile()`. You can reuse the same context for as many regex compilations as you like. Call

`pcre2_compile_context_free()` when you're done with it. Note that in the original PCRE, you could pass `PCRE_NEWLINE_LF` and the like directly to `pcre_compile()`. This does not work with PCRE2. PCRE2 will not complain if you pass `PCRE2_NEWLINE_LF` and the like to `pcre2_compile()`. But doing so has no effect. You have to use the match context.

Before you can use the compiled regular expression to find matches in a string, you need to call `pcre2_match_data_create_from_pattern()` to allocate memory to store the match results. Pass the compiled regex as the first parameter, and `NULL` as the second. The function returns a pointer to memory it allocated. You must free this with `pcre2_match_data_free()` when you're done with the match data. You can reuse the same match data for multiple calls to `pcre2_match()`.

To find a match, call `pcre2_match()` and pass the compiled regex, the subject string, the length of the entire string, the offset of the character where the match attempt must begin, matching options, the pointer to the match data object, and `NULL` for the context. The length and starting offset are in code units, not characters. The function returns a positive number when the match succeeds. `PCRE2_ERROR_NOMATCH` indicates no match was found. Any other non-positive return value indicates an error. Error messages can be obtained with `pcre2_get_error_message()`.

To find out which part of the string matched, call `pcre2_get_ovector_pointer()`. This returns a pointer to an array of `PCRE2_SIZE` values. You don't need to free this pointer. It will become invalid when you call `pcre2_match_data_free()`. The length of the array is the value returned by `pcre2_match()`. The first two values in the array are the start and end of the overall match. The second pair is the match of the first capturing group, and so on. Call `pcre2_substring_number_from_name()` to get group numbers if your regex has named capturing groups.

If you just want to get the matched text, you can use convenience functions like `pcre2_substring_copy_bynumber()` or `pcre2_substring_copy_byname()`. Pass the number or name of a capturing group, or zero for the overall match. Free the result with `pcre2_substring_free()`. If the result doesn't need to be zero-terminated, you can use `pcre2_substring_get_bynumber()` and `pcre2_substring_get_byname()` to get a pointer to the start of the match within the original subject string. `pcre2_substring_length_bynumber()` and `pcre2_substring_length_byname()` give you the length of the match.

PCRE2 does not provide a function that gives you all the matches of a regex in string. It never returns more than the first match. To get the second match, call `pcre2_match()` again and pass `ovector[1]` (the end of the first match) as the starting position for the second match attempt. If the first match was zero-length, include `PCRE2_NOTEMPTY_ATSTART` with the options passed to `pcre2_match()` in order to avoid finding the same zero-length match again. This is not the same as incrementing the starting position before the call. Passing the end of the previous match with `PCRE2_NOTEMPTY_ATSTART` may result in a non-zero-length match being found at the same position.

Substituting Matches

The biggest item item that was forever on PCRE's wishlist was likely the ability to search-and-replace. PCRE2 finally delivers. The replacement string syntax is fairly simple. There is nothing Perl-compatible about it, though. Backreferences can be specified as `$group` or `${group}` where "group" is either the name or the number of a group. The overall match is group number zero. To add literal dollar sign to the replacement, you need to double it up. Any single dollar sign that is not part of a valid backreference is an error. Like Python, but unlike Perl, PCRE2 treats backreferences to non-existent groups and backreferences to non-participating groups as errors. Backslashes are literals.

Before you can substitute matches, you need to compile your regular expression with `pcre2_compile()`. You can use the same compiled regex for both regex matching and regex substitution. You don't need a match data object for substitution.

Call `pcre2_substitute()` and pass it the compiled regex, the subject string, the length of the subject string, the position in the string where regex matching should begin, and the matching options. You can include `PCRE2_SUBSTITUTE_GLOBAL` with the matching options to replace all matches after the starting position, instead of just the first match. The next parameters are for match data and match context which can both be set to `NULL`. Then pass the replacement string and the length of the replacement string. Finally pass a pointer to a buffer where the result string should be stored, along with a pointer to a variable that holds the size of the buffer. The buffer needs to have space for a terminating zero. All lengths and offsets are in code units, not characters.

`pcre2_substitute()` returns the number of regex matches that were substituted. Zero means no matches were found. This function never returns `PCRE2_ERROR_NOMATCH`. A negative number indicates an error occurred. Call `pcre2_get_error_message()` to get the error message. The variable that holds the size of the buffer is updated to indicate the length of the string that it written to the buffer, excluding the terminating zero. (The terminating zero is written.)

Extended Replacement String Syntax

Starting with version 10.21, PCRE2 offers an extended replacement string syntax that you can enable by including `PCRE2_SUBSTITUTE_EXTENDED` with the matching options when calling `pcre2_substitute()`. The biggest difference is that backslashes are no longer literals. A backslash followed by a character that is not a letter or digit escapes that character. So you can escape dollar signs and backslashes with other backslashes to suppress their special meaning. If you want to have a literal backslash in your replacement, then you have to escape it with another backslash. Backslashes followed by a digit are an error. Backslashes followed by a letter are an error, unless that combination forms a replacement string token.

`\a`, `\e`, `\f`, `\n`, `\r`, and `\t` are the usual ASCII control character escapes. Notably missing are `\b` and `\v`. `\x0` through `\xF` and `\x00` through `\xFF` are hexadecimal escapes. `\x{0}` through `\x{10FFFF}` insert a Unicode code point. `\o{0}` through `\o{177777}` are octal escapes.

Case conversion is also supported. The syntax is the same as Perl, but the behavior is not. Perl allows you to combine `\u` or `\l` with `\L` or `\U` to make one character uppercase or lowercase and the remainder the opposite. With PCRE2, any case conversion escape cancels the preceding escape. So you can't combine them and even `\u` or `\l` will end a run of `\U` or `\L`.

Conditionals are supported using a newly invented syntax that extends the syntax for backreferences. `${group:+matched:unmatched}` inserts `matched` when the group participated and `unmatched` when it didn't. You can use the full replacement string syntax in the two alternatives, including other conditionals.

Case conversion runs through conditionals. Any case conversion in effect before the conditional also applies to the conditional. If the conditional contains its own case conversion escapes in the part of the conditional that is actually used, then those remain in effect after the conditional. So you could use `${1:+\U:\L}${2}` to insert the text matched by the second capturing group in uppercase if the first group participated, and in lowercase if it didn't.

16. Perl's Rich Support for Regular Expressions

Perl was originally designed by Larry Wall as a flexible text-processing language. Over the years, it has grown into a full-fledged programming language, keeping a strong focus on text processing. When the world wide web became popular, Perl became the de facto standard for creating CGI scripts. A CGI script is a small piece of software that generates a dynamic web page, based on a database and/or input from the person visiting the website. Since CGI script basically is a text-processing script, Perl was and still is a natural choice.

Because of Perl's focus on managing and mangling text, regular expression text patterns are an integral part of the Perl language. This in contrast with most other languages, where regular expressions are available as add-on libraries. In Perl, you can use the `m//` operator to test if a regex can match a string, e.g.:

```
if ($string =~ m/regex/) {
    print 'match';
} else {
    print 'no match';
}
```

Performing a regex search-and-replace is just as easy:

```
$string =~ s/regex/replacement/g;
```

I added a “g” after the last forward slash. The “g” stands for “global”, which tells Perl to replace all matches, and not just the first one. Options are typically indicated including the slash, like “/g”, even though you do not add an extra slash, and even though you could use any non-word character instead of slashes. If your regex contains slashes, use another character, like `s!regex!replacement!g`.

You can add an “i” to make the regex match case insensitive. You can add an “s” to make the dot match newlines. You can add an “m” to make the dollar and caret match at newlines embedded in the string, as well as at the start and end of the string.

Together you would get something like `m/regex/sim`;

Regex-Related Special Variables

Perl has a host of special variables that get filled after every `m//` or `s///` regex match. `$1`, `$2`, `$3`, etc. hold the backreferences. `$+` holds the last (highest-numbered) backreference. `$&` (dollar ampersand) holds the entire regex match.

`@-` is an array of match-start indices into the string. `$-[0]` holds the start of the entire regex match, `$-[1]` the start of the first backreference, etc. Likewise, `@+` holds match-end positions. To get the length of the match, subtract `$+[0]` from `$-[0]`.

In Perl 5.10 and later you can use the associative array `%+` to get the text matched by named capturing groups. For example, `${name}` holds the text matched by the group “name”. Perl does not provide a way to get match positions of capturing groups by referencing their names. Since named groups are also numbered, you can use `@-` and `@+` for named groups, but you have to figure out the group's number by yourself.

`$'` (dollar followed by an apostrophe or single quote) holds the part of the string after (to the right of) the regex match. `$`` (dollar backtick) holds the part of the string before (to the left of) the regex match. Using these variables is not recommended in scripts when performance matters, as it causes Perl to slow down *all* regex matches in your entire script.

All these variables are read-only, and persist until the next regex match is attempted. They are dynamically scoped, as if they had an implicit `'local'` at the start of the enclosing scope. Thus if you do a regex match, and call a sub that does a regex match, when that sub returns, your variables are still set as they were for the first match.

Finding All Matches In a String

The `"/g"` modifier can be used to process all regex matches in a string. The first `m/regex/g` will find the first match, the second `m/regex/g` the second match, etc. The location in the string where the next match attempt will begin is automatically remembered by Perl, separately for each string. Here is an example:

```
while ($string =~ m/regex/g) {  
    print "Found '$&'. Next attempt at character " . pos($string)+1 . "\n";  
}
```

The `pos()` function retrieves the position where the next attempt begins. The first character in the string has position zero. You can modify this position by using the function as the left side of an assignment, like in `pos($string) = 123;`.

17. PHP Provides Three Sets of Regular Expression Functions

PHP is an open source language for producing dynamic web pages. PHP has three sets of functions that allow you to work with regular expressions.

The most important set of regex functions start with `preg`. These functions are a PHP wrapper around the PCRE library (Perl-Compatible Regular Expressions). Anything said about the PCRE regex flavor in the regular expressions tutorial in this book applies to PHP's `preg` functions. When the tutorial talks about PHP specifically, it assumes you're using the `preg` functions. You should use the `preg` functions for all new PHP code that uses regular expressions. PHP includes PCRE by default as of PHP 4.2.0 (April 2002).

The oldest set of regex functions are those that start with `ereg`. They implement POSIX Extended Regular Expressions, like the traditional UNIX `egrep` command. These functions are mainly for backward compatibility with PHP 3. They are officially deprecated as of PHP 5.3.0. Many of the more modern regex features such as lazy quantifiers, lookaround and Unicode are not supported by the `ereg` functions. Don't let the "extended" moniker fool you. The POSIX standard was defined in 1986, and regular expressions have come a long way since then.

The last set is a variant of the `ereg` set, prefixing `mb_` for "multibyte" to the function names. While `ereg` treats the regex and subject string as a series of 8-bit characters, `mb_ereg` can work with multi-byte characters from various code pages. If you want your regex to treat Far East characters as individual characters, you'll either need to use the `mb_ereg` functions, or the `preg` functions with the `/u` modifier. `mb_ereg` is available in PHP 4.2.0 and later. It uses the same POSIX ERE flavor.

The preg Function Set

All of the `preg` functions require you to specify the regular expression as a string using Perl syntax. In Perl, `/regex/` defines a regular expression. In PHP, this becomes `preg_match('/regex/', $subject)`. When forward slashes are used as the regex delimiter, any forward slashes in the regular expression have to be escaped with a backslash. So `https://www.jgsoft.com/` becomes `'/https://www.jgsoft.com/'`. Just like Perl, the `preg` functions allow any non-alphanumeric character as regex delimiters. The URL regex would be more readable as `'%https://www.jgsoft.com/%'` using percentage signs as the regex delimiters, since then you don't need to escape the forward slashes. You would have to escape percentage signs if the regex contained any.

Unlike programming languages like C# or Java, PHP does not require all backslashes in strings to be escaped. If you want to include a backslash as a literal character in a PHP string, you only need to escape it if it is followed by another character that needs to be escaped. In single quoted-strings, only the single quote and the backslash itself need to be escaped. That is why in the above regex, I didn't have to double the backslashes in front of the literal dots. The regex `\\` to match a single backslash would become `'/\\\\/'` as a PHP `preg` string. Unless you want to use variable interpolation in your regular expression, you should always use single-quoted strings for regular expressions in PHP, to avoid messy duplication of backslashes.

To specify regex matching options such as case insensitivity are specified in the same way as in Perl. `'/regex/i'` applies the regex case insensitively. `'/regex/s'` makes the dot match all characters. `'/regex/m'` makes the start and end of line anchors match at embedded newlines in the subject string.

'/regex/x' turns on free-spacing mode. You can specify multiple letters to turn on several options. '/regex/misx' turns on all four options.

A special option is the /u which turns on the Unicode matching mode, instead of the default 8-bit matching mode. You should specify /u for regular expressions that use `\x{FFFF}`, `\X` or `\p{L}` to match Unicode characters, graphemes, properties or scripts. PHP will interpret '/regex/u' as a UTF-8 string rather than as an ASCII string.

Like the `ereg` function, bool `preg_match` (string pattern, string subject [, array groups]) returns TRUE if the regular expression pattern matches the subject string or part of the subject string. If you specify the third parameter, `preg` will store the substring matched by the first capturing group in `$groups[1]`. `$groups[2]` will contain the second pair, and so on. If the regex pattern uses named capture, you can access the groups by name with `$groups['name']`. `$groups[0]` will hold the overall match.

int `preg_match_all` (string pattern, string subject, array matches, int flags) fills the array “matches” with all the matches of the regular expression pattern in the subject string. If you specify `PREG_SET_ORDER` as the flag, then `$matches[0]` is an array containing the match and backreferences of the first match, just like the `$groups` array filled by `preg_match`. `$matches[1]` holds the results for the second match, and so on. If you specify `PREG_PATTERN_ORDER`, then `$matches[0]` is an array with full consecutive regex matches, `$matches[1]` an array with the first backreference of all matches, `$matches[2]` an array with the second backreference of each match, etc.

array `preg_grep` (string pattern, array subjects) returns an array that contains all the strings in the array “subjects” that can be matched by the regular expression pattern.

mixed `preg_replace` (mixed pattern, mixed replacement, mixed subject [, int limit]) returns a string with all matches of the regex pattern in the subject string replaced with the replacement string. At most `limit` replacements are made. One key difference is that all parameters, except `limit`, can be arrays instead of strings. In that case, `preg_replace` does its job multiple times, iterating over the elements in the arrays simultaneously. You can also use strings for some parameters, and arrays for others. Then the function will iterate over the arrays, and use the same strings for each iteration. Using an array of the pattern and replacement, allows you to perform a sequence of search and replace operations on a single subject string. Using an array for the subject string, allows you to perform the same search and replace operation on many subject strings.

`preg_replace_callback` (mixed pattern, callback replacement, mixed subject [, int limit]) works just like `preg_replace`, except that the second parameter takes a callback instead of a string or an array of strings. The callback function will be called for each match. The callback should accept a single parameter. This parameter will be an array of strings, with element 0 holding the overall regex match, and the other elements the text matched by capturing groups. This is the same array you’d get from `preg_match`. The callback function should return the text that the match should be replaced with. Return an empty string to delete the match. Return `$groups[0]` to skip this match.

Callbacks allow you to do powerful search-and-replace operations that you cannot do with regular expressions alone. E.g. if you search for the regex `(\d+)\s+(\d+)`, you can replace `2+3` with `5` using the callback:

```
function regexadd($groups) {
    return $groups[1] + $groups[2];
}
```

array **preg_split** (string pattern, string subject [, int limit]) works just like `split`, except that it uses the Perl syntax for the regex pattern.

See the PHP manual for more information on the `preg` function set

The `ereg` Function Set

The `ereg` functions require you to specify the regular expression as a string, as you would expect. `ereg('regex', "subject")` checks if `regex` matches `subject`. You should use single quotes when passing a regular expression as a literal string. Several special characters like the dollar and backslash are also special characters in double-quoted PHP strings, but not in single-quoted PHP strings.

int **ereg** (string pattern, string subject [, array groups]) returns the length of the match if the regular expression pattern matches the subject string or part of the subject string, or zero otherwise. Since zero evaluates to `False` and non-zero evaluates to `True`, you can use `ereg` in an `if` statement to test for a match. If you specify the third parameter, `ereg` will store the substring matched by the part of the regular expression between the first pair of parentheses in `$groups[1]`. `$groups[2]` will contain the second pair, and so on. Note that grouping-only parentheses are not supported by `ereg`. `ereg` is case sensitive. `eregi` is the case insensitive equivalent.

string **ereg_replace** (string pattern, string replacement, string subject) replaces all matches of the regex pattern in the subject string with the replacement string. You can use backreferences in the replacement string. `\\0` is the entire regex match, `\\1` is the first backreference, `\\2` the second, etc. The highest possible backreference is `\\9`. `ereg_replace` is case sensitive. `eregi_replace` is the case insensitive equivalent.

array **split** (string pattern, string subject [, int limit]) splits the subject string into an array of strings using the regular expression pattern. The array will contain the substrings between the regular expression matches. The text actually matched is discarded. If you specify a limit, the resulting array will contain at most that many substrings. The subject string will be split at most `limit-1` times, and the last item in the array will contain the unsplit remainder of the subject string. `split` is case sensitive. `spliti` is the case insensitive equivalent.

See the PHP manual for more information on the `ereg` function set

The `mb_ereg` Function Set

The `mb_ereg` functions work exactly the same as the `ereg` functions, with one key difference: while `ereg` treats the regex and subject string as a series of 8-bit characters, `mb_ereg` can work with multi-byte characters from various code pages. E.g. encoded with Windows code page 936 (Simplified Chinese), the word 中国 (“China”) consists of four bytes: `D6D0B9FA`. Using the `ereg` function with the regular expression `.` on this string would yield the first byte `D6` as the result. The dot matched exactly one byte, as the `ereg` functions are byte-oriented. Using the `mb_ereg` function after calling `mb_regex_encoding("CP936")` would yield the bytes `D6D0` or the first character 中 as the result.

To make sure your regular expression uses the correct code page, call `mb_regex_encoding()` to set the code page. If you don't, the code page returned by or set by `mb_internal_encoding()` is used instead.

If your PHP script uses UTF-8, you can use the `preg` functions with the `/u` modifier to match multi-byte UTF-8 characters instead of individual bytes. The `preg` functions do not support any other code pages.

See the PHP manual for more information on the `mb_ereg` function set

18. POSIX Basic Regular Expressions

POSIX or “Portable Operating System Interface for uniX” is a collection of standards that define some of the functionality that a (UNIX) operating system should support. One of these standards defines two flavors of regular expressions. Commands involving regular expressions, such as `grep` and `egrep`, implement these flavors on POSIX-compliant UNIX systems. Several database systems also use POSIX regular expressions.

The Basic Regular Expressions or BRE flavor standardizes a flavor similar to the one used by the traditional UNIX `grep` command. This is pretty much the oldest regular expression flavor still in use today. One thing that sets this flavor apart is that most metacharacters require a backslash to give the metacharacter its flavor. Most other flavors, including POSIX ERE, use a backslash to suppress the meaning of metacharacters. Using a backslash to escape a character that is never a metacharacter is an error.

A BRE supports POSIX bracket expressions, which are similar to character classes in other regex flavors, with a few special features. Shorthands are not supported. Other features using the usual metacharacters are the dot to match any character except a line break, the caret and dollar to match the start and end of the string, and the star to repeat the token zero or more times. To match any of these characters literally, escape them with a backslash.

The other BRE metacharacters require a backslash to give them their special meaning. The reason is that the oldest versions of UNIX `grep` did not support these. The developers of `grep` wanted to keep it compatible with existing regular expressions, which may use these characters as literal characters. The BRE `a{1,2}` matches `a{1,2}` literally, while `a\{1,2\}` matches `a` or `aa`. Some implementations support `\?` and `\+` as an alternative syntax to `\{0,1\}` and `\{1,\}`, but `\?` and `\+` are not part of the POSIX standard. Tokens can be grouped with `\(` and `\)`. Backreferences are the usual `\1` through `\9`. Only up to 9 groups are permitted. E.g. `\(ab\) \1` matches `abab`, while `(ab)\1` is invalid since there’s no capturing group corresponding to the backreference `\1`. Use `\\1` to match `\1` literally.

POSIX BRE does not support any other features. Even alternation is not supported.

POSIX Extended Regular Expressions

The Extended Regular Expressions or ERE flavor standardizes a flavor similar to the one used by the UNIX `egrep` command. “Extended” is relative to the original UNIX `grep`, which only had bracket expressions, dot, caret, dollar and star. An ERE support these just like a BRE. Most modern regex flavors are extensions of the ERE flavor. By today’s standard, the POSIX ERE flavor is rather bare bones. The POSIX standard was defined in 1986, and regular expressions have come a long way since then.

The developers of `egrep` did not try to maintain compatibility with `grep`, creating a separate tool instead. Thus `egrep`, and POSIX ERE, add additional metacharacters without backslashes. You can use backslashes to suppress the meaning of all metacharacters, just like in modern regex flavors. Escaping a character that is not a metacharacter is an error.

The quantifiers `?`, `+`, `{n}`, `{n,m}` and `{n,}` repeat the preceding token zero or once, once or more, `n` times, between `n` and `m` times, and `n` or more times, respectively. Alternation is supported through the usual vertical bar `|`. Unadorned parentheses create a group, e.g. `(abc){2}` matches `abcabc`. The POSIX standard does not define backreferences. Some implementations do support `\1` through `\9`, but these are not part of the standard for ERE. ERE is an extension of the old UNIX `grep`, not of POSIX BRE.

And that's exactly how far the extension goes.

POSIX ERE Alternation Returns The Longest Match

In the tutorial topic about alternation, I explained that the regex engine will stop as soon as it finds a matching alternative. The POSIX standard, however, mandates that the longest match be returned. When applying `Set|SetValue` to `SetValue`, a POSIX-compliant regex engine will match `SetValue` entirely. Even if the engine is a regex-directed NFA engine, POSIX requires that it simulates DFA text-directed matching by trying all alternatives, and returning the longest match, in this case `SetValue`. A traditional NFA engine would match `Set`, as do all other regex flavors discussed in this book.

A POSIX-compliant engine will still find the leftmost match. If you apply `Set|SetValue` to `Set or SetValue` once, it will match `Set`. The first position in the string is the leftmost position where our regex can find a valid match. The fact that a longer match can be found further in the string is irrelevant. If you apply the regex a second time, continuing at the first space in the string, then `SetValue` will be matched. A traditional NFA engine would match `Set` at the start of the string as the first match, and `Set` at the start of the 3rd word in the string as the second match.

19. PostgreSQL Has Three Regular Expression Flavors

PostgreSQL 7.4 and later use the exact same regular expression engine that was developed by Henry Spencer for Tcl 8.2. This means that PostgreSQL supports the same three regular expressions flavors: Tcl Advanced Regular Expressions, POSIX Extended Regular Expressions and POSIX Basic Regular Expressions. Just like in Tcl, AREs are the default. All my comments on Tcl's regular expression flavor, like the unusual mode modifiers and word boundary tokens, fully apply to PostgreSQL. You should definitely review them if you're not familiar with Tcl's AREs. Unfortunately, PostgreSQL's `regexp_replace` function does not use the same syntax for the replacement text as Tcl's `regsub` command, however.

PostgreSQL versions prior to 7.4 supported POSIX Extended Regular Expressions only. If you are migrating old database code to a new version of PostgreSQL, you can set PostgreSQL's "`regex_flavor`" run-time parameter to "`extended`" instead of the default "`advanced`" to make EREs the default.

PostgreSQL also supports the traditional SQL `LIKE` operator, and the SQL:1999 `SIMILAR TO` operator. These use their own pattern languages, which are not discussed here. AREs are far more powerful, and no more complicated if you don't use functionality not offered by `LIKE` or `SIMILAR TO`.

The Tilde Operator

The tilde infix operator returns true or false depending on whether a regular expression can match part of a string, or not. E.g. `'subject' ~ 'regexp'` returns false, while `'subject' ~ '\\w'` returns true. If the regex must match the whole string, you'll need to use anchors. E.g. `'subject' ~ '^\\w$'` returns false, while `'subject' ~ '^\\w+$'` returns true. There are 4 variations of this operator:

- `~` attempts a case sensitive match
- `~*` attempts a case insensitive match
- `!~` attempts a case sensitive match, and returns true if the regex does not match any part of the subject string
- `!~*` attempts a case insensitive match, and returns true if the regex does not match any part of the subject string

While only case sensitivity can be toggled by the operator, all other options can be set using mode modifiers at the start of the regular expression. Mode modifiers override the operator type. E.g. `(?c)regex` forces the to be regex case sensitive.

The most common use of this operator is to select rows based on whether a column matches a regular expression, e.g.:

```
select * from mytable where mycolumn ~* 'regexp'
```

Regular Expressions as Literal PostgreSQL Strings

The backslash is used to escape characters in PostgreSQL strings. So a regular expression like `\w` that contains a backslash becomes `'\\w'` when written as a literal string in a PostgreSQL statement. To match a single literal backslash, you'll need the regex `\\` which becomes `'\\\\'` in PostgreSQL.

PostgreSQL Regexp Functions

With the `substring(string from pattern)` function, you can extract part of a string or column. It takes two parameters: the string you want to extract the text from, and the pattern the extracted text should match. If there is no match, `substring()` returns null. E.g. `substring('subject' from 'regexp')` returns null. If there is a match, and the regex has one or more capturing groups, the text matched by the first capturing group is returned. E.g. `substring('subject' from 's(\w)')` returns 'u'. If there is a match, but the regex has no capturing groups, the whole regex match is returned. E.g. `substring('subject' from 's\w')` returns 'su'. If the regex matches the string more than once, only the first match is returned. Since the `substring()` function doesn't take a "flags" parameter, you'll need to toggle any matching options using mode modifiers.

This function is particularly useful to extract information from columns. E.g. to extract the first number from the column *mycolumn* for each row, use:

```
select substring(mycolumn from '\d+') from mytable
```

With `regexp_replace(subject, pattern, replacement [, flags])` you can replace regex matches in a string. If you omit the *flags* parameter, the regex is applied case sensitively, and only the first match is replaced. If you set the flags to 'i', the regex is applied case insensitively. The 'g' flag (for "global") causes all regex matches in the string to be replaced. You can combine both flags as 'gi'.

You can use the backreferences `\1` through `\9` in the replacement text to re-insert the text matched by a capturing group into the regular expression. `&` re-inserts the whole regex match. Remember to double up the backslashes in literal strings.

E.g. `regexp_replace('subject', '(\w)\w', '\&\1', 'g')` returns 'susbjbecet'.

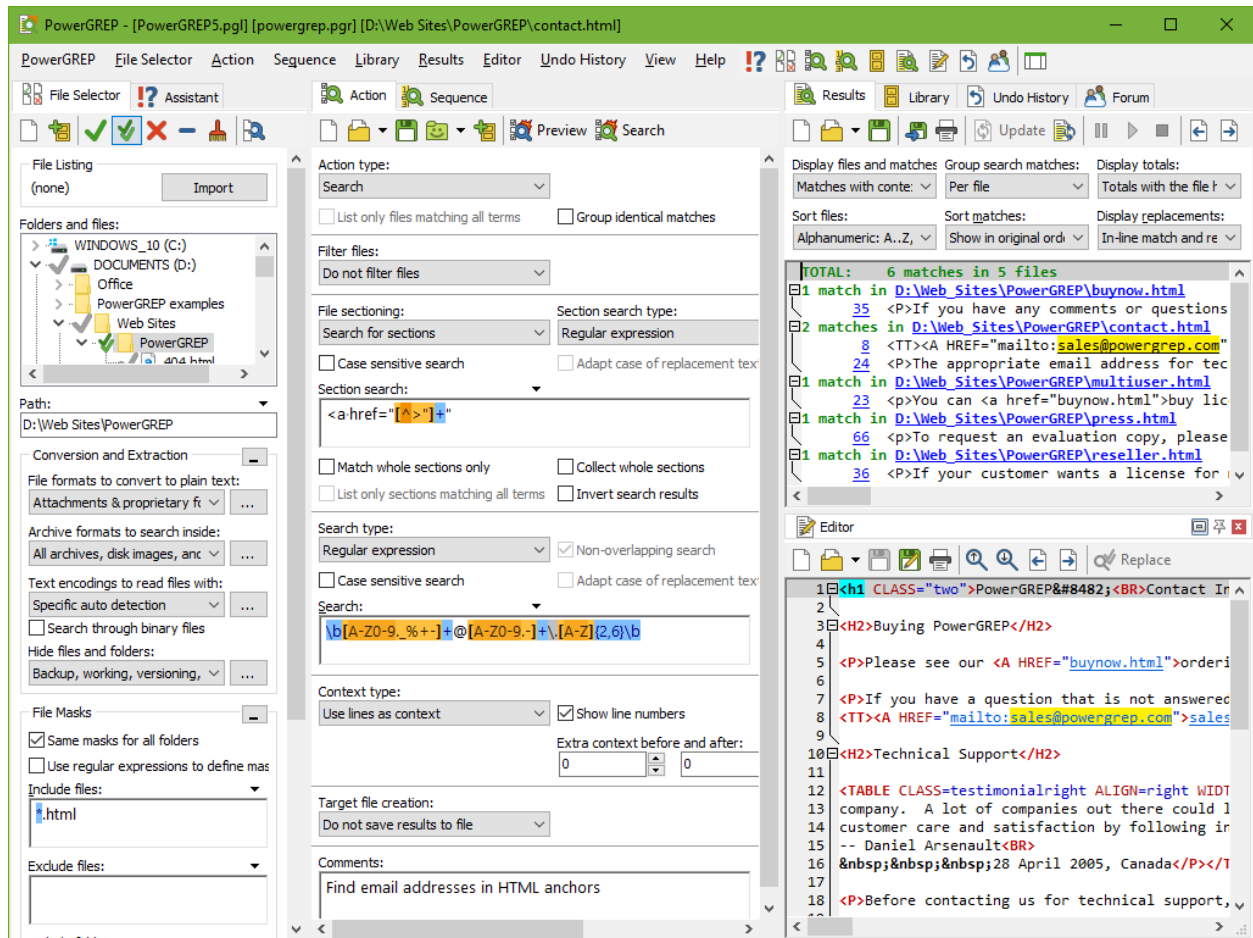
PostgreSQL 8.3 and later have two new functions to split a string along its regex matches. `regexp_split_to_table(subject, pattern [, flags])` returns the split string as a new table. `regexp_split_to_array(subject, pattern [, flags])` returns the split string as an array of text. If the regex finds no matches, both functions return the subject string.

20. PowerGREP: Taking grep Beyond The Command Line

While all of PowerGREP's functionality is also available from the command line, the key benefit of PowerGREP over the traditional grep is its flexible and convenient graphical interface. Instead of just listing the matching lines, PowerGREP will also highlight the actual matches and make them clickable. When you click on a match, PowerGREP will load the file, with syntax coloring, allowing you to easily inspect the context of a match.

PowerGREP also provides a full-featured multi-line text editor box for composing the regular expression you want to use in your search.

PowerGREP's regex flavor is a perfect blend of the Perl 5, PCRE2, Java and .NET regex flavors. If you're already familiar with one of these regex flavors, you'll be immediately at home with PowerGREP. PowerGREP's flavor is indicated as "JGsoft V2" in the tutorial and reference in this book are available in PowerGREP.



The Ultimate Search and Replace

If you already have some experience with regular expressions, then you already know that searching and replacing with regular expressions and backreferences is a powerful way to maintain all sorts of text files. If not, I suggest you download a copy of PowerGREP and take a look at the examples in the help file.

One of the benefits of using PowerGREP for such tasks, is that you can preview the replacements, and inspect the context of the replacements, just like with the search function described above. Replace or revert all matches or all matches in a file after previewing or executing the search-and-replace. Replace or revert individual or selected matches in PowerGREP's full-featured file editor. Naturally, an undo feature is available as well.

Another benefit is PowerGREP's ability to work with lists of regular expressions. You can specify as many search and replace operations as you want, to be executed together, one after the other, on the same files. Saving lists that you use regularly into a PowerGREP action file will save you a lot of time.

Collecting Information and Statistics

PowerGREP's "collect" feature is a unique and useful variation on the traditional regular expression search. Instead of outputting the line on which a match was found, it will output the regex match itself, or a variation of it. This variation is a piece of text you can compose using backreferences, just like the replacement text for a search and replace. You can have the collected matches sorted, and have identical matches grouped together. This way you can compute simple statistics. The "collect" feature is most useful if you want to extract information from log files for which no specialized analysis software exists.

Rename, Copy, Merge and Split Files

PowerGREP can do much more with regular expressions than the traditional search and search-and-replace jobs. Rename or copy files or entire folders by searching and replacing within their file names, folder names, or full paths. You can even compress and decompress files this way by adding or removing a .gz or .bz2 extension, or by changing the path to be inside a .zip or .7z archive, or not. Merge or split the contents of files into new files by searching with a regular expression and using the replacement text to build a path for the target file or files.

File Filtering, File Sectioning, Extra Processing, and Context

Most grep tools can work with only one regular expression at a time. With PowerGREP, you can use up to five lists of any number of regular expressions. One list is the main search, search-and-replace, collect, rename, merge, or split action. The other lists are used for file filtering, file sectioning, extra processing, and context. Use file filtering to skip certain files based on a regex match or lack thereof. Use file sectioning to limit the main action to only certain parts of each file. Use extra processing to apply an extra search-and-replace to each individual search match. Use regexes to match blocks of context to display the results more clearly if your files aren't line-based.

If this sounds complicated, it isn't. You can often use far simpler regular expressions with PowerGREP. Instead of creating a complicated regex to match an email address inside an HTML anchor tag, use a standard

regex matching an email address as the search action, and a standard regex matching an HTML anchor tag for file sectioning.

More Information on PowerGREP and Free Trial Download

PowerGREP works under Windows XP, Vista, 7, 8, 8.1, 10, and 11. For more information on PowerGREP, please visit www.powergrep.com.

21. Regular Expressions with PowerShell

PowerShell is a programming language from Microsoft that is primarily designed for system administration. PowerShell is built on top of .NET so .NET's excellent regular expression support is also available to PowerShell programmers.

The discussion below applies equally to Windows PowerShell 1.0 to 5.1, PowerShell Core 6, and PowerShell 7. There haven't been any changes to the regex syntax in .NET or in .NET Core since .NET 2.0. PowerShell 1.0 was released after .NET 2.0. So all versions of PowerShell use the same regex syntax. Windows PowerShell 2.0 and 5.0 added some features that make it easier to split strings and invoke other `Regex()` constructors. Other than that, there are no differences between any of the PowerShell versions regarding the use of regular expressions.

PowerShell -Match and -Replace Operators

With the `-match` operator, you can quickly check if a regular expression matches part of a string. E.g. `'test' -match '\w'` returns true, because `\w` matches `t` in `test`.

As a side effect, the `-match` operator sets a special variable called `$matches`. This is an associative array that holds the overall regex match and all capturing group matches. `$matches[0]` gives you the overall regex match, `$matches[1]` the first capturing group, and `$matches['name']` the text matched by the named group "name".

The `-replace` operator uses a regular expression to search-and-replace through a string. E.g. `'test' -replace '\w', '$&$&'` returns `tteesstt`. The regex `\w` matches one letter. The replacement text re-inserts the regex match twice using `$&`. The replacement text parameter must be specified, and the regex and replacement must be separated by a comma. If you want to replace the regex matches with nothing, pass an empty string as the replacement.

Traditionally, regular expressions are case sensitive by default. This is true for the .NET framework too. However, it is not true in PowerShell. `-match` and `-replace` are case insensitive, as are `-imatch` and `-ireplace`. For case sensitive matching, use `-cmatch` and `-creplace`. I recommend that you always use the "i" or "c" prefix to avoid confusion regarding case sensitivity.

The operators do not provide a way to pass options from .NET's `RegexOptions` enumeration. Instead, use mode modifiers in the regular expression. E.g. `(?m)^test$` is the same as using `^test$` with `RegexOptions.Multiline` passed to the `Regex()` constructor. Mode modifiers take precedence over options set externally to the regex. `-cmatch '(?i)test'` is case insensitive, while `-imatch '(?-i)test'` is case sensitive. The mode modifier overrides the case insensitivity preference of the `-match` operator.

PowerShell -Split Operator

PowerShell 2.0, introduced with Windows 7 SP1, added the `-split` operator. It allows you to split a string along regex matches. E.g. `$subject -split "\w+"` splits the subject string along non-word characters, thus returning an array of all the words in the string.

Replacement Text as a Literal String

The `-replace` operator supports the same replacement text placeholders as the `Regex.Replace()` function in .NET. `$$` is the overall regex match, `$1` is the text matched by the first capturing group, and `${name}` is the text matched by the named group “name”.

But with PowerShell, there’s an extra caveat: double-quoted strings use the dollar syntax for variable interpolation. Variable interpolation is done before the `Regex.Replace()` function (which `-replace` uses internally) parses the replacement text. Unlike Perl, `$1` is not a magical variable in PowerShell. That syntax only works in the replacement text. The `-replace` operator does not set the `$matches` variable either. The effect is that `'test' -replace '(\w)(\w)', '$2$1'` (double-quoted replacement) returns the empty string (assuming you did not set the variables `$1` and `$2` in preceding PowerShell code). Due to variable interpolation, the `Replace()` function never sees `$2$1`. To allow the `Replace()` function to substitute its placeholders, use `'test' -replace '(\w)(\w)', '$2$1'` (single-quoted replacement) or `'test' -replace '(\w)(\w)', "`$2`$1"` (dollars escaped with backticks) to make sure `$2$1` is passed literally to `Regex.Replace()`.

Using The System.Text.RegularExpressions.Regex Class

To use all of .NET’s regex processing functionality with PowerShell, create a regular expression object by instantiating the `System.Text.RegularExpressions.Regex` class. PowerShell provides a handy shortcut if you want to use the `Regex()` constructor that takes a string with your regular expression as the only parameter. `$regex = [regex] '\W+'` compiles the regular expression `\W+` (which matches one or more non-word characters) and stores the result in the variable `$regex`. You can now call all the methods of the `Regex` class on your `$regex` object.

In PowerShell 5.0 and later you can invoke another `Regex()` constructor on the class name:

```
using namespace System.Text.RegularExpressions
$regex = [Regex]::new('^test$', [RegexOptions]::MultiLine)
```

In older versions of PowerShell, you have to resort to PowerShell’s `new-object` cmdlet. To set the flag `RegexOptions.MultiLine`, for example, you’d need this line of code:

```
$regex = new-object System.Text.RegularExpressions.Regex ('^test$',
[System.Text.RegularExpressions.RegexOptions]::MultiLine)
```

In any version of PowerShell, mode modifiers inside the regex can provide a shorter solution:

```
$regex = [regex] '(?m)^test$'
```

Mode modifiers also work with the `-match`, `-replace`, and `-split` operators.

22. Python's re Module

Python is a high level open source scripting language. Python's built-in "re" module provides excellent support for regular expressions, with a modern and complete regex flavor. Two significant missing features, atomic grouping and possessive quantifiers, were added in Python 3.11. Though Python's regex engine correctly handles Unicode strings, its syntax is still missing Unicode properties and shorthand character classes only match ASCII characters.

The first thing to do is to import the `re` module into your script with `import re`.

Regex Search and Match

Call `re.search(regex, subject)` to apply a regex pattern to a subject string. The function returns `None` if the matching attempt fails, and a `Match` object otherwise. Since `None` evaluates to `False`, you can easily use `re.search()` in an `if` statement. The `Match` object stores details about the part of the string matched by the regular expression pattern.

You can set regex matching modes by specifying a special constant as a third parameter to `re.search()`. `re.I` or `re.IGNORECASE` applies the pattern case insensitively. `re.S` or `re.DOTALL` makes the dot match newlines. `re.M` or `re.MULTILINE` makes the caret and dollar match after and before line breaks in the subject string. There is no difference between the single-letter and descriptive options, except for the number of characters you have to type in. To specify more than one option, "or" them together with the `|` operator: `re.search("^a", "abc", re.I | re.M)`.

By default, Python's regex engine only considers the letters A through Z, the digits 0 through 9, and the underscore as "word characters". Specify the flag `re.L` or `re.LOCALE` to make `\w` match all characters that are considered letters given the current locale settings. Alternatively, you can specify `re.U` or `re.UNICODE` to treat all letters from all scripts as word characters. The setting also affects word boundaries.

Do not confuse `re.search()` with `re.match()`. Both functions do exactly the same, with the important distinction that `re.search()` will attempt the pattern throughout the string, until it finds a match. `re.match()` on the other hand, only attempts the pattern at the very start of the string. Basically, `re.match("regex", subject)` is the same as `re.search("\Aregex", subject)`. Note that `re.match()` does *not* require the regex to match the entire string. `re.match("a", "ab")` will succeed.

Python 3.4 adds a new `re.fullmatch()` function. This function only returns a `Match` object if the regex matches the string entirely. Otherwise it returns `None`. `re.fullmatch("regex", subject)` is the same as `re.search("\Aregex\Z", subject)`. This is useful for validating user input. If `subject` is an empty string then `fullmatch()` evaluates to `True` for any regex that can find a zero-length match.

To get all matches from a string, call `re.findall(regex, subject)`. This will return an array of all non-overlapping regex matches in the string. "Non-overlapping" means that the string is searched through from left to right, and the next match attempt starts beyond the previous match. If the regex contains one or more capturing groups, `re.findall()` returns an array of tuples, with each tuple containing text matched by all the capturing groups. The overall regex match is *not* included in the tuple, unless you place the entire regex inside a capturing group.

More efficient than `re.findall()` is `re.finditer(regex, subject)`. It returns an iterator that enables you to loop over the regex matches in the subject string: `for m in re.finditer(regex, subject)`. The for-loop variable `m` is a `Match` object with the details of the current match.

Unlike `re.search()` and `re.match()`, `re.findall()` and `re.finditer()` do not support an optional third parameter with regex matching flags. Instead, you can use global mode modifiers at the start of the regex. E.g. “`(?i)regex`” matches `regex` case insensitively.

Strings, Backslashes and Regular Expressions

The backslash is a metacharacter in regular expressions. It is used to escape other metacharacters. The regex `\\` matches a single backslash. `\d` is a single token matching a digit.

Python strings also use the backslash to escape characters. The above regexes are written as Python strings as `“\\”` and `“\d”`. Confusing indeed.

Fortunately, Python also has “raw strings” which do not apply special treatment to backslashes. As raw strings, the above regexes become `r“\\”` and `r“\d”`. The only limitation of using raw strings is that the delimiter you’re using for the string must not appear in the regular expression, as raw strings do not offer a means to escape it.

You can use `\n` and `\t` in raw strings. Though raw strings do not support these escapes, the regular expression engine does. The end result is the same.

Unicode

Prior to Python 3.3, Python’s `re` module did not support any Unicode regular expression tokens. Python Unicode strings, however, have always supported the `\uFFFF` notation. Python’s `re` module can use Unicode strings. So you could pass the Unicode string `u“\u00E0\d”` to the `re` module to match `à` followed by a digit. The backslash for `\d` was escaped, while the one for `\u` was not. That’s because `\d` is a regular expression token, and a regular expression backslash needs to be escaped. `\u00E0` is a Python string token that shouldn’t be escaped. The string `u“\u00E0\d”` is seen by the regular expression engine as `à\d`.

If you did put another backslash in front of the `\u`, the regex engine would see `\u00E0\d`. If you use this regex with Python 3.2 or earlier, it will match the literal text `u00E0` followed by a digit instead.

To avoid any confusion about whether backslashes need to be escaped, just use Unicode raw strings like `ur“\u00E0\d”`. Then backslashes don’t need to be escaped. Python does interpret Unicode escapes in raw strings.

In Python 3.0 and later, strings are Unicode by default. So the `u` prefix shown in the above samples is no longer necessary. Python 3.3 also adds support for the `\uFFFF` notation to the regular expression engine. So in Python 3.3, you can use the string `“\u00E0\d”` to pass the regex `\u00E0\d` which will match something like `à0`.

Search and Replace

`re.sub(regex, replacement, subject)` performs a search-and-replace across `subject`, replacing all matches of `regex` in `subject` with `replacement`. The result is returned by the `sub()` function. The `subject` string you pass is not modified.

If the `regex` has capturing groups, you can use the text matched by the part of the `regex` inside the capturing group. To substitute the text from the third group, insert `\3` into the replacement string. If you want to use the text of the third group followed by a literal three as the replacement, use `\g<3>3`. `\33` is interpreted as the 33rd group. It is an error if there are fewer than 33 groups. If you used named capturing groups, you can use them in the replacement text with `\g<name>`.

The `re.sub()` function applies the same backslash logic to the replacement text as is applied to the regular expression. Therefore, you should use raw strings for the replacement text, as I did in the examples above. The `re.sub()` function will also interpret `\n` and `\t` in raw strings. If you want `c:\temp` as a replacement, either use `r"c:\\temp"` or `"c:\\\\temp"`. The 3rd backreference is `r"\3"` or `"\3"`.

Splitting Strings

`re.split(regex, subject)` returns an array of strings. The array contains the parts of `subject` between all the `regex` matches in the `subject`. Adjacent `regex` matches will cause empty strings to appear in the array. The `regex` matches themselves are not included in the array. If the `regex` contains capturing groups, then the text matched by the capturing groups is included in the array. The capturing groups are inserted between the substrings that appeared to the left and right of the `regex` match. If you don't want the capturing groups in the array, convert them into non-capturing groups. The `re.split()` function does not offer an option to suppress capturing groups.

You can specify an optional third parameter to limit the number of times the `subject` string is split. Note that this limit controls the number of splits, not the number of strings that will end up in the array. The unsplit remainder of the `subject` is added as the final string to the array. If there are no capturing groups, the array will contain `limit+1` items.

The behavior of `re.split()` has changed between Python versions when the regular expression can find zero-length matches. In Python 3.4 and prior, `re.split()` ignores zero-length matches. In Python 3.5 and 3.6 `re.split()` throws a `FutureWarning` when it encounters a zero-length match. This warning signals the change in Python 3.7. Now `re.split()` also splits on zero-length matches.

Match Details

`re.search()` and `re.match()` return a `Match` object, while `re.finditer()` generates an iterator to iterate over a `Match` object. This object holds lots of useful information about the `regex` match. I will use `m` to signify a `Match` object in the discussion below.

`m.group()` returns the part of the string matched by the entire regular expression. `m.start()` returns the offset in the string of the start of the match. `m.end()` returns the offset of the character beyond the match. `m.span()` returns a 2-tuple of `m.start()` and `m.end()`. You can use the `m.start()` and `m.end()` to slice the `subject` string: `subject[m.start():m.end()]`.

If you want the results of a capturing group rather than the overall regex match, specify the name or number of the group as a parameter. `m.group(3)` returns the text matched by the third capturing group. `m.group('groupname')` returns the text matched by a named group 'groupname'. If the group did not participate in the overall match, `m.group()` returns an empty string, while `m.start()` and `m.end()` return -1.

If you want to do a regular expression based search-and-replace without using `re.sub()`, call `m.expand(replacement)` to compute the replacement text. The function returns the replacement string with backreferences etc. substituted.

Regular Expression Objects

If you want to use the same regular expression more than once, you should compile it into a regular expression object. Regular expression objects are more efficient, and make your code more readable. To create one, just call `re.compile(regex)` or `re.compile(regex, flags)`. The flags are the matching options described above for the `re.search()` and `re.match()` functions.

The regular expression object returned by `re.compile()` provides all the functions that the `re` module also provides directly: `search()`, `match()`, `findall()`, `finditer()`, `sub()` and `split()`. The difference is that they use the pattern stored in the regex object, and do not take the regex as the first parameter. `re.compile(regex).search(subject)` is equivalent to `re.search(regex, subject)`.

23. Regular Expressions with The R Language

The R Project for Statistical Computing provides seven regular expression functions in its `base` package. The R documentation claims that the default flavor implements POSIX extended regular expressions. That is not correct. In R 2.10.0 and later, the default regex engine is a modified version of Ville Laurikari's TRE engine. It mimics POSIX but deviates from the standard in many subtle and not-so-subtle ways. What this book says about POSIX ERE does not (necessarily) apply to R.

Older versions of R used the GNU library to implement both POSIX BRE and ERE. ERE was the default. Passing the `extended=FALSE` parameter allowed you to switch to BRE. This parameter was deprecated in R 2.10.0 and removed in R 2.11.0.

The best way to use regular expressions with R is to pass the `perl=TRUE` parameter. This tells R to use the PCRE regular expressions library. When this book talks about R, it assumes you're using the `perl=TRUE` parameter. Starting with R 4.0.0, passing `perl=TRUE` makes R use the PCRE2 library.

All the functions use case sensitive matching by default. You can pass `ignore.case=TRUE` to make them case insensitive. R's functions do not have any parameters to set any other matching modes. When using `perl=TRUE`, as you should, you can add mode modifiers to the start of the regex.

Finding Regex Matches in String Vectors

The `grep` function takes your regex as the first argument, and the input vector as the second argument. If you pass `value=FALSE` or omit the `value` parameter then `grep` returns a new vector with the indexes of the elements in the input vector that could be (partially) matched by the regular expression. If you pass `value=TRUE`, then `grep` returns a vector with copies of the actual elements in the input vector that could be (partially) matched.

```
> grep("a+", c("abc", "def", "cba a", "aa"), perl=TRUE, value=FALSE)
[1] 1      3      4
> grep("a+", c("abc", "def", "cba a", "aa"), perl=TRUE, value=TRUE)
[1] "abc" "cba a" "aa"
```

The `grepl` function takes the same arguments as the `grep` function, except for the `value` argument, which is not supported. `grepl` returns a logical vector with the same length as the input vector. Each element in the returned vector indicates whether the regex could find a match in the corresponding string element in the input vector.

```
> grepl("a+", c("abc", "def", "cba a", "aa"), perl=TRUE)
[1] TRUE FALSE TRUE TRUE
```

The `regexpr` function takes the same arguments as `grepl`. `regexpr` returns an integer vector with the same length as the input vector. Each element in the returned vector indicates the character position in each corresponding string element in the input vector at which the (first) regex match was found. A match at the start of the string is indicated with character position 1. If the regex could not find a match in a certain string, its corresponding element in the result vector is -1. The returned vector also has a `match.length` attribute. This is another integer vector with the number of characters in the (first) regex match in each string, or -1 for strings that didn't match.

`gregexpr` is the same as `regexpr`, except that it finds all matches in each string. It returns a vector with the same length as the input vector. Each element is another vector, with one element for each match found in the string indicating the character position at which that match was found. Each vector element in the returned vector also has a `match.length` attribute with the lengths of all matches. If no matches could be found in a particular string, the element in the returned vector is still a vector, but with just one element `-1`.

```
> regexpr("a+", c("abc", "def", "cba a", "aa"), perl=TRUE)
[1] 1 -1 3 1
attr(,"match.length")
[1] 1 -1 1 2
> gregexpr("a+", c("abc", "def", "cba a", "aa"), perl=TRUE)
[[1]] [1] 1 attr(,"match.length") [1] 1
[[2]] [1] -1 attr(,"match.length") [1] -1
[[3]] [1] 3 5 attr(,"match.length") [1] 1 1
[[4]] [1] 1 attr(,"match.length") [1] 2
```

Use `regmatches` to get the actual substrings matched by the regular expression. As the first argument, pass the same input that you passed to `regexpr` or `gregexpr`. As the second argument, pass the vector returned by `regexpr` or `gregexpr`. If you pass the vector from `regexpr` then `regmatches` returns a character vector with all the strings that were matched. This vector may be shorter than the input vector if no match was found in some of the elements. If you pass the vector from `gregexpr` then `regmatches` returns a vector with the same number of elements as the input vector. Each element is a character vector with all the matches of the corresponding element in the input vector, or `NULL` if an element had no matches.

```
>x <- c("abc", "def", "cba a", "aa")
> m <- regexpr("a+", x, perl=TRUE)
> regmatches(x, m)
[1] "a" "a" "aa"
> m <- gregexpr("a+", x, perl=TRUE)
> regmatches(x, m)
[[1]] [1] "a"
[[2]] character(0)
[[3]] [1] "a" "a"
[[4]] [1] "aa"
```

Replacing Regex Matches in String Vectors

The `sub` function has three required parameters: a string with the regular expression, a string with the replacement text, and the input vector. `sub` returns a new vector with the same length as the input vector. If a regex match could be found in a string element, it is replaced with the replacement text. Only the first match in each string element is replaced. If no matches could be found in some strings, those are copied into the result vector unchanged.

Use `gsub` instead of `sub` to replace all regex matches in all the string elements in your vector. Other than replacing all matches, `gsub` works in exactly the same way, and takes exactly the same arguments.

R uses its own replacement string syntax. Even though R 4.0.0 uses the PCRE2 regex flavor when you pass `perl=TRUE`, it still uses the R replacement string syntax. There is no option to use the PCRE2 replacement string syntax.

You can use the backreferences `\1` through `\9` in the replacement text to reinsert text matched by a capturing group. You cannot use backreferences to groups 10 and beyond. If your regex has named groups, you can use

numbered backreferences to the first 9 groups. There is no replacement text token for the overall match. Place the entire regex in a capturing group and then use `\1` to insert the whole regex match.

```
> sub("(a+)", "z\\1z", c("abc", "def", "cba a", "aa"), perl=TRUE)
[1] "zazbc" "def" "cbzaz a" "zaaz"
> gsub("(a+)", "z\\1z", c("abc", "def", "cba a", "aa"), perl=TRUE)
[1] "zazbc" "def" "cbzaz zaz" "zaaz"
```

You can use `\U` and `\L` to change the text inserted by all following backreferences to uppercase or lowercase. You can use `\E` to insert the following backreferences without any change of case. These escapes do not affect literal text.

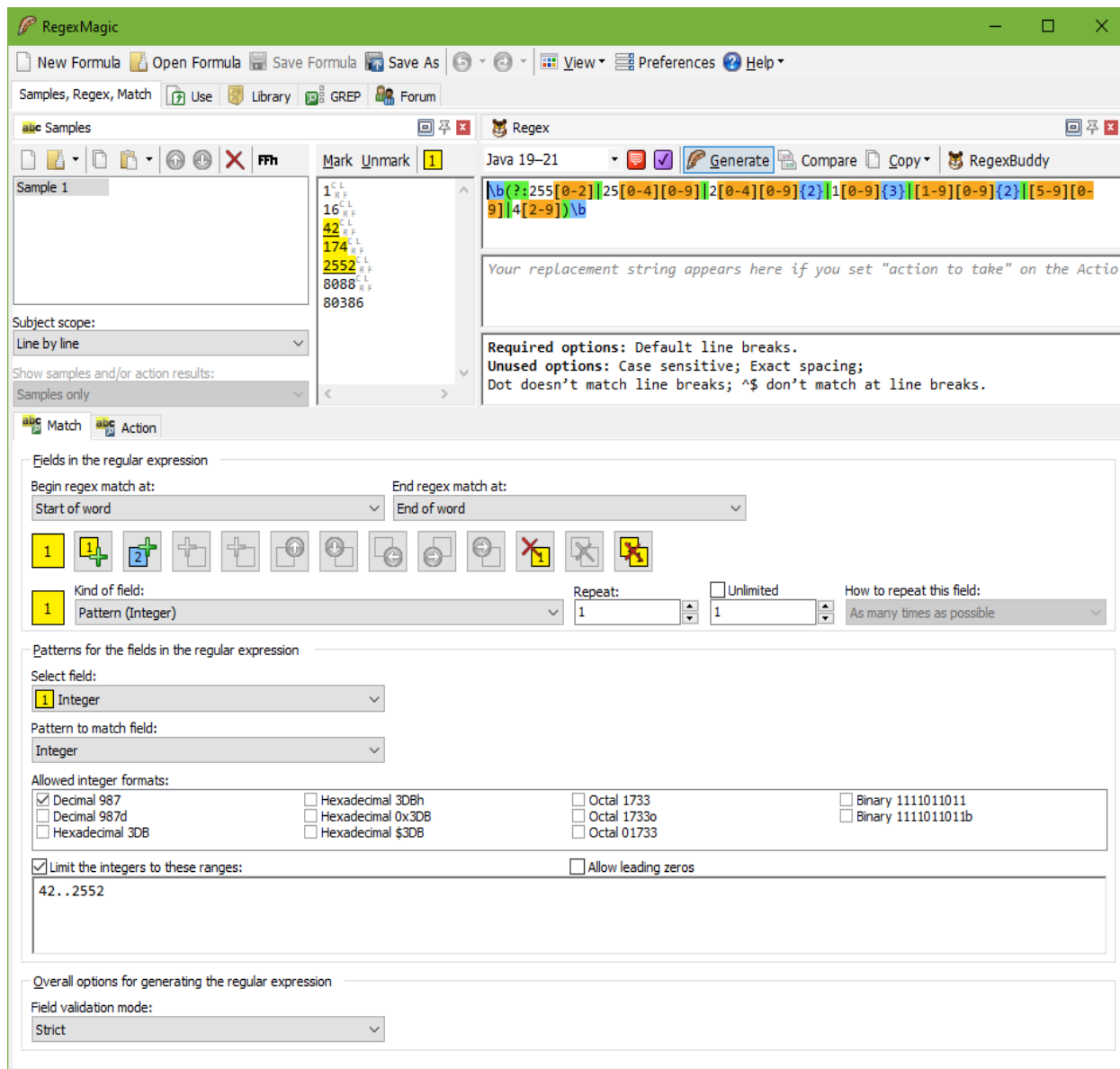
```
> sub("(a+)", "z\\U\\1z", c("abc", "def", "cba a", "aa"), perl=TRUE)
[1] "zAzbc" "def" "cbzAz a" "zAAz"
> gsub("(a+)", "z\\U\\1z", c("abc", "def", "cba a", "aa"), perl=TRUE)
[1] "zAzbc" "def" "cbzAz zAz" "zAAz"
```

A very powerful way of making replacements is to assign a new vector to the `regmatches` function when you call it on the result of `gregexpr`. The vector you assign should have as many elements as the original input vector. Each element should be a character vector with as many strings as there are matches in that element. The original input vector is then modified to have all the regex matches replaced with the text from the new vector.

```
> x <- c("abc", "def", "cba a", "aa")
> m <- gregexpr("a+", x, perl=TRUE)
> regmatches(x, m) <- list(c("one"), character(0), c("two", "three"), c("four"))
> x
[1] "onebc" "def" "cbtwo three" "four"
```

24. RegexMagic: Regular Expression Generator

If the detailed regular expressions tutorial in this book makes your head spin, RegexMagic may be just the tool for you.



RegexMagic makes creating regular expressions easier than ever. While other regex tools such as RegxBuddy merely make it easier to work with regular expressions, with RegexMagic you don't have to deal with the regular expression syntax at all. RegexMagic generates complete regular expressions to your specifications.

First, you provide RegexMagic with some samples of the text you want your regular expression to match. RegexMagic can automatically detect what sort of pattern your text looks like. Numbers, dates, and email addresses are just a few examples of the wide range of patterns that RegexMagic supports. By marking

different parts of your samples, you can create regular expressions that combine multiple patterns to match exactly what you want. RegexMagic's patterns provide many options, so you can make your regular expression as loose or as strict as you want.

Best of all, RegexMagic supports nearly all popular regular expression flavors. Select your flavor, and RegexMagic makes sure to generate a regular expression that works with it. RegexMagic can even generate snippets in many programming languages that you can copy and paste directly into your source code to implement your regular expression.

Find out More and Get Your Own Copy of RegexMagic

RegexMagic works under Windows XP, Vista, 7, 8, 8.1, 10, and 11. For more information on RegexMagic, please visit www.regexmagic.com. You will be creating your own regular expressions in no time.

25. Using Regular Expressions with Ruby

Ruby supports regular expressions as a language feature. In Ruby, a regular expression is written in the form of `/pattern/modifiers` where “pattern” is the regular expression itself, and “modifiers” are a series of characters indicating various options. The “modifiers” part is optional. This syntax is borrowed from Perl. Ruby supports the following modifiers:

- `/i` makes the regex match case insensitive.
- `/m` makes the dot match newlines. Ruby indeed uses `/m`, whereas Perl and many other programming languages use `/s` for “dot matches newlines”.
- `/x` tells Ruby to ignore whitespace between regex tokens.
- `/o` causes any `#{...}` substitutions in a particular regex literal to be performed just once, the first time it is evaluated. Otherwise, the substitutions will be performed every time the literal generates a Regexp object.

You can combine multiple modifiers by stringing them together as in `/regex/is`.

In Ruby, the caret and dollar always match before and after newlines. Ruby does not have a modifier to change this. Use `\A` and `\Z` to match at the start or the end of the string.

Since forward slashes delimit the regular expression, any forward slashes that appear in the regex need to be escaped. E.g. the regex `1/2` is written as `/1\/2/` in Ruby.

How To Use The Regexp Object

`/regex/` creates a new object of the class `Regexp`. You can assign it to a variable to repeatedly use the same regular expression, or use the literal regex directly. Ruby provides several different ways to test whether a particular regexp matches (part of) a string.

The `===` method allows you to compare a regexp to a string. It returns true if the regexp matches (part of) the string or false if it does not. This allows regular expressions to be used in case statements. Do not confuse `===` (3 equals signs) with `==` (2 equals signs). `==` allows you to compare one regexp to another regexp to see if the two regexes are identical and use the same matching modes.

The `=~` method returns the character position in the string of the start of the match or nil if no match was found. In a boolean test, the character position evaluates to true and nil evaluates to false. So you can use `=~` instead of `===` to make your code a little more easier to read as `=~` is more obviously a regex matching operator. Ruby borrowed the `=~` syntax from Perl. `print(/\w+/ =~ "test")` prints “0”. The first character in the string has index zero. Switching the order of the `=~` operator’s operands makes no difference.

The `match()` method returns a `MatchData` object when a match is found, or nil if no matches was found. In a boolean context, the `MatchData` object evaluates to true. In a string context, the `MatchData` object evaluates to the text that was matched. So `print(/\w+/.match("test"))` prints “test”.

Ruby 2.4 adds the `match?()` method. It returns true or false like the `===` method. The difference is that `match?()` does not set `$~` (see below) and thus doesn’t need to create a `MatchData` object. If you don’t need any match details you should use `match?()` to improve performance.

Special Variables

The `===`, `=~`, and `match()` methods create a `MatchData` object and assign it to the special variable `$~`. `Regexp.match()` also returns this object. The variable `$~` is thread-local and method-local. That means you can use this variable until your method exits, or until the next time you use the `=~` operator in your method, without worrying that another thread or another method in your thread will overwrite them.

A number of other special variables are derived from the `$~` variable. All of these are read-only. If you assign a new `MatchData` instance to `$~`, all of these variables will change too. `$&` holds the text matched by the whole regular expression. `$1`, `$2`, etc. hold the text matched by the first, second, and following capturing groups. `$+` holds the text matched by the highest-numbered capturing group that actually participated in the match. `$`` and `$'` hold the text in the subject string to the left and to the right of the regex match.

Search And Replace

Use the `sub()` and `gsub()` methods of the `String` class to search-and-replace the first regex match, or all regex matches, respectively, in the string. Specify the regular expression you want to search for as the first parameter, and the replacement string as the second parameter, e.g.: `result = subject.gsub(/before/, "after")`.

To re-insert the regex match, use `\0` in the replacement string. You can use the contents of capturing groups in the replacement string with backreferences `\1`, `\2`, `\3`, etc. Note that numbers escaped with a backslash are treated as octal escapes in double-quoted strings. Octal escapes are processed at the language level, before the `sub()` function sees the parameter. To prevent this, you need to escape the backslashes in double-quoted strings. So to use the first backreference as the replacement string, either pass `'\1'` or `"\\1"`. `'\\1'` also works.

Splitting Strings and Collecting Matches

To collect all regex matches in a string into an array, pass the `regexp` object to the string's `scan()` method, e.g.: `myarray = mystring.scan(/regex/)`. Sometimes, it is easier to create a regex to match the delimiters rather than the text you are interested in. In that case, use the `split()` method instead, e.g.: `myarray = mystring.split(/delimiter/)`. The `split()` method discards all regex matches, returning the text between the matches. The `scan()` method does the opposite.

If your regular expression contains capturing groups, `scan()` returns an array of arrays. Each element in the overall array will contain an array consisting of the overall regex match, plus the text matched by all capturing groups.

26. C++ Regular Expressions with `std::regex`

The C++ standard library as defined in the C++11 standard provides support for regular expressions in the `<regex>` header. Prior to C++11, `<regex>` was part of the TR1 extension to the C++ standard library. When this book mentions `std::regex`, this refers to the Dinkumware implementation of the C++ standard library that is included with Visual C++ 2008 and later. It is also supported by C++Builder XE3 and later when targeting Win64. In Visual C++ 2008, the namespace is `std::tr1::regex` rather than `std::regex`.

C++Builder 10 and later support the Dinkumware implementation `std::regex` when targeting Win32 if you disable the option to use the classic Borland compiler. When using the classic Borland compiler in C++Builder XE3 and later, you can use `boost::regex` instead of `std::regex`. While `std::regex` as defined in TR1 and C++11 defines pretty much the same operations and classes as `boost::regex`, there are a number of important differences in the actual regex flavor. Most importantly the ECMAScript regex syntax in Boost adds a number of features borrowed from Perl that aren't part of the ECMAScript standard and that aren't implemented in the Dinkumware library.

Six Regular Expression Flavors

Six different regular expression flavors or grammars are defined in `std::regex_constants`:

- `ECMAScript`: Similar to JavaScript
- `basic`: Similar to POSIX BRE.
- `extended`: Similar to POSIX ERE.
- `grep`: Same as `basic`, with the addition of treating line feeds as alternation operators.
- `egrep`: Same as `extended`, with the addition of treating line feeds as alternation operators.
- `awk`: Same as `extended`, with the addition of supporting common escapes for non-printable characters.

Most C++ references talk as if C++11 implements regular expressions as defined in the ECMA-262v3 and POSIX standards. But in reality the C++ implementation is very loosely based these standards. The syntax is quite close. The only significant differences are that `std::regex` supports POSIX classes even in ECMAScript mode, and that it is a bit peculiar about which characters must be escaped (like curly braces and closing square brackets) and which must not be escaped (like letters).

But there are important differences in the actual behavior of this syntax. The caret and dollar always match at embedded line breaks in `std::regex`, while in JavaScript and POSIX this is an option. Backreferences to non-participating groups fail to match as in most regex flavors, while in JavaScript they find a zero-length match. In JavaScript, `\d` and `\w` are ASCII-only while `\s` matches all Unicode whitespace. This is odd, but all modern browsers follow the spec. In `std::regex` all the shorthands are ASCII-only when using strings of `char`. In Visual C++, but not in C++Builder, they support Unicode when using strings of `wchar_t`. The POSIX classes also match non-ASCII characters when using `wchar_t` in Visual C++, but do not consistently include all the Unicode characters that one would expect.

In practice, you'll mostly use the ECMAScript grammar. It's the default grammar and offers far more features than the other grammars. Whenever the tutorial in this book mentions `std::regex` without mentioning any grammars then what is written applies to the ECMAScript grammar and may or may not apply to any of the other grammars. You'll really only use the other grammars if you want to reuse existing regular expressions from old POSIX code or UNIX scripts.

Creating a Regular Expression Object

Before you can use a regular expression, you have to create an object of the template class `std::basic_regex`. You can easily do this with the `std::regex` instantiation of this template class if your subject is an array of `char` or an `std::string` object. Use the `std::wregex` instantiation if your subject is an array of `wchar_t` or an `std::wstring` object.

Pass your regex as a string as the first parameter to the constructor. If you want to use a regex flavor other than ECMAScript, pass the appropriate constant as a second parameter. You can “or” this constant with `std::regex_constants::icase` to make the regex case insensitive. You can also “or” it with `std::regex_constants::nosubs` to turn all capturing groups into non-capturing groups, which makes your regex more efficient if you only care about the overall regex match and don’t want to extract text matched by any of the capturing groups.

Finding a Regex Match

Call `std::regex_search()` with your subject string as the first parameter and the regex object as the second parameter to check whether your regex can match any part of the string. Call `std::regex_match()` with the same parameters if you want to check whether your regex can match the *entire* subject string. Since `std::regex` lacks anchors that exclusively match at the start and end of the string, you have to call `regex_match()` when using a regex to validate user input.

Both `regex_search()` and `regex_match()` return just true or false. To get the part of the string matched by `regex_search()`, or to get the parts of the string matched by capturing groups when using either function, you need to pass an object of the template class `std::match_results` as the second parameter. The regex object then becomes the third parameter. Create this object using the default constructor of one of these four template instantiations:

- `std::cmatch` when your subject is an array of `char`
- `std::smatch` when your subject is an `std::string` object
- `std::wcmatch` when your subject is an array of `wchar_t`
- `std::wsmatch` when your subject is an `std::wstring` object

When the function call returns true, you can call the `str()`, `position()`, and `length()` member functions of the `match_results` object to get the text that was matched, or the starting position and its length of the match relative to the subject string. Call these member functions without a parameter or with 0 as the parameter to get the overall regex match. Call them passing 1 or greater to get the match of a particular capturing group. The `size()` member function indicates the number of capturing groups plus one for the overall match. Thus you can pass a value up to `size()-1` to the other three member functions.

Putting it all together, we can get the text matched by the first capturing group like this:

```
std::string subject("Name: John Doe");
std::string result;
try {
    std::regex re("Name: (.*)");
    std::smatch match;
    if (std::regex_search(subject, match, re) && match.size() > 1) {
        result = match.str(1);
    } else {
```

```

    result = std::string("");
}
} catch (std::regex_error& e) {
    // Syntax error in the regular expression
}

```

Finding All Regex Matches

To find all regex matches in a string, you need to use an iterator. Construct an object of the template class `std::regex_iterator` using one of these four template instantiations:

- `std::cregex_iterator` when your subject is an array of `char`
- `std::sregex_iterator` when your subject is an `std::string` object
- `std::wcregex_iterator` when your subject is an array of `wchar_t`
- `std::wsregex_iterator` when your subject is an `std::wstring` object

Construct one object by calling the constructor with three parameters: a string iterator indicating the starting position of the search, a string iterator indicating the ending position of the search, and the regex object. If there are any matches to be found, the object will hold the first match when constructed. Construct another iterator object using the default constructor to get an end-of-sequence iterator. You can compare the first object to the second to determine whether there are any further matches. As long as the first object is not equal to the second, you can dereference the first object to get a `match_results` object.

```

std::string subject("This is a test");
try {
    std::regex re("\\w+");
    std::sregex_iterator next(subject.begin(), subject.end(), re);
    std::sregex_iterator end;
    while (next != end) {
        std::smatch match = *next;
        std::cout << match.str() << "\n";
        next++;
    }
} catch (std::regex_error& e) {
    // Syntax error in the regular expression
}

```

Replacing All Matches

To replace all matches in a string, call `std::regex_replace()` with your subject string as the first parameter, the regex object as the second parameter, and the string with the replacement text as the third parameter. The function returns a new string with the replacements applied.

The replacement string syntax is similar but not identical to that of JavaScript. The same replacement string syntax is used regardless of which regex syntax or grammar you are using. You can use `$&` or `$0` to insert the whole regex match and `$1` through `$9` to insert the text matched by the first nine capturing groups. There is no way to insert the text matched by groups 10 or higher. `$10` and higher are always replaced with nothing, and `$9` and lower are replaced with nothing if there are fewer capturing groups in the regex than the requested number. `$`` (dollar backtick) is the part of the string to the left of the match, and `$'` (dollar quote) is the part of the string to the right of the match.

27. Tcl Has Three Regular Expression Flavors

Tcl 8.2 and later support three regular expression flavors. The Tcl man pages dub them Basic Regular Expressions (BRE), Extended Regular Expressions (ERE) and Advanced Regular Expressions (ARE). BRE and ERE are mainly for backward compatibility with previous versions of Tcl. These flavors implement the two flavors defined in the POSIX standard. AREs are new in Tcl 8.2. They're the default and recommended flavor. This flavor implements the POSIX ERE flavor, with a whole bunch of added features. Most of these features are inspired by similar features in Perl regular expressions.

Tcl's regular expression support is based on a library developed for Tcl by Henry Spencer. This library has since been used in a number of other programming languages and applications, such as the PostgreSQL database and the wxWidgets GUI library for C++. Everything said about Tcl in this regular expressions tutorial applies to any tool that uses Henry Spencer's Advanced Regular Expressions.

There are a number of important differences between Tcl Advanced Regular Expressions and Perl-style regular expressions. Tcl uses `\m`, `\M`, `\y` and `\Y` for word boundaries. Perl and most other modern regex flavors use `\b` and `\B`. In Tcl, these last two match a backspace and a backslash, respectively.

Tcl also takes a completely different approach to mode modifiers. The `(?letters)` syntax is the same, but the available mode letters and their meanings are quite different. Instead of adding mode modifiers to the regular expression, you can pass more descriptive switches like `-nocase` to the `regexp` and `regsub` commands for some of the modes. Mode modifier spans in the style of `(?modes:regex)` are not supported. Mode modifiers must appear at the start of the regex. They affect the whole regex. Mode modifiers in the `regex` override command switches. Tcl supports these modes:

- `(?i)` or `-nocase` makes the regex match case insensitive.
- `(?c)` makes the regex match case sensitive. This mode is the default.
- `(?x)` or `-expanded` activates the free-spacing regex syntax.
- `(?t)` disables the free-spacing regex syntax. This mode is the default. The “t” stands for “tight”, the opposite of “expanded”.
- `(?b)` tells Tcl to interpret the remainder of the regular expression as a Basic Regular Expression.
- `(?e)` tells Tcl to interpret the remainder of the regular expression as an Extended Regular Expression.
- `(?q)` tells Tcl to interpret the remainder of the regular expression as plain text. The “q” stands for “quoted”.
- `(?s)` selects “non-newline-sensitive matching”, which is the default. The “s” stands for “single line”. In this mode, the dot and negated character classes match all characters, including newlines. The caret and dollar match only at the very start and end of the subject string.
- `(?p)` or `-linestop` enables “partial newline-sensitive matching”. In this mode, the dot and negated character classes do not match newlines. The caret and dollar match only at the very start and end of the subject string.
- `(?w)` or `-lineanchor` enables “inverse partial newline-sensitive matching”. The “w” stands for “weird”. (Don't look at me! I didn't come up with this.) In this mode, the dot and negated character classes match all characters, including newlines. The caret and dollar match after and before newlines.
- `(?n)` or `-line` enables what Tcl calls “newline-sensitive matching”. The dot and negated character classes do not match newlines. The caret and dollar match after and before newlines. Specifying `(?n)` or `-line` is the same as specifying `(?pw)` or `-linestop -lineanchor`.
- `(?m)` is a historical synonym for `(?n)`.

If you use regular expressions with Tcl and other programming languages, be careful when dealing with the newline-related matching modes. Tcl's designers found Perl's `/m` and `/s` modes confusing. They are confusing, but at least Perl has only two, and they both affect only one thing. In Perl, `/m` or `(?m)` enables “multi-line mode”, which makes the caret and dollar match after and before newlines. By default, they match at the very start and end of the string only. In Perl, `/s` or `(?s)` enables “single line mode”. This mode makes the dot match all characters, including line break. By default, it doesn't match line breaks. Perl does not have a mode modifier to exclude line breaks from negated character classes. In Perl, `[^a]` matches anything except `a`, including newlines. The only way to exclude newlines is to write `[^a\n]`. Perl's default matching mode is like Tcl's `(?p)`, except for the difference in negated character classes.

Why compare Tcl with Perl? Many popular regex flavors such as .NET, Java, PCRE and Python support the same `(?m)` and `(?s)` modifiers with the exact same defaults and effects as in Perl. Negated character classes work the same in all these languages and libraries. It's unfortunate that Tcl didn't follow Perl's standard, since Tcl's four options are just as confusing as Perl's two options. Together they make a very nice alphabet soup.

If you ignore the fact that Tcl's options affect negated character classes, you can use the following table to translate between Tcl's newline modes and Perl-style newline modes. Note that the defaults are different. If you don't use any switches, `(?s)` and `.` are equivalent in Tcl, but not in Perl.

Tcl	Perl	anchors	Dot
<code>(?s)</code> (default)	<code>(?s)</code>	Start and end of string only	Any character
<code>(?p)</code>	(default)	Start and end of string only	Any character except newlines
<code>(?w)</code>	<code>(?sm)</code>	Start and end of string, and at newlines	Any character
<code>(?n)</code>	<code>(?m)</code>	Start and end of string, and at newlines	Any character except newlines

Regular Expressions as Tcl Words

You can insert regular expressions in your Tcl source code either by enclosing them with double quotes (e.g. `"my regexp"`) or by enclosing them with curly braces (e.g. `{my regexp}`). Since the braces don't do any substitution like the quotes, they're by far the best choice for regular expressions.

The only thing you need to worry about is that unescaped braces in the regular expression must be balanced. Escaped braces don't need to be balanced, but the backslash used to escape the brace remains part of the regular expression. You can easily satisfy these requirements by escaping all braces in your regular expression, except those used as a quantifier. This way your regex will work as expected, and you don't need to change it at all when pasting it into your Tcl source code, other than putting a pair of braces around it.

The regular expression `^\{ \d{3} \}` matches a string that consists entirely of an opening brace, three digits and one backslash. In Tcl, this becomes `{^\{ \d{3} \}}`. There's no doubling of backslashes or any sort of escaping needed, as long as you escape literal braces in the regular expression. `{` and `\{` are both valid regular expressions to match a single opening brace in a Tcl ARE (and any Perl-style regex flavor, for that matter). Only the latter works correctly in a Tcl literal enclosed with braces.

Finding Regex Matches

In Tcl, you can use the `regexp` command to test if a regular expression matches (part of) a string, and to retrieve the matched part(s). The syntax of the command is:

```
regexp ?switches? regexp subject ?matchvar? ?group1var group2var ...?
```

Immediately after the `regexp` command, you can place zero or more switches from the list above to indicate how Tcl should apply the regular expression. The only required parameters are the regular expression and the subject string. You can specify a literal regular expression using braces as I just explained. Or, you can reference any string variable holding a regular expression read from a file or user input.

If you pass the name of a variable as an additional argument, Tcl stores the part of the string matched by the regular expression into that variable. Tcl does *not* set the variable to an empty string if the match attempt fails. If the regular expressions has capturing groups, you can add additional variable names to capture the text matched by each group. If you specify fewer variables than the regex has capturing groups, the text matched by the additional groups is not stored. If you specify more variables than the regex has capturing groups, the additional variables are set to an empty string if the overall regex match was successful.

The `regexp` command returns 1 if (part of) the string could be matched, and zero if there's no match. The following script applies the regular expression `my regex` case insensitively to the string stored in the variable `subjectstring` and displays the result:

```
if [  
  regexp -nocase {my regex} $subjectstring matchresult  
] then {  
  puts $matchresult  
} else {  
  puts "my regex could not match the subject string"  
}
```

The `regexp` command supports three more switches that aren't regex mode modifiers. The `-all` switch causes the command to return a number indicating how many times the regex could be matched. The variables storing the regex and group matches will store the last match in the string only.

The `-inline` switch tells the `regexp` command to return an array with the substring matched by the regular expression and all substrings matched by all capturing groups. If you also specify the `-all` switch, the array will contain the first regex match, all the group matches of the first match, then the second regex match, the group matches of the first match, etc.

The `-start` switch must be followed by a number (as a separate Tcl word) that indicates the character offset in the subject string at which Tcl should attempt the match. Everything before the starting position will be invisible to the regex engine. This means that `\A` will match at the character offset you specify with `-start`, even if that position is not at the start of the string.

Replacing Regex Matches

With the `regsub` command, you can replace regular expression matches in a string.

```
regsub ?switches? regexp subject replacement ?resultvar?
```

Just like the `regexp` command, `regsub` takes zero or more switches followed by a regular expression. It supports the same switches, except for `-inline`. Remember to specify `-all` if you want to replace all matches in the string.

The argument after the `regexp` should be the replacement text. You can specify a literal replacement using the brace syntax, or reference a string variable. The `regsub` command recognizes a few metacharacters in the replacement text. You can use `\0` as a placeholder for the whole regex match, and `\1` through `\9` for the text matched by one of the first nine capturing groups. You can also use `&` as a synonym of `\0`. Note that there's no backslash in front of the ampersand. `&` is substituted with the whole regex match, while `\&` is substituted with a literal ampersand. Use `\\` to insert a literal backslash. You only need to escape backslashes if they're followed by a digit, to prevent the combination from being seen as a backreference. Again, to prevent unnecessary duplication of backslashes, you should enclose the replacement text with braces instead of double quotes. The replacement text `\1` becomes `{\1}` when using braces, and `"\1"` when using quotes.

If you pass a variable reference as the final argument, that variable receives the string with the replacements applied, and `regsub` returns an integer indicating the number of replacements made. Tcl 8.4 and later allow you to omit the final argument. In that case `regsub` returns the string with the replacements applied.

28. VBScript's Regular Expression Support

VBScript has built-in support for regular expressions. If you use VBScript to validate user input on a web page at the client side, using VBScript's regular expression support will greatly reduce the amount of code you need to write.

Microsoft made some significant enhancements to VBScript's regular expression support in version 5.5 of Internet Explorer. Version 5.5 implements quite a few essential regex features that were missing in previous versions of VBScript. Whenever this book mentions VBScript, the statements refer to VBScript's version 5.5 regular expression support.

Basically, Internet Explorer 5.5 implements the JavaScript regular expression flavor. But IE 5.5 did not score very high on web standards. There are quite a few differences between its implementation of JavaScript regular expressions and the actual standard. Fortunately, most are corner cases that are not likely to affect you. Therefore, everything said about JavaScript's regular expression flavor in this book also applies to VBScript. Modern versions of IE still use the IE 5.5 implementation when rendering web pages in quirks mode. In standards mode, modern versions of IE follow the JavaScript standard very closely. VBScript regular expressions also still use the IE 5.5 implementation, even when a modern version of IE is installed.

JavaScript and VBScript implement Perl-style regular expressions. However, they lack quite a number of advanced features available in Perl and other modern regular expression flavors:

- No `\A` or `\Z` anchors to match the start or end of the string. Use a caret or dollar instead.
- Lookbehind is not supported at all. Lookahead is fully supported.
- No atomic grouping or possessive quantifiers
- No Unicode support, except for matching single characters with `\uFFFF`
- No named capturing groups. Use numbered capturing groups instead.
- No mode modifiers to set matching options within the regular expression.
- No conditionals.
- No regular expression comments. Describe your regular expression with VBScript apostrophe comments instead, outside the regular expression string.

Version 1.0 of the `RegExp` object even lacks basic features like lazy quantifiers. This is the main reason this book does not discuss VBScript `RegExp` 1.0. All versions of Internet Explorer prior to 5.5 include version 1.0 of the `RegExp` object. There are no other versions than 1.0 and 5.5.

How to Use the VBScript RegExp Object

You can use regular expressions in VBScript by creating one or more instances of the `RegExp` object. This object allows you to find regular expression matches in strings, and replace regex matches in strings with other strings. The functionality offered by VBScript's `RegExp` object is pretty much bare bones. However, it's more than enough for simple input validation and output formatting tasks typically done in VBScript.

The advantage of the `RegExp` object's bare-bones nature is that it's very easy to use. Create one, put in a regex, and let it match or replace. Only four properties and three methods are available.

After creating the object, assign the regular expression you want to search for to the **Pattern** property. If you want to use a literal regular expression rather than a user-supplied one, simply put the regular expression

in a double-quoted string. By default, the regular expression is case sensitive. Set the **IgnoreCase** property to True to make it case insensitive. The caret and dollar only match at the very start and very end of the subject string by default. If your subject string consists of multiple lines separated by line breaks, you can make the caret and dollar match at the start and the end of those lines by setting the **Multiline** property to True. VBScript does not have an option to make the dot match line break characters. Finally, if you want the RegExp object to return or replace all matches instead of just the first one, set the **Global** property to True.

```
'Prepare a regular expression object
Set myRegExp = New RegExp
myRegExp.IgnoreCase = True
myRegExp.Global = True
myRegExp.Pattern = "regex"
```

After setting the RegExp object's properties, you can invoke one of the three methods to perform one of three basic tasks. The **Test** method takes one parameter: a string to test the regular expression on. **Test** returns True or False, indicating if the regular expression matches (part of) the string. When validating user input, you'll typically want to check if the *entire* string matches the regular expression. To do so, put a caret at the start of the regex, and a dollar at the end, to anchor the regex at the start and end of the subject string.

The **Execute** method also takes one string parameter. Instead of returning True or False, it returns a **MatchCollection** object. If the regex could not match the subject string at all, **MatchCollection.Count** will be zero. If the **RegExp.Global** property is False (the default), **MatchCollection** will contain only the first match. If **RegExp.Global** is true, **Matches** will contain all matches.

The **Replace** method takes two string parameters. The first parameter is the subject string, while the second parameter is the replacement text. If the **RegExp.Global** property is False (the default), **Replace** will return the subject string with the first regex match (if any) substituted with the replacement text. If **RegExp.Global** is true, **Replace** will return the subject string with all regex matches replaced.

You can specify an empty string as the replacement text. This will cause the **Replace** method to return the subject string with all regex matches deleted from it. To re-insert the regex match as part of the replacement, include **\$&** in the replacement text. E.g. to enclose each regex match in the string between square brackets, specify **[\$&]** as the replacement text. If the regex contains capturing parentheses, you can use backreferences in the replacement text. **\$1** in the replacement text inserts the text matched by the first capturing group, **\$2** the second, etc. up to **\$9**. To include a literal dollar sign in the replacements, put two consecutive dollar signs in the string you pass to the **Replace** method.

Getting Information about Individual Matches

The **MatchCollection** object returned by the **RegExp.Execute** method is a collection of **Match** objects. It has only two read-only properties. The **Count** property indicates how many matches the collection holds. The **Item** property takes an index parameter (ranging from zero to **Count-1**), and returns a **Match** object. The **Item** property is the default member, so you can write **MatchCollection(7)** as a shorthand to **MatchCollection.Item(7)**.

The easiest way to process all matches in the collection is to use a **For Each** construct, e.g.:

```
' Pop up a message box for each match
Set myMatches = myRegExp.Execute(subjectString)
For Each myMatch in myMatches
    msgbox myMatch.Value, 0, "Found Match"
Next
```

The **Match** object has four read-only properties. The **FirstIndex** property indicates the number of characters in the string to the left of the match. If the match was found at the very start of the string, **FirstIndex** will be zero. If the match starts at the second character in the string, **FirstIndex** will be one, etc. Note that this is different from the VBScript **Mid** function, which extracts the first character of the string if you set the **start** parameter to one. The **Length** property of the **Match** object indicates the number of characters in the match. The **Value** property returns the text that was matched.

The **SubMatches** property of the **Match** object is a collection of strings. It will only hold values if your regular expression has capturing groups. The collection will hold one string for each capturing group. The **Count** property indicates the number of string in the collection. The **Item** property takes an index parameter, and returns the text matched by the capturing group. The **Item** property is the default member, so you can write `SubMatches(7)` as a shorthand to `SubMatches.Item(7)`. Unfortunately, VBScript does not offer a way to retrieve the match position and length of capturing groups.

Also unfortunately is that the **SubMatches** property does *not* hold the complete regex match as `SubMatches(0)`. Instead, `SubMatches(0)` holds the text matched by the first capturing group, while `SubMatches(SubMatches.Count-1)` holds the text matched by the last capturing group. This is different from most other programming languages. E.g. in VB.NET, `Match.Groups(0)` returns the whole regex match, and `Match.Groups(1)` returns the first capturing group's match. Note that this is also different from the backreferences you can use in the replacement text passed to the `RegExp.Replace` method. In the replacement text, `$1` inserts the text matched by the first capturing group, just like most other regex flavors do. `$0` is not substituted with anything but inserted literally.

29. How to Use Regular Expressions in Visual Basic

Unlike Visual Basic.NET, which has access to the excellent regular expression support of .NET, good old Visual Basic 6 does not ship with any regular expression support. However, VB6 does make it very easy to use functionality provided by ActiveX and COM libraries.

One such library is Microsoft's VBScript scripting library, which has decent regular expression capabilities starting with version 5.5. It implements the regular expression flavor used in JavaScript. This library is part of Internet Explorer 5.5 and later. It is available on all computers running Windows XP, Vista, 7, 8, 8.1, or 10, and previous versions of Windows if the user upgraded to IE 5.5 or later. That includes almost every Windows PC that is used to connect to the Internet.

Internet Explorer 5.5 did not score very high on web standards. There are quite a few differences between its implementation of JavaScript regular expressions and the actual standard. Fortunately, most are corner cases that are not likely to affect you. Modern versions of IE still use the IE 5.5 implementation when rendering web pages in quirks mode. In standards mode, modern versions of IE follow the JavaScript standard very closely. VBScript regular expressions also still use the IE 5.5 implementation, even when a modern version of IE is installed.

To use this library in your Visual Basic application, select Project|References in the VB IDE's menu. Scroll down the list to find the item "Microsoft VBScript Regular Expressions 5.5". It's immediately below the "Microsoft VBScript Regular Expressions 1.0" item. Make sure to tick the 5.5 version, *not* the 1.0 version. The 1.0 version is only provided for backward compatibility. Its capabilities are less than satisfactory.

After adding the reference, you can see which classes and class members the library provides. Select View|Object Browser in the menu. In the Object Browser, select the "VBScript_RegExp_55" library in the drop-down list in the upper left corner. For a detailed description, see the VBScript regular expression reference in this book. Anything said about JavaScript's flavor of regular expressions in the tutorial also applies to VBScript's flavor. The only exception is the character escape support for `\xFF` and `\uFFFF` in the replacement text. JavaScript supports these in string literals, but Visual Basic does not.

The only difference between VB6 and VBScript is that you'll need to use a `Dim` statement to declare the objects prior to creating them. Here's a complete code snippet. It's the two code snippets on the VBScript page put together, with three `Dim` statements added.

```
'Prepare a regular expression object
Dim myRegExp As RegExp
Dim myMatches As MatchCollection
Dim myMatch As Match
Set myRegExp = New RegExp
myRegExp.IgnoreCase = True
myRegExp.Global = True
myRegExp.Pattern = "regex"
Set myMatches = myRegExp.Execute(subjectString)
For Each myMatch in myMatches
    MsgBox(myMatch.Value)
Next
```

30. wxWidgets Supports Three Regular Expression Flavors

wxWidgets uses the exact same regular expression engine that was developed by Henry Spencer for Tcl 8.2. This means that wxWidgets supports the same three regular expressions flavors: Tcl Advanced Regular Expressions, POSIX Extended Regular Expressions and POSIX Basic Regular Expressions. Unlike in Tcl, EREs rather than the far more powerful AREs are the default. The `wxRegex::Replace()` method uses the same syntax for the replacement text as Tcl's `regsub` command.

The wxRegex Class

To use the wxWidgets regex engine, you need to instantiate the `wxRegex` class. The class has two constructors. `wxRegex()` creates an empty regex object. Before you can use the object, you have to call `wxRegex::Compile()`. `wxRegex::IsValid` will return false until you do.

`wxRegex(const wxString& expr, int flags = wxRE_EXTENDED)` creates a `wxRegex` object with a compiled regular expression. The constructor will always create the object, even if your regular expression is invalid. Check `wxRegex::IsValid` to determine if the regular expression was compiled successfully.

`bool wxRegex::Compile(const wxString& pattern, int flags = wxRE_EXTENDED)` compiles a regular expression. You can call this method on any `wxRegex` object, including one that already holds a compiled regular expression. Doing so will simply replace the regular expression held by the `wxRegex` object. Pass your regular expression as a string as the first parameter. The second parameter allows you to set certain matching options.

To set the regex flavor, specify one of the flags `wxRE_EXTENDED`, `wxRE_ADVANCED` or `wxRE_BASIC`. If you specify a flavor, `wxRE_EXTENDED` is the default. I recommend you always specify the `wxRE_ADVANCED` flag. AREs are far more powerful than EREs. Every valid ERE is also a valid ARE, and will give identical results. The only reason to use the ERE flavor is when your code has to work when wxWidgets is compiled without the “built-in” regular expression library (i.e. Henry Spencer’s code).

You can set three other flags in addition to the flavor. `wxRE_ICASE` makes the regular expression case insensitive. The default is case sensitive. `wxRE_NOSUB` makes the regex engine treat all capturing groups as non-capturing. This means you won’t be able to use backreferences in the replacement text, or query the part of the regex matched by each capturing group. If you won’t be using these anyway, setting the `wxRE_NOSUB` flag improves performance.

As discussed in the Tcl section, Henry Spencer’s “ARE” regex engine did away with the confusing “single line” `(?s)` and “multi line” `(?m)` matching modes, replacing them with the equally confusing “non-newline-sensitive” `(?S)`, “partial newline-sensitive” `(?p)`, “inverse partial newline-sensitive” `(?w)` and “newline-sensitive matching” `(?n)`. Since the `wxRegex` class encapsulates the ARE engine, it supports all 4 modes when you use the mode modifiers inside the regular expression. But the `flags` parameter only allows you to set two.

If you add `wxRE_NEWLINE` to the flags, you’re turning on “newline-sensitive matching” `(?n)`. In this mode, the dot will not match newline characters (`\n`). The caret and dollar will match after and before newlines in the string, as well as at the start and end of the subject string.

If you don't set the `wxRE_NEWLINE` flag, the default is “non-newline-sensitive” `(?s)`. In this mode, the dot will match all characters, including newline characters (`\n`). The caret and dollar will match only at the start and end of the subject string. Note that this default is different from the default in Perl and every other regex engine on the planet. In Perl, by default, the dot does not match newline characters, and the caret and dollar only match at the start and end of the subject string. The only way to set this mode in `wxWidgets` is to put `(?p)` at the start of your regex.

Putting it all together, `wxRegex(_T("(?p)^[a-z].*$"), wxRE_ADVANCED + wxRE_ICASE)` will check if your subject string consists of a single line that starts with a letter. The equivalent in Perl is `m/^[a-z].*/i`.

wxRegex Status Functions

`wxRegex::IsValid()` returns true when the `wxRegex` object holds a compiled regular expression.

`wxRegex::GetMatchCount()` is rather poorly named. It does not return the number of matches found by `Matches()`. In fact, you can call `GetMatchCount()` right after `Compile()`, before you call `Matches`. `GetMatchCount()` it returns the number of capturing groups in your regular expression, plus one for the overall regex match. You can use this to determine the number of backreferences you can use the replacement text, and the highest index you can pass to `GetMatch()`. If your regex has no capturing groups, `GetMatchCount()` returns 1. In that case, `\0` is the only valid backreference you can use in the replacement text.

`GetMatchCount()` returns 0 in case of an error. This will happen if the `wxRegex` object does not hold a compiled regular expression, or if you compiled it with `wxRE_NOSUB`.

Finding and Extracting Matches

If you want to test whether a regex matches a string, or extract the substring matched by the regex, you first need to call the `wxRegex::Matches()` method. It has 3 variants, allowing you to pass `wxChar` or `wxString` as the subject string. When using a `wxChar`, you can specify the length as a third parameter. If you don't, `wxStrLen()` will be called to compute the length. If you plan to loop over all regex matches in a string, you should call `wxStrLen()` yourself outside the loop and pass the result to `wxRegex::Matches()`.

```
bool wxRegex::Matches(const wxChar* text, int flags = 0) const
bool wxRegex::Matches(const wxChar* text, int flags, size_t len) const
bool wxRegex::Matches(const wxString& text, int flags = 0) const
```

`Matches()` returns true if the regex matches all or part of the subject string that you passed in the `text` parameter. Add anchors to your regex if you want to set whether the regex matches the whole subject string.

Do not confuse the `flags` parameter with the one you pass to the `Compile()` method or the `wxRegex()` constructor. All the flavor and matching mode options can only be set when compiling the regex.

The `Matches()` method allows only two flags: `wxRE_NOTBOL` and `wxRE_NOTEOL`. If you set `wxRE_NOTBOL`, then `^` and `\A` will not match at the start of the string. They will still match after embedded newlines if you turned on that matching mode. Likewise, specifying `wxRE_NOTEOL` tells `$` and `\Z` not to match at the end of the string.

`wxRE_NOTBOL` is commonly used to implement a “find next” routine. The `wxRegEx` class does not provide such a function. To find the second match in the string, you’ll need to call `wxRegEx::Matches()` and pass it the part of the original subject string after the first match. Pass the `wxRE_NOTBOL` flag to indicate that you’ve cut off the start of the string you’re passing.

`wxRE_NOTEOL` can be useful if you’re processing a large set of data, and you want to apply the regex before you’ve read the whole data. Pass `wxRE_NOTEOL` while calling `wxRegEx::Matches()` as long as you haven’t read the entire string yet. Pass both `wxRE_NOTBOL` and `wxRE_NOTEOL` when doing a “find next” on incomplete data.

After a call to `Matches()` returns true, and you compiled your regex without the `wxRE_NOSUB` flag, you can call `GetMatch()` to get details about the overall regex match, and the parts of the string matched by the capturing groups in your regex.

`bool wxRegEx::GetMatch(size_t* start, size_t* len, size_t index = 0) const` retrieves the starting position of the match in the subject string, and the number of characters in the match.

`wxString wxRegEx::GetMatch(const wxString& text, size_t index = 0) const` returns the text that was matched.

For both calls, set the index parameter to zero (or omit it) to get the overall regex match. Set $1 \leq \text{index} < \text{GetMatchCount}()$ to get the match of a capturing group in your regular expression. To determine the number of a group, count the opening brackets in your regular expression from left to right.

Searching and Replacing

The `wxRegEx` class offers three methods to do a search-and-replace. `Replace()` is the method that does the actual work. You can use `ReplaceAll()` and `ReplaceFirst()` as more readable ways to specify the 3rd parameter to `Replace()`.

`int wxRegEx::ReplaceAll(wxString* text, const wxString& replacement) const` replaces all regex matches in `text` with `replacement`.

`int wxRegEx::ReplaceFirst(wxString* text, const wxString& replacement) const` replaces the first match of the regular expression in `text` with `replacement`.

`int wxRegEx::Replace(wxString* text, const wxString& replacement, size_t maxMatches = 0) const` allows you to specify how many replacements will be made. Passing 0 for `maxMatches` or omitting it does the same as `ReplaceAll()`. Setting it to 1 does the same as `ReplaceFirst()`. Pass a number greater than 1 to replace only the first `maxMatches` matches. If `text` contains fewer matches than you’ve asked for, then all matches will be replaced, without triggering an error.

All three calls return the actual number of replacements made. They return zero if the regex failed to match the subject text. A return value of -1 indicates an error. The replacements are made directly to the `wxString` that you pass as the first parameter.

`wxWidgets` uses the same syntax as `Tcl` for the replacement text. You can use `\0` as a placeholder for the whole regex match, and `\1` through `\9` for the text matched by one of the first nine capturing groups. You can also use `&` as a synonym of `\0`. Note that there’s no backslash in front of the ampersand. `&` is substituted

with the whole regex match, while `\&` is substituted with a literal ampersand. Use `\\` to insert a literal backslash. You only need to escape backslashes if they're followed by a digit, to prevent the combination from being seen as a backreference. When specifying the replacement text as a literal string in C++ code, you need to double up all the backslashes, as the C++ compiler also treats backslashes as escape characters. So if you want to replace the match with the first backreference followed by the text `&co`, you'll need to code that in C++ as `_T("\\1\\&co")`.

31. XML Schema Regular Expressions

The W3C XML Schema standard defines its own regular expression flavor. You can use it in the `pattern` facet of simple type definitions in your XML schemas. E.g. the following defines the simple type “SSN” using a regular expression to require the element to contain a valid US social security number.

```
<xsd:simpleType name="SSN">
  <xsd:restriction base="xsd:token">
    <xsd:pattern value="[0-9]{3}-[0-9]{2}-[0-9]{4}"/>
  </xsd:restriction>
</xsd:simpleType>
```

Compared with other regular expression flavors, the XML schema flavor is quite limited in features. Since it’s only used to validate whether an entire element matches a pattern or not, rather than for extracting matches from large blocks of data, you won’t really miss the features often found in other flavors. The limitations allow schema validators to be implemented with efficient text-directed engines.

Particularly noteworthy is the complete absence of anchors like the caret and dollar, word boundaries, and lookaround. XML schema always implicitly anchors the entire regular expression. The regex must match the whole element for the element to be considered valid. If you have the pattern `regex`, the XML schema validator will apply it in the same way as say Perl, Java or .NET would do with the pattern `^regex$`. If you want to accept all elements with `regex` somewhere in the middle of their contents, you’ll need to use the regular expression `.*regex.*`. The two `.*` expand the match to cover the whole element, assuming it doesn’t contain line breaks. If you want to allow line breaks, you can use something like `[\s\S]*regex[\s\S]*`. Combining a shorthand character class with its negated version results in a character class that matches anything.

XML schemas do not provide a way to specify matching modes. The dot never matches line breaks, and patterns are always applied case sensitively. If you want to apply `literal` case insensitively, you’ll need to rewrite it as `[lL][iI][tT][eE][rR][aA][lL]`.

XML regular expressions don’t have any tokens like `\xFF` or `\uFFFF` to match particular (non-printable) characters. You have to add them as literal characters to your regex. If you are entering the regex into an XML file using a plain text editor, then you can use the `ÿ`; XML syntax. Otherwise, you’ll need to paste in the characters from a character map.

Lazy quantifiers are not available. Since the pattern is anchored at the start and the end of the subject string anyway, and only a success/failure result is returned, the only potential difference between a greedy and lazy quantifier would be performance. You can never make a fully anchored pattern match or fail by changing a greedy quantifier into a lazy one or vice versa.

XML Schema regular expressions support the following:

- Character classes, including shorthands, ranges and negated classes.
- Character class subtraction.
- The dot, which matches any character except line breaks.
- Alternation and groups.
- Greedy quantifiers `?`, `*`, `+` and `{n,m}`
- Unicode properties and blocks

Note that the regular expression functions available in XQuery and XPath use a different regular expression flavor. This flavor is a superset of the XML Schema flavor described here. It adds some of the features that are available in many modern regex flavors, but not in the XML Schema flavor.

XML Character Classes

Despite its limitations, XML schema regular expressions introduce two handy features. The special shorthand character classes `\i` and `\c` make it easy to match XML names. No other regex flavor supports these.

Character class subtraction makes it easy to match a character that is in a certain list, but not in another list. E.g. `[a-z-[aeiou]]` matches an English consonant. This feature is now also available in the JGsoft and .NET regex engines. It is particularly handy when working with Unicode properties. E.g. `[\p{L}-[\p{IsBasicLatin}]]` matches any letter that is not an English letter.

32. XQuery and XPath Regular Expressions

The W3C standard for XQuery 1.0 and XPath 2.0 Functions and Operators defines three functions `fn:matches`, `fn:replace` and `fn:tokenize` that take a regular expression as one of their parameters. The XQuery and XPath standard introduces a new regular expression flavor for this purpose. This flavor is identical to the XML Schema flavor, with the addition of several features that are available in many modern regex flavors, but not in the XML Schema flavor. All valid XML Schema regexes are also valid XQuery/XPath regexes. The opposite is not always true.

Because the XML Schema flavor is only used for true/false validity tests, these features were eliminated for performance reasons. The XQuery and XPath functions perform more complex regular expression operators, which require a more feature-rich regular expression flavor. That said, the XQuery and XPath regex flavor is still limited by modern standards.

XQuery and XPath support the following features on top of the features in the XML Schema flavor:

- `^` and `$` anchors that match at the start or end of the string, or the start or end of a line (see matching modes below). These are the only two anchors supported.
- Lazy quantifiers, using the familiar question mark syntax.
- Backreferences and capturing groups. The XML Schema standard supports grouping, but groups were always non-capturing. XQuery/XPath allows backreferences to be used in the regular expression. `fn:replace` supports backreferences in the replacement text using the `$1` notation.

While XML Schema allows no matching modes at all, the XQuery and XPath functions all accept an optional `flags` parameter to set matching modes. Mode modifiers within the regular expression are not supported. These four matching modes are available:

- `i` makes the regex match case insensitive.
- `s` enables “single-line mode”. In this mode, the dot matches newlines.
- `m` enables “multi-line mode”. In this mode, the caret and dollar match before and after newlines in the subject string.
- `x` enables “free-spacing mode”. In this mode, whitespace between regex tokens is ignored. Comments are not supported.

The flags are specified as a string with the letters of the modes you want to turn on. E.g. “`ix`” turns on case insensitivity and free-spacing. If you don’t want to set any matching modes, you can pass an empty string for the `flags` parameter, or omit the parameter entirely.

Three Regex Functions

`fn:matches`(`subject`, `pattern`, `flags`) takes a subject string and a regular expression as input. If the regular expression matches any part of the subject string, the function returns true. If it cannot match at all, it returns false. You’ll need to use anchors if you only want the function to return true when the regex matches the entire subject string.

`fn:replace`(`subject`, `pattern`, `replacement`, `flags`) takes a subject string, a regular expression, and a replacement string as input. It returns a new string that is the subject string with all matches of the regex pattern replaced with the replacement text. You can use `$1` to `$99` to re-insert capturing groups into

the replacement. `$0` inserts the whole regex match. Literal dollar signs and backslashes in the replacement must be escaped with a backslash.

`fn:replace` cannot replace zero-length matches. E.g. `fn:replace("test", "^", "prefix")` will raise an error rather than returning “prefixtext” like regex-based search-and-replace does in most programming languages.

`fn:tokenize`(`subject`, `pattern`, `flags`) is like the “split” function in many programming languages. It returns an array of strings that consists of all the substrings in `subject` between all the regex matches. The array will not contain the regex matches themselves. If the regex matches the first or last character in the `subject` string, then the first or last string in the resulting array will be empty strings.

`fn:tokenize` also cannot handle zero-length regular expression matches.

33. XRegExp Regular Expression Library for JavaScript

XRegExp is an open source JavaScript library developed by Steven Levithan. It supports all modern browsers, as well as many older and even ancient browser versions. It can also be used on the server with Node.js. You can download XRegExp at xregexp.com.

Using the XRegExp object instead of JavaScript's built-in RegExp object provides you with a regular expression syntax with more features and fewer cross-browser inconsistencies. Notable added features include free-spacing, named capture, mode modifiers, and Unicode categories, blocks, and scripts. It also treats invalid escapes and non-existent backreferences as errors.

XRegExp also provides its own `replace()` method with a replacement text syntax that is enhanced with named backreferences and no cross-browser inconsistencies. It also provides a `split()` method that is fully compliant with the JavaScript standard.

To use XRegExp, first create a regular expression object with `var myre = XRegExp('regex', 'flags')` where `flags` is a combination of the letters `g` (global), `i` (case insensitive), `m` (anchors match at line breaks), `s` (dot matches line breaks), `x` (free-spacing), and `n` (explicit capture). XRegExp 3 adds the `A` (astral) flag which includes Unicode characters beyond U+FFFF when matching Unicode properties and blocks. The ECMAScript 6 flags `y` (sticky) and `u` (Unicode) can also be used in modern browsers that support them natively, but they'll throw errors in browsers that don't have built-in support for these flags.

You can then pass the XRegExp instance you constructed to various XRegExp methods. It's important to make the calls as shown below to get the full XRegExp functionality. The object returned by the XRegExp constructor is a native JavaScript RegExp object. That object's methods are the browser's built-in RegExp methods. You can replace the built-in RegExp methods with XRegExp's methods by calling `XRegExp.install('natives')`. Doing so also affects RegExp objects constructed by the normal RegExp constructor or double-slashed regex literals.

`XRegExp.test(str, regex, [pos=0], [sticky=false])` tests whether the regex can match part of a string. The `pos` argument is a zero-based index in the string where the match attempt should begin. If you pass `true` or `'sticky'` for the `sticky` parameter, then the match is only attempted at `pos`. This is similar to adding the start-of-attempt anchor `\G` (which XRegExp doesn't support) to the start of your regex in other flavors.

`XRegExp.exec(str, regex, [pos=0], [sticky=false])` does the same as `XRegExp.test()` but returns `null` or an array instead of `false` or `true`. Index 0 in the array holds the overall regex match. Indexes 1 and beyond hold the text matched by capturing groups, if any. If the regex has named capturing groups then their matches are available as properties on the array in XRegExp 4 and prior. In XRegExp 5 the array has a `group` property which then has the names of the capturing groups as properties. `XRegExp.exec()` does not rely on the `lastIndex` property and thus avoids cross-browser problems with that property.

`XRegExp.forEach(str, regex, callback)` makes it easy to iterate over all matches of the regex in a string. It always iterates over all matches, regardless of the `global` flag and the `lastIndex` property. The callback is called with four arguments. The first two are an array like returned by `exec()` and the index in the string that the match starts at. The last two are `str` and `regex` exactly as you passed them to `forEach()`.

`XRegExp.replace(str, regex, replacement, [scope])` returns a string with the matches of `regex` in `str` replaced with `replacement`. Pass `'one'` or `'all'` as the `scope` argument to replace only the first

match or all matches. If you omit the `scope` argument then the `regex.global` flag determines whether only the first or all matches are replaced.

The `XRegExp.replace()` method uses its own replacement text syntax. It is very similar to the native JavaScript syntax. It is somewhat incompatible by making dollar signs that don't form valid replacement tokens an error. But the benefit is that it eliminates all cross-browser inconsistencies. `$$` inserts a single literal dollar sign. `&` and `$0` insert the overall regex match. ``` and `'` insert the part of the subject string to the left and the right of the regex match. `$n`, `$nn`, `{n}`, and `{nn}` are numbered backreferences while `{name}` is a named backreference.

If you pass a function as the `replacement` parameter, then it will be called with three or more arguments. The first argument is the string that was matched, with named capturing groups available through properties on that string. The second and following arguments are the strings matched by each of the capturing groups in the regex, if any. The final two arguments are the index in the string at which the match was found and the original subject string.

`XRegExp.split(str, regex, [limit])` is an alternative to `String.prototype.split`. It accurately follows the JavaScript standard for splitting strings, eliminating all cross-browser inconsistencies and bugs.

Part 5

Regular Expressions Reference

1. Regular Expressions Reference

The regular expressions reference in this book functions both as a reference to all available regex syntax and as a comparison of the features supported by the regular expression flavors discussed in the tutorial. The reference tables pack an incredible amount of information. To get the most out of them, follow this legend to learn how to read them.

The tables have five lines for each regular expression feature. The first four explain the feature.

Feature	The name of the feature, which also serves as a link to the relevant section in the tutorial.
Syntax	The actual regex syntax for this feature. If the syntax is fixed, it is simply shown as such. If the syntax has variable elements, the syntax is described.
Description	Summary of what the feature does.
Example	Functional regular expression that demonstrates the feature.

The fifth line lists the flavors that support the feature. The name of the flavor may be followed by an indicator between parentheses. The table below explains the possible indicators.

(no indicator)	All versions of this flavor support this feature.
(3.0)	Version 3.0 and all later versions of this flavor support this feature. Earlier versions do not support it.
(2.0 only)	Only version 2.0 supports this feature. Earlier and later versions do not support it.
(2.0–2.9)	Only versions 2.0 through 2.9 supports this feature. Earlier and later versions do not support it.
(Unicode)	This feature works with Unicode characters in all versions of this flavor.
(code page)	This feature works with the characters in the active code page in all versions of this flavor.
(ASCII)	This feature works with ASCII characters only in all versions of this flavor.
(3.0 Unicode)	This feature works with Unicode characters in versions 3.0 and later of this flavor. Earlier versions do not support it at all.
(3.0 Unicode 2.0 ASCII)	This feature works with Unicode characters in versions 3.0 and later this flavor. It works with ASCII characters in versions 2.0 through 2.9. Earlier versions do not support it at all.
(3.0 Unicode 2.0 code page)	This feature works with Unicode characters in versions 3.0 and later this flavor. It works with the characters in the active code page in versions 2.0 through 2.9. Earlier versions do not support it at all.
(string)	The regex flavor does not support this syntax. But string literals in the programming language that this regex flavor is normally used with do support this syntax.
(3.0; 1.0 string)	Version 3.0 and later of this regex flavor support this syntax. Earlier versions of the regex flavor do not support this syntax. But string literals in the programming language that this regex flavor is normally used with have supported this syntax since version 1.0.
(option)	All versions of this regex flavor support this feature if you set a particular option or precede it with a particular mode modifier.
(option 3.0)	Version 3.0 and all later versions of this regex flavor support this feature if you set a particular option or precede it with a particular mode modifier. Earlier versions either do not support the syntax at all or do not support the mode modifier to change the behavior of the syntax to what the feature describes.

(3.0; 2.0 fail)	Version 3.0 and all later versions of this regex flavor support this feature. Version 2.0 all later releases prior to 3.0 recognize the syntax, but always fail to match this regex token. Versions prior to 2.0 do not support the syntax.
(fail)	The syntax is recognized by the flavor and regular expressions using it work, but this particular regex token always fails to match. The regex can only find matches if this token is made optional by alternation or a quantifier.
(2.0–2.9 fail)	Versions 2.0 through 2.9 recognize the syntax, but always fail to match this regex token. Earlier and later versions either don't recognize the syntax or treat it as a syntax error.
(ignored)	The syntax is recognized by the flavor but it does not do anything useful. This particular regex token always finds a zero-length match.
(error)	The syntax is recognized by the flavor but it is treated as a syntax error.

If a flavor is omitted from the list of flavors, then no version of that flavor support the feature. The flavor may use the same syntax for a different feature which is indicated elsewhere in the reference table. Or the syntax may trigger an error or it may be interpreted as plain text.

When this legend says “all versions” or “no version”, that means all or none of the versions of each flavor that are covered by the reference tables:

JGsoft	V1: EditPad 6 and 7; PowerGREP 3 and 4; AceText 3 V2: EditPad 8; PowerGREP 5; AceText 4
.NET	1.0–1.1: first 2 releases of the original .NET framework 2.0–7.0: .NET Framework 2.0–4.8, .NET Core 1.0.0–3.1.0, and .NET 5.0–7.0 1.0–7.0: all versions of the .NET Framework and .NET Core
Java	4–21
Perl	5.8–5.32
PCRE	4.0–8.45
PCRE2	10.00–10.39
PHP	5.0.0–8.1.24
Delphi	XE–XE8 & 10–10.4 & 11; TRegEx only; also applies to C++Builder XE–XE8 & 10–10.4 & 11
R	2.14.0–4.2.1
JavaScript	Latest versions of Chrome, Edge, and Firefox
VBScript	VBScript and Internet Explorer in quirks mode
XRegExp	2.0.0–5.1.0
Python	2.4–3.12
Ruby	1.8–3.2
std::regex	Visual C++ 2008–2022 (Dinkumware std library)
boost::regex	1.38–1.39 & 1.42–1.83
Tcl ARE	8.4–8.6
POSIX BRE	IEEE Std 1003.1
POSIX ERE	IEEE Std 1003.1
GNU BRE	
GNU ERE	
Oracle	10gR1, 10gR2, 11gR1, 11gR2, 12c

XML	1.0–1.1
XPath	2.0–3.1

For the .NET flavor, some features are indicated with “ECMA” or “non-ECMA”. That means the feature is only supported when `RegexOptions.ECMAScript` is set or is not set. Features indicated with “non-ECMA Unicode” match ASCII characters when `RegexOptions.ECMAScript` is set and Unicode characters when `RegexOptions.ECMAScript` is not set. Everything that applies to .NET 2.0 or later also applies to any version of .NET Core. The Visual Studio IDE uses the non-ECMA .NET flavor starting with VS 2012.

For the `std::regex` and `boost::regex` flavor there are additional indicators ECMA, basic, extended, grep, egrep, and awk. When one or more of these appear, that means that the feature is only supported if you specify one of these grammars when compiling your regular expression. Features with Unicode indicators match Unicode characters when using `std::wregex` or `boost::wregex` on wide character strings. In the replacement string reference, the additional indicators are `sed` and `default`. When either one appears, the feature is only supported when you either pass or don't pass `match_flag_type::format_sed` to `regex_replace()`. For `boost`, there is one more replacement indicator “all” that indicates the feature is only supported when you pass `match_flag_type::format_all` to `regex_replace()`.

For the PCRE2 flavor, some replacement string features are indicated with “extended”. This means the feature is only supported when you pass `PCRE2_SUBSTITUTE_EXTENDED` to `pcre2_substitute`.

2. Special and Non-Printable Characters

Feature: Literal character
 Syntax: Any character except `[\^$.|?*+()`
 Description: All characters except the listed special characters match a single instance of themselves
 Example: `a` matches `a`
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex; Boost; Tcl ARE; POSIX BRE; POSIX ERE; GNU BRE; GNU ERE; Oracle; XML; XPath

Feature: Literal curly braces
 Syntax: `{` and `}`
 Description: `{` and `}` are literal characters, unless they're part of a valid regular expression token such as a quantifier `{3}`
 Example: `{` matches `{`
 Supported by: JGsoft; .NET; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby (1.9); std::regex (basic; grep); Boost (ECMA 1.54–1.83; basic 1.38–1.83; grep 1.38–1.83); Tcl ARE; POSIX BRE; POSIX ERE; GNU BRE; Oracle; XML

Feature: Backslash escapes a metacharacter
 Syntax: `\` followed by any of `[\^$.|?*+(){}]`
 Description: A backslash escapes special characters to suppress their special meaning
 Example: `*` matches `*`
 Supported by: JGsoft; .NET; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby (1.9); std::regex (basic; grep); Boost (ECMA 1.54–1.83; basic 1.38–1.83; grep 1.38–1.83); Tcl ARE; POSIX BRE; POSIX ERE; GNU BRE; Oracle; XML

Feature: Escape sequence
 Syntax: `\Q` . . . `\E`
 Description: Matches the characters between `\Q` and `\E` literally, suppressing the meaning of special characters
 Example: `\Q+ - */ \E` matches `+ - */`
 Supported by: JGsoft; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; Boost (ECMA; extended; egrep; awk)

Feature: Hexadecimal escape
 Syntax: `\xFF` where FF are 2 hexadecimal digits
 Description: Matches the character at the specified position in the code page
 Example: `\xA9` matches `©` when using the Latin-1 code page
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex (ECMA); Boost (ECMA; extended; egrep; awk); Tcl ARE; POSIX BRE (string); POSIX ERE (string); GNU BRE (string); GNU ERE (string)

Feature: Character escape
 Syntax: `\n`, `\r` and `\t`
 Description: Match an LF character, CR character and a tab character respectively
 Example: `\r\n` matches a Windows CRLF line break
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex (ECMA; awk); Boost (ECMA; extended; egrep; awk); Tcl ARE; POSIX BRE (string); POSIX ERE (string); GNU BRE (string); GNU ERE (string); XML; XPath

Feature: Line break
 Syntax: `\R`
 Description: Matches any line break, including CRLF as a pair, CR only, LF only, form feed, vertical tab, and any Unicode line break
 Example:
 Supported by: JGsoft (V2); Java (8); Perl (5.10); PCRE (7.0); PCRE2; PHP (5.2.2); Delphi; R; Ruby (2.0); Boost (ECMA; 1.42–1.83)

Feature: Line break
 Syntax: `\R`
 Description: Matches the next line control character U+0085
 Example:
 Supported by: JGsoft (V2); Java (8); Perl (5.10); PCRE (7.0); PCRE2; PHP (5.2.2); Delphi; R; Ruby (2.0); Boost (ECMA; 1.54–1.83)

Feature: Line break
 Syntax: `\R`
 Description: CRLF line breaks are indivisible
 Example: `\R{2}` and `\R\R` cannot match `\r\n`
 Supported by: JGsoft; PCRE2; R; Ruby; Boost

Feature: Line break
 Syntax: Literal CRLF, LF, or CR line break
 Description: Matches CRLF as a pair, CR only, and LF only regardless of the line break style used in the regex
 Example:
 Supported by: JGsoft; XML; XPath

Feature: Character escape
 Syntax: `\a`
 Description: Match the “alert” or “bell” control character (ASCII 0x07)
 Example:
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; Python; Ruby; std::regex (awk); Boost (ECMA; extended; egrep; awk); Tcl ARE

Feature: Character escape
 Syntax: `\b`
 Description: Match the “backspace” control character (ASCII 0x08)
 Example:
 Supported by: std::regex (awk); Tcl ARE

Feature: Character escape
 Syntax: `\B`
 Description: Match a backslash
 Example: `\B` matches `\`
 Supported by: Tcl ARE

Feature: Character escape
 Syntax: `\e`
 Description: Match the “escape” control character (ASCII 0x1B)
 Example:
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; Ruby; Boost (ECMA; extended; egrep; awk); Tcl ARE

Feature: Character escape
 Syntax: `\f`
 Description: Match the “form feed” control character (ASCII 0x0C)
 Example:
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex (ECMA; awk); Boost (ECMA; extended; egrep; awk); Tcl ARE

Feature: Character escape
 Syntax: `\v`
 Description: Match the “vertical tab” control character (ASCII 0x0B), but not any other vertical whitespace.
 Example:
 Supported by: JGsoft (V1 only); .NET; Java (4–7); JavaScript; VBScript; XRegExp; Python; Ruby; std::regex (ECMA; awk); Boost (ECMA 1.38–1.39; extended 1.38–1.83; egrep 1.38–1.83; awk 1.38–1.83); Tcl ARE

Feature: Control character escape
 Syntax: `\cA` through `\cZ`
 Description: Match an ASCII character Control+A through Control+Z, equivalent to `\x01` through `\x1A`
 Example: `\cM\cJ` matches a Windows CRLF line break
 Supported by: JGsoft (V1 only); .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Ruby; std::regex (ECMA); Boost (ECMA; extended; egrep; awk); Tcl ARE

Feature: Control character escape
 Syntax: `\ca` through `\cz`
 Description: Match an ASCII character Control+A through Control+Z, equivalent to `\x01` through `\x1A`
 Example: `\cm\cj` matches a Windows CRLF line break
 Supported by: JGsoft (V1 only); .NET; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Ruby; std::regex (ECMA); Boost (ECMA; extended; egrep; awk); Tcl ARE

Feature: NULL escape
 Syntax: `\0`
 Description: Match the NULL character
 Example:
 Supported by: .NET; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex (ECMA); Boost; Tcl ARE

Feature: Octal escape
 Syntax: `\o{7777}` where 7777 is any octal number
 Description: Matches the character at the specified position in the active code page
 Example: `\o{20254}` matches € when using Unicode
 Supported by: JGsoft (V2); Perl (5.14); PCRE (8.34); PCRE2; PHP (5.5.10); Delphi (XE7); R (3.0.3)

- Feature: Octal escape
 Syntax: `\1` through `\7`
 Description: Matches the character at the specified position in the ASCII table
 Example: `\7` matches the “bell” character
 Supported by: .NET (ECMA); JavaScript; VBScript; std::regex (awk)
- Feature: Octal escape
 Syntax: `\10` through `\77`
 Description: Matches the character at the specified position in the ASCII table
 Example: `\77` matches `?`
 Supported by: .NET; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; Ruby; std::regex (awk); Tcl ARE
- Feature: Octal escape
 Syntax: `\100` through `\177`
 Description: Matches the character at the specified position in the ASCII table
 Example: `\100` matches `@`
 Supported by: .NET; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; Python; Ruby; std::regex (awk); Tcl ARE
- Feature: Octal escape
 Syntax: `\200` through `\377`
 Description: Matches the character at the specified position in the active code page
 Example: `\377` matches `ÿ` when using the Latin-1 code page
 Supported by: .NET (2.0–7.0); Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; Python; Ruby (1.8 only fail); std::regex (awk); Tcl ARE
- Feature: Octal escape
 Syntax: `\400` through `\777`
 Description: Matches the character at the specified position in the active code page
 Example: `\777` matches `ø` when using Unicode
 Supported by: .NET (non-ECMA; 1.0–1.1 fail); Perl (5.14); PCRE (6.7); PCRE2; PHP (5.2.0); Delphi; R; Python (3.5 error); Ruby (1.8 only fail); std::regex (awk); Tcl ARE (8.4–8.5)
- Feature: Octal escape
 Syntax: `\01` through `\07`
 Description: Matches the character at the specified position in the ASCII table
 Example: `\07` matches the “bell” character
 Supported by: JGsoft (V1 only); .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; Python; Ruby; std::regex (awk); Boost; Tcl ARE
- Feature: Octal escape
 Syntax: `\010` through `\077`
 Description: Matches the character at the specified position in the ASCII table
 Example: `\077` matches `?`
 Supported by: JGsoft (V1 only); .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; Python; Ruby; std::regex (awk); Boost; Tcl ARE

Feature: Octal escape
Syntax: `\0100` through `\0177`
Description: Matches the character at the specified position in the ASCII table
Example: `\0100` matches `@`
Supported by: JGsoft (V1 only); Java; Boost

Feature: Octal escape
Syntax: `\0200` through `\0377`
Description: Matches the character at the specified position in the active code page
Example: `\0377` matches `ÿ` when using the Latin-1 code page
Supported by: JGsoft (V1 only); Java; Boost

3. Basic Features

Feature: Dot
 Syntax: `.` (dot)
 Description: Matches any single character except line break characters. Most regex flavors have an option to make the dot match line break characters too.
 Example: `.` matches `x` or (almost) any other character
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex; Boost; Tcl ARE; POSIX BRE; POSIX ERE; GNU BRE; GNU ERE; Oracle; XML; XPath

Feature: Not a line break
 Syntax: `\N`
 Description: Matches any single character except line break characters, like the dot, but is not affected by any options that make the dot match all characters including line breaks.
 Example: `\N` matches `x` or any other character that is not a line break
 Supported by: JGsoft (V2); Perl (5.12); PCRE (8.10); PCRE2; PHP (5.3.4); Delphi (XE7); R

Feature: Alternation
 Syntax: `|` (pipe)
 Description: Causes the regex engine to match either the part on the left side, or the part on the right side. Can be strung together into a series of alternatives.
 Example: `abc|def|xyz` matches `abc`, `def` or `xyz`
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex (ECMA; extended; egrep; awk); Boost (ECMA; extended; egrep; awk); Tcl ARE; POSIX ERE; GNU ERE; Oracle; XML; XPath

Feature: Alternation
 Syntax: `\|` (backslash pipe)
 Description: Causes the regex engine to match either the part on the left side, or the part on the right side. Can be strung together into a series of alternatives.
 Example: `abc\|def\|xyz` matches `abc`, `def` or `xyz`
 Supported by: GNU BRE

Feature: Alternation
 Syntax: Literal line feed not inside a group or character class
 Description: Causes the regex engine to match either the part on the left side, or the part on the right side. Can be strung together into a series of alternatives.
 Example: `abc`
`def`
`xyz` matches `abc`, `def` or `xyz`
 Supported by: std::regex (grep; egrep); Boost (grep; egrep)

Feature: Alternation
 Syntax: Literal line feed inside a group but not inside a character class
 Description: Causes the group to match either the part on the left side, or the part on the right side. Can be strung together into a series of alternatives.
 Example: `a(bc`
`de`
`fg)h` matches `abch`, `adeh` or `afgh`
 Supported by: Boost (grep; egrep)

Feature: Alternation is eager
 Syntax: `|` or `\|`
 Description: Alternation returns the first alternative that matches.
 Example: `a|ab` matches `a` in `ab`
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex (ECMA); Boost (ECMA); Oracle; XML; XPath

Feature: Alternation is greedy
 Syntax: `|` or `\|`
 Description: Alternation returns the longest alternative that matches.
 Example: `a|ab` matches `ab` in `ab`
 Supported by: std::regex (extended; grep; egrep; awk); Boost (extended; grep; egrep; awk); Tcl ARE; POSIX ERE; GNU BRE; GNU ERE

4. Character Classes

- Feature: Character class
 Syntax: **[**
 Description: When used outside a character class, **[** begins a character class. Inside a character class, different rules apply. Unless otherwise noted, the syntax on this page is only valid inside character classes, while the syntax on all other reference pages is not valid inside character classes.
- Example:
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex; Boost; Tcl ARE; POSIX BRE; POSIX ERE; GNU BRE; GNU ERE; Oracle; XML; XPath
- Feature: Literal character
 Syntax: Any character except **^ -] **
 Description: All characters except the listed special characters are literal characters that add themselves to the character class.
- Example: **[abc]** matches **a**, **b** or **c**
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex; Boost; Tcl ARE; POSIX BRE; POSIX ERE; GNU BRE; GNU ERE; Oracle; XML; XPath
- Feature: Backslash escapes a metacharacter
 Syntax: **** (backslash) followed by any of **^ -] **
 Description: A backslash escapes special characters to suppress their special meaning.
- Example: **[\^\^]** matches **^** or **]**
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex (ECMA); Boost (ECMA; awk); Tcl ARE; XML; XPath
- Feature: Literal backslash
 Syntax: ****
 Description: A backslash is a literal character that adds a backslash to the character class.
- Example: **[\]** matches ****
 Supported by: std::regex (basic; extended; grep; egrep; awk); Boost (basic; extended; grep; egrep); POSIX BRE; POSIX ERE; GNU BRE; GNU ERE; Oracle
- Feature: Range
 Syntax: **-** (hyphen) between two tokens that each specify a single character.
 Description: Adds a range of characters to the character class.
- Example: **[a-zA-Z0-9]** matches any ASCII letter or digit
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex; Boost; Tcl ARE; POSIX BRE; POSIX ERE; GNU BRE; GNU ERE; Oracle; XML; XPath

Feature: Negated character class
 Syntax: `^` (caret) immediately after the opening `[`
 Description: Negates the character class, causing it to match a single character *not* listed in the character class.
 Example: `[^a-d]` matches `x` (any character except a, b, c or d)
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex; Boost; Tcl ARE; POSIX BRE; POSIX ERE; GNU BRE; GNU ERE; Oracle; XML; XPath

Feature: Literal opening bracket
 Syntax: `[`
 Description: An opening square bracket is a literal character that adds an opening square bracket to the character class.
 Example: `[ab[cd]ef]` matches `ae]f`, `bef]`, `[ef]`, `cef]`, and `def]`
 Supported by: JGsoft; .NET; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; std::regex; Boost; Tcl ARE; POSIX BRE; POSIX ERE; GNU BRE; GNU ERE; Oracle

Feature: Nested character class
 Syntax: `[`
 Description: An opening square bracket inside a character class begins a nested character class.
 Example: `[ab[cd]ef]` is the same as `[abcdef]` and matches any letter between `a` and `f`.
 Supported by: Java; Ruby (1.9)

Feature: Character class subtraction
 Syntax: `[base-[subtract]]`
 Description: Removes all characters in the “subtract” class from the “base” class.
 Example: `[a-z-[aeiou]]` matches a single letter that is not a vowel.
 Supported by: JGsoft; .NET (2.0–7.0); XML; XPath

Feature: Character class intersection
 Syntax: `[base&&[intersect]]`
 Description: Reduces the character class to the characters present in both “base” and “intersect”.
 Example: `[a-z&&[^aeiou]]` matches a single letter that is not a vowel.
 Supported by: JGsoft (V2); Java; Ruby (1.9)

Feature: Character class intersection
 Syntax: `[base&&intersect]`
 Description: Reduces the character class to the characters present in both “base” and “intersect”.
 Example: `[\p{Nd}&&\p{InThai}]` matches a single Thai digit.
 Supported by: Java; Ruby (1.9)

Feature: Character escape
 Syntax: `\n`, `\r` and `\t`
 Description: Add an LF character, a CR character, or a tab character to the character class, respectively.
 Example: `[\n\r\t]` a line feed, a carriage return, or a tab.
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex (ECMA; awk); Boost (ECMA; awk); Tcl ARE; POSIX BRE (string); POSIX ERE (string); GNU BRE (string); GNU ERE (string); XML; XPath

- Feature: Character escape
 Syntax: `\a`
 Description: Add the “alert” or “bell” control character (ASCII 0x07) to the character class.
 Example: `[\a\t]` matches a bell or a tab character.
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; Python; Ruby; std::regex (awk); Boost (ECMA; awk); Tcl ARE
- Feature: Character escape
 Syntax: `\b`
 Description: Add the “backspace” control character (ASCII 0x08) to the character class.
 Example: `[\b\t]` matches a backspace or a tab character.
 Supported by: JGsoft; .NET; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex (ECMA VC'12–VC'15; awk VC'08–VC'22); Boost (ECMA; awk); Tcl ARE; XML; XPath
- Feature: Character escape
 Syntax: `\B`
 Description: Add a backslash to the character class.
 Example: `[\B]` matches `\`
 Supported by: Tcl ARE
- Feature: Character escape
 Syntax: `\e`
 Description: Add the “escape” control character (ASCII 0x1B) to the character class.
 Example: `[\e\t]` matches an escape or a tab character.
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; Ruby; Boost (ECMA; awk); Tcl ARE
- Feature: Character escape
 Syntax: `\f`
 Description: Add the “form feed” control character (ASCII 0x0C) to the character class.
 Example: `[\f\t]` matches a form feed or a tab character.
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex (ECMA; awk); Boost (ECMA; awk); Tcl ARE
- Feature: Character escape
 Syntax: `\v`
 Description: Add the “vertical tab” control character (ASCII 0x0B) to the character class, without adding any other vertical whitespace.
 Example: `[\v\t]` matches a vertical tab or a tab character.
 Supported by: JGsoft (V1 only); .NET; Java (4–7); JavaScript; VBScript; XRegExp; Python; Ruby; std::regex (ECMA; awk); Boost (ECMA; awk); Tcl ARE

- Feature:** POSIX class
Syntax: `[:alpha:]`
Description: Matches one character from a POSIX character class. Can only be used in a bracket expression.
Example: `[[:digit:]][:lower:]]` matches one of `0` through `9` or `a` through `z`
Supported by: JGsoft (ASCII); Perl (Unicode); PCRE (ASCII); PCRE2 (ASCII); PHP (5.3.4 Unicode 5.0.0 code page); Delphi (ASCII); R (ASCII); Ruby (1.9 Unicode 1.8 ASCII); std::regex (Unicode); Boost (Unicode); Tcl ARE (Unicode); POSIX BRE (ASCII); POSIX ERE (ASCII); GNU BRE (ASCII); GNU ERE (ASCII); Oracle (Unicode)
- Feature:** POSIX class
Syntax: `[:^alpha:]`
Description: Matches one character that is not part of a specific POSIX character class. Can only be used in a bracket expression.
Example: `[5[:^digit:]]` matches the digit `5` or any other character that is not a digit.
Supported by: Perl; PCRE; PCRE2; PHP; Delphi; R; Python (3.7–3.10 error); Ruby (1.9); std::regex (error); Boost; Tcl ARE (error); POSIX BRE (error); POSIX ERE (error); GNU BRE (error); GNU ERE (error); Oracle (error)
- Feature:** POSIX shorthand class
Syntax: `[:d:]`, `[:s:]`, `[:w:]`
Description: Matches one character from the POSIX character classes “digit”, “space”, or “word”. Can only be used in a bracket expression.
Example: `[[:s:]][:d:]]` matches a space, a tab, a line break, or one of `0` through `9`
Supported by: JGsoft (V2 ASCII); std::regex (Unicode); Boost (Unicode)
- Feature:** POSIX shorthand class
Syntax: `[:l:]` and `[:u:]`
Description: Matches one character from the POSIX character classes “lower” or “upper”. Can only be used in a bracket expression.
Example: `[[:u:]][[:l:]]` matches `Aa` but not `aA`.
Supported by: JGsoft (V2 ASCII); Boost (Unicode)
- Feature:** POSIX shorthand class
Syntax: `[:h:]`
Description: Matches one character from the POSIX character classes “blank”. Can only be used in a bracket expression.
Example: `[[:h:]]` matches a space.
Supported by: JGsoft (V2 ASCII); Boost (1.42–1.83; Unicode)
- Feature:** POSIX shorthand class
Syntax: `[:V:]`
Description: Matches a vertical whitespace character. Can only be used in a bracket expression.
Example: `[[:v:]]` match any single vertical whitespace character.
Supported by: JGsoft (V2 ASCII); Boost (1.42–1.83; Unicode)
- Feature:** POSIX class
Syntax: Any supported `\p{...}` syntax
Description: `\p{...}` syntax can be used inside character classes.
Example: `[:\p{Digit}\p{Lower}]` matches one of `0` through `9` or `a` through `z`
Supported by: JGsoft; Java (9); Perl; Ruby (1.9); Boost (extended; egrep)

Feature: POSIX class
 Syntax: `\p{Alpha}`
 Description: Matches one character from a POSIX character class.
 Example: `\p{Digit}` matches any single digit.
 Supported by: JGsoft (Unicode); Java (ASCII); Perl (Unicode); Ruby (1.9 Unicode); Boost (ECMA; extended; egrep; awk; Unicode)

Feature: POSIX class
 Syntax: `\p{IsAlpha}`
 Description: Matches one character from a POSIX character class.
 Example: `\p{IsDigit}` matches any single digit.
 Supported by: JGsoft (Unicode); Java (9 Unicode 4 ASCII); Perl (Unicode)

Feature: POSIX collation sequence
 Syntax: `[.span-ll.]`
 Description: Matches a POSIX collation sequence. Can only be used in a bracket expression.
 Example: `[.span-ll.]` matches `ll` in the Spanish locale
 Supported by: JGsoft (error); Perl (error); PCRE (error); PCRE2 (error); PHP (error); Delphi (error); R (error); Python (3.7–3.10 error); Ruby (1.8 only error); std::regex (fail); Boost; Tcl ARE; POSIX BRE; POSIX ERE; GNU BRE; GNU ERE; Oracle

Feature: POSIX character equivalence
 Syntax: `[=x=]`
 Description: Matches a POSIX character equivalence. Can only be used in a bracket expression.
 Example: `[=e=]` matches `e`, `é`, `è` and `ê` in the French locale
 Supported by: JGsoft (error); Perl (error); PCRE (error); PCRE2 (error); PHP (error); Delphi (error); R (error); Python (3.7–3.10 error); Ruby (1.8 only error); std::regex; Boost; Tcl ARE; POSIX BRE; POSIX ERE; GNU BRE; GNU ERE; Oracle

5. Shorthand Character Classes

Feature: Shorthand
 Syntax: Any shorthand outside character classes
 Description: Shorthands can be used outside character classes.
 Example: `\w` matches a single word character
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex (ECMA); Boost (ECMA; extended; egrep; awk); Tcl ARE; GNU BRE; GNU ERE; Oracle (10gR2); XML; XPath

Feature: Shorthand
 Syntax: Any shorthand inside a character class
 Description: Shorthands can be used inside character classes.
 Example: `[\w]` matches a single word character
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex (ECMA); Boost; Tcl ARE; XML; XPath

Feature: Shorthand
 Syntax: Any negated shorthand inside a character class
 Description: Negated shorthands can be used inside character classes.
 Example: `[\W]` matches a single character that is not a word character
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex (ECMA); Boost; Tcl ARE (error); XML; XPath

Feature: Shorthand
 Syntax: `\d`
 Description: Adds all digits to the character class. Matches a single digit if used outside character classes.
 Example: `[\d]` and/or `\d` match a character that is a digit
 Supported by: JGsoft (Unicode); .NET (non-ECMA Unicode); Java (ASCII); Perl (Unicode); PCRE (ASCII); PCRE2 (ASCII); PHP (5.3.4 Unicode 5.0.0 ASCII); Delphi (ASCII); R (ASCII); JavaScript (ASCII); VBScript (ASCII); XRegExp (ASCII); Python (3.0 Unicode 2.4 ASCII); Ruby (ASCII); std::regex (ECMA Unicode); Boost (Unicode); Tcl ARE (Unicode); Oracle (10gR2 Unicode); XML (Unicode); XPath (Unicode)

Feature: Shorthand
 Syntax: `\w`
 Description: Adds all word characters to the character class. Matches a single word character if used outside character classes.
 Example: `[\w]` and/or `\w` match any single word character
 Supported by: JGsoft (Unicode); .NET (non-ECMA Unicode); Java (ASCII); Perl (Unicode); PCRE (ASCII); PCRE2 (ASCII); PHP (5.3.4 Unicode 5.0.0 code page); Delphi (ASCII); R (ASCII); JavaScript (ASCII); VBScript (ASCII); XRegExp (ASCII); Python (3.0 Unicode 2.4 ASCII); Ruby (ASCII); std::regex (ECMA Unicode); Boost (Unicode); Tcl ARE (Unicode); GNU BRE (ASCII); GNU ERE (ASCII); Oracle (10gR2 Unicode); XML (Unicode); XPath (Unicode)

Feature: Shorthand
 Syntax: `\s`
 Description: Adds all whitespace to the character class. Matches a single whitespace character if used outside character classes.
 Example: `[\s]` and/or `\s` match any single whitespace character
 Supported by: JGsoft (Unicode); .NET (non-ECMA Unicode); Java (ASCII); Perl (Unicode); PCRE (ASCII); PCRE2 (ASCII); PHP (5.3.4 Unicode 5.0.0 code page); Delphi (ASCII); R (ASCII); JavaScript (Unicode); VBScript (ASCII); XRegExp (Unicode); Python (3.0 Unicode 2.4 ASCII); Ruby (ASCII); std::regex (ECMA Unicode); Boost (Unicode); Tcl ARE (Unicode); GNU BRE (ASCII); GNU ERE (ASCII); Oracle (10gR2 Unicode); XML (ASCII); XPath (ASCII)

Feature: Shorthand
 Syntax: `\l` and `\u`
 Description: Adds all lowercase letters or all uppercase letters to the character class. Matches a single lowercase or uppercase letter if used outside character classes.
 Example: `\u\l` matches `Aa` but not `aA`.
 Supported by: JGsoft (V2 Unicode); Boost (Unicode)

Feature: Shorthand
 Syntax: `\v`
 Description: Adds all vertical whitespace to the character class. Matches a single vertical whitespace character if used outside character classes.
 Example: `[\v]` and/or `\v` match any single vertical whitespace character
 Supported by: JGsoft (V2 Unicode); Java (8 Unicode); Perl (5.10 Unicode); PCRE (7.2 Unicode); PCRE2 (Unicode); PHP (5.2.4 Unicode); Delphi (Unicode); R (Unicode); Boost (ECMA; 1.42–1.83 Unicode)

Feature: Shorthand
 Syntax: `\h`
 Description: Adds all horizontal whitespace to the character class. Matches a single horizontal whitespace character if used outside character classes.
 Example: `[\h]` and/or `\h` match any single horizontal whitespace character
 Supported by: JGsoft (V2 Unicode); Java (8 Unicode); Perl (5.10 Unicode); PCRE (7.2 Unicode); PCRE2 (Unicode); PHP (5.2.4 Unicode); Delphi (Unicode); R (Unicode); Boost (1.42–1.83 Unicode)

Feature: Shorthand
 Syntax: `\h`
 Description: Adds all hexadecimal digits to the character class. Matches a hexadecimal digit if used outside character classes.
 Example: `[\h]` and/or `\h` match any single hexadecimal digit
 Supported by: Ruby (1.9 ASCII)

Feature: XML Shorthand
 Syntax: `\i`
 Description: Adds all characters that are allowed as the initial character in XML names to the character class. Matches one such character if used outside character classes.
 Example: `\i\c*` matches an XML name
 Supported by: JGsoft (V2); XML; XPath

Feature: XML Shorthand
Syntax: `\c`
Description: Adds all characters that are allowed as the second and following characters in XML names to the character class. Matches one such character if used outside character classes.
Example: `\i\c*` matches an XML name
Supported by: JGsoft (V2); XML; XPath

6. Anchors

Feature: String anchor
 Syntax: `^` (caret)
 Description: Matches at the start of the string the regex pattern is applied to.
 Example: `^.` matches `a` in `abc\ndef`
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex; Boost; Tcl ARE; POSIX BRE; POSIX ERE; GNU BRE; GNU ERE; Oracle; XPath

Feature: String anchor
 Syntax: `$` (dollar)
 Description: Matches at the end of the string the regex pattern is applied to.
 Example: `.$` matches `f` in `abc\ndef`
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex; Boost; Tcl ARE; POSIX BRE; POSIX ERE; GNU BRE; GNU ERE; Oracle; XPath

Feature: String anchor
 Syntax: `$` (dollar)
 Description: Matches before the final line break in the string (if any) in addition to matching at the very end of the string.
 Example: `.$` matches `f` in `abc\ndef\n`
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; Python

Feature: Line anchor
 Syntax: `^` (caret)
 Description: Matches after each line break in addition to matching at the start of the string, thus matching at the start of each line in the string.
 Example: `^.` matches `a` and `d` in `abc\ndef`
 Supported by: JGsoft; .NET (option); Java (option); Perl (option); PCRE (option); PCRE2 (option); PHP (option); Delphi (option); R (option); JavaScript (option); VBScript (option); XRegExp (option); Python (option); Ruby; std::regex; Boost (basic; extended; grep; egrep; awk); Tcl ARE (option); POSIX BRE (option); POSIX ERE (option); GNU BRE (option); GNU ERE (option); Oracle (option); XPath (option)

Feature: Line anchor
 Syntax: `$` (dollar)
 Description: Matches before each line break in addition to matching at the end of the string, thus matching at the end of each line in the string.
 Example: `.$` matches `c` and `f` in `abc\ndef`
 Supported by: JGsoft; .NET (option); Java (option); Perl (option); PCRE (option); PCRE2 (option); PHP (option); Delphi (option); R (option); JavaScript (option); VBScript (option); XRegExp (option); Python (option); Ruby; std::regex; Boost (basic; extended; grep; egrep; awk); Tcl ARE (option); POSIX BRE (option); POSIX ERE (option); GNU BRE (option); GNU ERE (option); Oracle (option); XPath (option)

Feature: String anchor
 Syntax: `\A`
 Description: Matches at the start of the string the regex pattern is applied to.
 Example: `\A\w` matches only `a` in `abc`
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; Python; Ruby; Boost (ECMA; extended; egrep; awk); Oracle (10gR2)

Feature: Attempt anchor
 Syntax: `\A`
 Description: Matches at the start of the match attempt.
 Example: `\A\w` matches `a`, `b`, and `c` when iterating over all matches in `abc def`
 Supported by: Tcl ARE

Feature: Attempt anchor
 Syntax: `\G`
 Description: Matches at the start of the match attempt.
 Example: `\G\w` matches `a`, `b`, and `c` when iterating over all matches in `abc def`
 Supported by: JGsoft; PCRE (4.0–7.9); PHP; Delphi; R; Ruby

Feature: Match anchor
 Syntax: `\G`
 Description: Matches at the end of the previous match during the second and following match attempts. Matches at the start of the string during the first match attempt.
 Example: `\G\w` matches `a`, `b`, and `c` when iterating over all matches in `abc def`
 Supported by: .NET; Java; Perl; PCRE (8.00); PCRE2; Boost (ECMA; extended; egrep; awk)

Feature: String anchor
 Syntax: `\z`
 Description: Matches at the end of the string the regex pattern is applied to.
 Example: `\w\z` matches `f` in `abc\ndef` but fails to match `abc\ndef\n`
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; Ruby; Boost (ECMA; extended; egrep; awk); Oracle (10gR2)

Feature: String anchor
 Syntax: `\Z`
 Description: Matches at the end of the string the regex pattern is applied to.
 Example: `\w\Z` matches `f` in `abc\ndef` but fails to match `abc\ndef\n` or `abc\ndef\n\n`
 Supported by: Python; Tcl ARE

Feature: String anchor
 Syntax: `\Z`
 Description: Matches at the end of the string as well as before the final line break in the string (if any).
 Example: `.\Z` matches `f` in `abc\ndef` and in `abc\ndef\n` but fails to match `abc\ndef\n\n`
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; Ruby; Oracle (10gR2)

Feature: String anchor
 Syntax: `\Z`
 Description: Matches at the end of the string as well as before all trailing line breaks in the string (if any).
 Example: `.\Z` matches `f` in `abc\ndef` and in `abc\ndef\n` and in `abc\ndef\n\n`
 Supported by: Boost (ECMA; extended; egrep; awk)

Feature: String anchor
Syntax: `\`` (backslash backtick)
Description: Matches at the start of the string the regex pattern is applied to.
Example: `\`w` matches only `a` in `abc`
Supported by: Boost (ECMA; extended; egrep; awk)

Feature: Attempt anchor
Syntax: `\`` (backslash backtick)
Description: Matches at the start of the match attempt.
Example: `\`w` matches `a`, `b`, and `c` when iterating over all matches in `abc def`
Supported by: GNU BRE; GNU ERE

Feature: String anchor
Syntax: `\'` (backslash quote)
Description: Matches at the end of the string the regex pattern is applied to.
Example: `w\'` matches `f` in `abc\ndef` but fails to match `abc\ndef\n`
Supported by: Boost (ECMA; extended; egrep; awk); GNU BRE; GNU ERE

7. Word Boundaries

Feature: Word boundary
 Syntax: `\b`
 Description: Matches at a position that is followed by a word character but not preceded by a word character, or that is preceded by a word character but not followed by a word character.
 Example: `\b` matches `a`, , and `d` in `abc def`
 Supported by: JGsoft (Unicode); .NET (non-ECMA Unicode); Java (ASCII); Perl (Unicode); PCRE (ASCII); PCRE2 (ASCII); PHP (5.3.4 Unicode 5.0.0 code page); Delphi (ASCII); R (ASCII); JavaScript (ASCII); VBScript (ASCII); XRegExp (ASCII); Python (3.0 Unicode 2.4 ASCII); Ruby (Unicode); std::regex (ECMA ASCII); Boost (ECMA; extended; egrep; awk; Unicode); GNU BRE (ASCII); GNU ERE (ASCII)

Feature: Word boundary
 Syntax: `\B`
 Description: Matches at a position that is preceded and followed by a word character, or that is not preceded and not followed by a word character.
 Example: `\B` matches `b`, `c`, `e`, and `f` in `abc def`
 Supported by: JGsoft (Unicode); .NET (non-ECMA Unicode); Java (ASCII); Perl (Unicode); PCRE (ASCII); PCRE2 (ASCII); PHP (5.3.4 Unicode 5.0.0 code page); Delphi (ASCII); R (ASCII); JavaScript (ASCII); VBScript (ASCII); XRegExp (ASCII); Python (3.0 Unicode 2.4 ASCII); Ruby (Unicode); std::regex (ECMA ASCII); Boost (ECMA; extended; egrep; awk; Unicode); GNU BRE (ASCII); GNU ERE (ASCII)

Feature: Tcl word boundary
 Syntax: `\y`
 Description: Matches at a position that is followed by a word character but not preceded by a word character, or that is preceded by a word character but not followed by a word character.
 Example: `\y` matches `a`, , and `d` in `abc def`
 Supported by: JGsoft (Unicode); Tcl ARE (Unicode)

Feature: Tcl word boundary
 Syntax: `\Y`
 Description: Matches at a position that is preceded and followed by a word character, or that is not preceded and not followed by a word character.
 Example: `\Y` matches `b`, `c`, `e`, and `f` in `abc def`
 Supported by: JGsoft (Unicode); Tcl ARE (Unicode)

Feature: Tcl word boundary
 Syntax: `\m`
 Description: Matches at a position that is followed by a word character but not preceded by a word character.
 Example: `\m` matches `a` and `d` in `abc def`
 Supported by: JGsoft (Unicode); Tcl ARE (Unicode)

Feature: Tcl word boundary
 Syntax: `\M`
 Description: Matches at a position that is preceded by a word character but not followed by a word character.

Example: `.\M` matches `c` and `f` in `abc def`
 Supported by: JGsoft (Unicode); Tcl ARE (Unicode)

Feature: GNU word boundary
 Syntax: `\<`
 Description: Matches at a position that is followed by a word character but not preceded by a word character.

Example: `\<.` matches `a` and `d` in `abc def`
 Supported by: Boost (ECMA; extended; egrep; awk; Unicode); GNU BRE (ASCII); GNU ERE (ASCII)

Feature: GNU word boundary
 Syntax: `\>`
 Description: Matches at a position that is preceded by a word character but not followed by a word character.

Example: `.\>` matches `c` and `f` in `abc def`
 Supported by: Boost (ECMA; extended; egrep; awk; Unicode); GNU BRE (ASCII); GNU ERE (ASCII)

Feature: POSIX word boundary
 Syntax: `[[[:<:]]`
 Description: Matches at a position that is followed by a word character but not preceded by a word character.

Example: `[[[:<:]]` matches `a` and `d` in `abc def`
 Supported by: PCRE (8.34 ASCII); PCRE2 (ASCII); PHP (5.5.10 Unicode); Delphi (XE7 ASCII); R (3.0.3 ASCII); Boost (Unicode); Tcl ARE (Unicode); POSIX BRE (ASCII); POSIX ERE (ASCII)

Feature: POSIX word boundary
 Syntax: `[[[:>:]]`
 Description: Matches at a position that is preceded by a word character but not followed by a word character.

Example: `[[[:>:]]` matches `c` and `f` in `abc def`
 Supported by: PCRE (8.34 ASCII); PCRE2 (ASCII); PHP (5.5.10 Unicode); Delphi (XE7 ASCII); R (3.0.3 ASCII); Boost (Unicode); Tcl ARE (Unicode); POSIX BRE (ASCII); POSIX ERE (ASCII)

Feature: Word boundary behavior
 Syntax:
 Description: Word boundaries always match at the start of the match attempt if that position is followed by a word character, regardless of the character that precedes the start of the match attempt. (Thus word boundaries are not handled correctly for the second and following match attempts in the same string.)

Example: `\b` matches all of the letters but not the space when iterating over all matches in the string `abc def`
 Supported by: Tcl ARE; GNU BRE; GNU ERE

8. Quantifiers

Feature: Greedy quantifier
 Syntax: `?` (question mark)
 Description: Makes the preceding item optional. Greedy, so the optional item is included in the match if possible.
 Example: `abc?` matches `abc` or `ab`
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; `std::regex` (ECMA; extended; `egrep`; `awk`); Boost (ECMA; extended; `egrep`; `awk`); Tcl ARE; POSIX ERE; GNU ERE; Oracle; XML; XPath

Feature: Greedy quantifier
 Syntax: `\?`
 Description: Makes the preceding item optional. Greedy, so the optional item is included in the match if possible.
 Example: `abc\?` matches `abc` or `ab`
 Supported by: GNU BRE

Feature: Lazy quantifier
 Syntax: `??`
 Description: Makes the preceding item optional. Lazy, so the optional item is excluded in the match if possible.
 Example: `abc??` matches `ab` or `abc`
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; `std::regex` (ECMA); Boost (ECMA); Tcl ARE; Oracle (10gR2); XPath

Feature: Possessive quantifier
 Syntax: `?+`
 Description: Makes the preceding item optional. Possessive, so if the optional item can be matched, then the quantifier won't give up its match even if the remainder of the regex fails.
 Example: `abc?+c` matches `abcc` but not `abc`
 Supported by: JGsoft; Java; Perl (5.10); PCRE; PCRE2; PHP; Delphi; R; Ruby (1.9); Boost (ECMA; 1.42–1.83)

Feature: Greedy quantifier
 Syntax: `*` (star)
 Description: Repeats the previous item zero or more times. Greedy, so as many items as possible will be matched before trying permutations with less matches of the preceding item, up to the point where the preceding item is not matched at all.
 Example: `".*"` matches `"def"` `"ghi"` in `abc "def" "ghi" jkl`
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; `std::regex`; Boost; Tcl ARE; POSIX BRE; POSIX ERE; GNU BRE; GNU ERE; Oracle; XML; XPath

Feature: Lazy quantifier
 Syntax: `*?`
 Description: Repeats the previous item zero or more times. Lazy, so the engine first attempts to skip the previous item, before trying permutations with ever increasing matches of the preceding item.
 Example: `".*?"` matches "def" and "ghi" in `abc "def" "ghi" jkl`
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex (ECMA); Boost (ECMA); Tcl ARE; Oracle (10gR2); XPath

Feature: Possessive quantifier
 Syntax: `*+`
 Description: Repeats the previous item zero or more times. Possessive, so as many items as possible will be matched, without trying any permutations with less matches even if the remainder of the regex fails.
 Example: `".*+"` can never match anything
 Supported by: JGsoft; Java; Perl (5.10); PCRE; PCRE2; PHP; Delphi; R; Ruby (1.9); Boost (ECMA; 1.42–1.83)

Feature: Greedy quantifier
 Syntax: `+` (plus)
 Description: Repeats the previous item once or more. Greedy, so as many items as possible will be matched before trying permutations with less matches of the preceding item, up to the point where the preceding item is matched only once.
 Example: `".+"` matches "def" "ghi" in `abc "def" "ghi" jkl`
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex (ECMA; extended; egrep; awk); Boost (ECMA; extended; egrep; awk); Tcl ARE; POSIX ERE; GNU ERE; Oracle; XML; XPath

Feature: Greedy quantifier
 Syntax: `\+`
 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`
 Supported by: GNU BRE

Feature: Lazy quantifier
 Syntax: `+?`
 Description: Repeats the previous item once or more. Lazy, so the engine first matches the previous item only once, before trying permutations with ever increasing matches of the preceding item.
 Example: `".+?"` matches "def" and "ghi" in `abc "def" "ghi" jkl`
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex (ECMA); Boost (ECMA); Tcl ARE; Oracle (10gR2); XPath

Feature: Possessive quantifier
 Syntax: `++`
 Description: Repeats the previous item once or more. Possessive, so as many items as possible will be matched, without trying any permutations with less matches even if the remainder of the regex fails.
 Example: `".++"` can never match anything
 Supported by: JGsoft; Java; Perl (5.10); PCRE; PCRE2; PHP; Delphi; R; Ruby (1.9); Boost (ECMA; 1.42–1.83)

Feature: Fixed quantifier
 Syntax: `{n}` where n is an integer ≥ 1
 Description: Repeats the previous item exactly n times.
 Example: `a{3}` matches `aaa`
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; `std::regex` (ECMA; extended; `egrep`; `awk`); Boost (ECMA; extended; `egrep`; `awk`); Tcl ARE; POSIX ERE; GNU ERE; Oracle; XML; XPath

Feature: Greedy quantifier
 Syntax: `{n,m}` where $n \geq 0$ and $m \geq n$
 Description: Repeats the previous item between n and m times. Greedy, so repeating m times is tried before reducing the repetition to n times.
 Example: `a{2,4}` matches `aaaa`, `aaa` or `aa`
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; `std::regex` (ECMA; extended; `egrep`; `awk`); Boost (ECMA; extended; `egrep`; `awk`); Tcl ARE; POSIX ERE; GNU ERE; Oracle; XML; XPath

Feature: Greedy quantifier
 Syntax: `{n,}` where $n \geq 0$
 Description: Repeats the previous item at least n times. Greedy, so as many items as possible will be matched before trying permutations with less matches of the preceding item, up to the point where the preceding item is matched only n times.
 Example: `a{2,}` matches `aaaaa` in `aaaaa`
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; `std::regex` (ECMA; extended; `egrep`; `awk`); Boost (ECMA; extended; `egrep`; `awk`); Tcl ARE; POSIX ERE; GNU ERE; Oracle; XML; XPath

Feature: Greedy quantifier
 Syntax: `{,m}` where $m \geq 1$
 Description: Repeats the previous item between zero and m times. Greedy, so repeating m times is tried before reducing the repetition to zero times.
 Example: `a{,4}` matches `aaaa`, `aaa`, `aa`, `a`, or the empty string
 Supported by: JGsoft (V2); Python; Ruby (1.9); GNU ERE

Feature: Fixed quantifier
 Syntax: `\{n\}` where n is an integer ≥ 1
 Description: Repeats the previous item exactly n times.
 Example: `a\{3\}` matches `aaa`
 Supported by: `std::regex` (basic; `grep`); Boost (basic; `grep`); POSIX BRE; GNU BRE

Feature: Greedy quantifier
 Syntax: $\{n, m\}$ where $n \geq 0$ and $m \geq n$
 Description: Repeats the previous item between n and m times. Greedy, so repeating m times is tried before reducing the repetition to n times.
 Example: `a{2,4}` matches `aaaa`, `aaa` or `aa`
 Supported by: `std::regex` (basic; grep); Boost (basic; grep); POSIX BRE; GNU BRE

Feature: Greedy quantifier
 Syntax: $\{n, \}$ where $n \geq 0$
 Description: Repeats the previous item at least n times. Greedy, so as many items as possible will be matched before trying permutations with less matches of the preceding item, up to the point where the preceding item is matched only n times.
 Example: `a{2,}` matches `aaaaa` in `aaaaa`
 Supported by: `std::regex` (basic; grep); Boost (basic; grep); POSIX BRE; GNU BRE

Feature: Greedy quantifier
 Syntax: $\{, m\}$ where $m \geq 1$
 Description: Repeats the previous item between zero and m times. Greedy, so repeating m times is tried before reducing the repetition to zero times.
 Example: `a{,4}` matches `aaaa`, `aaa`, `aa`, `a`, or the empty string
 Supported by: GNU BRE

Feature: Lazy quantifier
 Syntax: $\{n, m\}?$ where $n \geq 0$ and $m \geq n$
 Description: Repeats the previous item between n and m times. Lazy, so repeating n times is tried before increasing the repetition to m times.
 Example: `a{2,4}?` matches `aa`, `aaa` or `aaaa`
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; `std::regex` (ECMA); Boost (ECMA); Tcl ARE; Oracle (10gR2); XPath

Feature: Lazy quantifier
 Syntax: $\{n, \}?$ where $n \geq 0$
 Description: Repeats the previous item n or more times. Lazy, so the engine first matches the previous item n times, before trying permutations with ever increasing matches of the preceding item.
 Example: `a{2,}?` matches `aa` in `aaaaa`
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; `std::regex` (ECMA); Boost (ECMA); Tcl ARE; Oracle (10gR2); XPath

Feature: Lazy quantifier
 Syntax: $\{, m\}?$ where $m \geq 1$
 Description: Repeats the previous item between zero and m times. Lazy, so repeating zero times is tried before increasing the repetition to m times.
 Example: `a{,4}?` matches the empty string, `a`, `aa`, `aaa` or `aaaa`
 Supported by: JGsoft (V2); Python; Ruby (1.9)

Feature: Possessive quantifier
Syntax: $\{n, m\}+$ where $n \geq 0$ and $m \geq n$
Description: Repeats the previous item between n and m times. Possessive, so as many items as possible up to m will be matched, without trying any permutations with less matches even if the remainder of the regex fails.
Example: `a{2,4}+a` matches `aaaaa` but not `aaaa`
Supported by: JGsoft; Java; Perl (5.10); PCRE; PCRE2; PHP; Delphi; R; Boost (ECMA; 1.42–1.83)

Feature: Possessive quantifier
Syntax: $\{n, \}+$ where $n \geq 0$
Description: Repeats the previous item n or more times. Possessive, so as many items as possible will be matched, without trying any permutations with less matches even if the remainder of the regex fails.
Example: `a{2,}+a` never matches anything
Supported by: JGsoft; Java; Perl (5.10); PCRE; PCRE2; PHP; Delphi; R; Boost (ECMA; 1.42–1.83)

9. Unicode Syntax Reference

This reference page explains what the Unicode tokens do when used outside character classes. All of these except `\X` can also be used inside character classes. Inside a character class, these tokens add the characters that they normally match to the character class.

Feature: Grapheme
 Syntax: `\X`
 Description: Matches a single Unicode grapheme, whether encoded as a single code point or multiple code points using combining marks. A grapheme most closely resembles the everyday concept of a “character”.
 Example: `\X` matches `ā` encoded as U+0061 U+0300, `ā` encoded as U+00E0, `©`, etc.
 Supported by: JGsoft; Java (9); Perl; PCRE (5.0); PCRE2; PHP (5.0.5); Delphi; R; Ruby (2.0); Boost (ECMA; extended; egrep; awk)

Feature: Code point
 Syntax: `\uFFFF` where FFFF are 4 hexadecimal digits
 Description: Matches a specific Unicode code point.
 Example: `\u00E0` matches `ā` encoded as U+00E0 only. `\u00A9` matches `©`
 Supported by: JGsoft; .NET; Java; JavaScript; VBScript; XRegExp; Python (3.3; 2.4 string); Ruby (1.9); std::regex (ECMA); Tcl ARE

Feature: Code point
 Syntax: `\u{FFFF}` where FFFF are 1 to 4 hexadecimal digits
 Description: Matches a specific Unicode code point.
 Example: `\u{E0}` matches `ā` encoded as U+00E0 only. `\u{A9}` matches `©`
 Supported by: JGsoft (V2); PHP (7.0.0 string); XRegExp (3); Ruby (1.9)

Feature: Code point
 Syntax: `\xFFFF` where FFFF are 4 hexadecimal digits
 Description: Matches a specific Unicode code point.
 Example: `\x00E0` matches `ā` encoded as U+00E0 only. `\x00A9` matches `©`
 Supported by: std::regex (string); Tcl ARE (8.4–8.5)

Feature: Code point
 Syntax: `\x{FFFF}` where FFFF are 1 to 4 hexadecimal digits
 Description: Matches a specific Unicode code point.
 Example: `\x{E0}` matches `ā` encoded as U+00E0 only. `\x{A9}` matches `©`
 Supported by: JGsoft; Java (7); Perl; PCRE; PCRE2; PHP; Delphi; R; Boost (ECMA; extended; egrep; awk)

Feature: Unicode category
 Syntax: `\pL` where L is a Unicode category
 Description: Matches a single Unicode code point in the specified Unicode category.
 Example: `\pL` matches `ā` encoded as U+00E0; `\pS` matches `©`
 Supported by: JGsoft; Java; Perl; PCRE (5.0); PCRE2; PHP (5.0.5); Delphi; R; XRegExp (3)

Feature: Unicode category
 Syntax: `\PL` where L is a Unicode category
 Description: Matches a single Unicode code point that is *not* in the specified Unicode category.
 Example: `\PS` matches `à` encoded as U+00E0; `\PL` matches `©`
 Supported by: JGsoft; Java; Perl; PCRE (5.0); PCRE2; PHP (5.0.5); Delphi; R; XRegExp (3)

Feature: Unicode category
 Syntax: `\p{L}` where L is a Unicode category
 Description: Matches a single Unicode code point in the specified Unicode category.
 Example: `\p{L}` matches `à` encoded as U+00E0; `\p{S}` matches `©`
 Supported by: JGsoft; .NET; Java; Perl; PCRE (5.0); PCRE2; PHP (5.0.5); Delphi; R; XRegExp; Ruby (1.9); XML; XPath

Feature: Unicode category
 Syntax: `\p{IsL}` where L is a Unicode category
 Description: Matches a single Unicode code point in the specified Unicode category.
 Example: `\p{IsL}` matches `à` encoded as U+00E0; `\p{IsS}` matches `©`
 Supported by: JGsoft; Java; Perl

Feature: Unicode category
 Syntax: `\p{Category}`
 Description: Matches a single Unicode code point in the specified Unicode category.
 Example: `\p{Letter}` matches `à` encoded as U+00E0; `\p{Symbol}` matches `©`
 Supported by: JGsoft; Perl; XRegExp; Ruby (1.9)

Feature: Unicode category
 Syntax: `\p{IsCategory}`
 Description: Matches a single Unicode code point in the specified Unicode category.
 Example: `\p{IsLetter}` matches `à` encoded as U+00E0; `\p{IsSymbol}` matches `©`
 Supported by: JGsoft; Perl

Feature: Unicode script
 Syntax: `\p{Script}`
 Description: Matches a single Unicode code point that is part of the specified Unicode script. Each Unicode code point is part of exactly one script. Scripts never contain unassigned code points.
 Example: `\p{Greek}` matches `Ω`
 Supported by: JGsoft; Perl; PCRE (6.5); PCRE2; PHP (5.1.3); Delphi; R; XRegExp; Ruby (1.9)

Feature: Unicode script
 Syntax: `\p{IsScript}`
 Description: Matches a single Unicode code point that is part of the specified Unicode script. Each Unicode code point is part of exactly one script. Scripts never contain unassigned code points.
 Example: `\p{IsGreek}` matches `Ω`
 Supported by: JGsoft; Java (7); Perl

Feature: Unicode block
 Syntax: `\p{Block}`
 Description: Matches a single Unicode code point that is part of the specified Unicode block. Each Unicode code point is part of exactly one block. Blocks may contain unassigned code points.
 Example: `\p{Arrows}` matches any of the code points from U+2190 until U+21FF (← until ↔)
 Supported by: JGsoft; Perl

Feature: Unicode block
 Syntax: `\p{InBlock}`
 Description: Matches a single Unicode code point that is part of the specified Unicode block. Each Unicode code point is part of exactly one block. Blocks may contain unassigned code points.
 Example: `\p{InArrows}` matches any of the code points from U+2190 until U+21FF (← until ↔)
 Supported by: JGsoft; Java; Perl; XRegExp (2–4); Ruby (2.0)

Feature: Unicode block
 Syntax: `\p{IsBlock}`
 Description: Matches a single Unicode code point that is part of the specified Unicode block. Each Unicode code point is part of exactly one block. Blocks may contain unassigned code points.
 Example: `\p{IsArrows}` matches any of the code points from U+2190 until U+21FF (← until ↔)
 Supported by: JGsoft; .NET; Perl; XML; XPath

Feature: Negated Unicode property
 Syntax: `\P{Property}`
 Description: Matches a single Unicode code point that does *not* have the specified property (category, script, or block).
 Example: `\P{L}` matches ©
 Supported by: JGsoft; .NET; Java; Perl; PCRE (5.0); PCRE2; PHP (5.0.5); Delphi; R; XRegExp; Ruby (1.9); Boost (ECMA; extended; egrep; awk); XML; XPath

Feature: Negated Unicode property
 Syntax: `\p{^Property}`
 Description: Matches a single Unicode code point that does *not* have the specified property (category, script, or block).
 Example: `\p{^L}` matches ©
 Supported by: JGsoft; Perl; PCRE (5.0); PCRE2; PHP (5.0.5); Delphi; R; XRegExp; Ruby (1.9)

Feature: Unicode property
 Syntax: `\P{^Property}`
 Description: Matches a single Unicode code point that *does have* the specified property (category, script, or block). Double negative is taken as positive.
 Example: `\P{^L}` matches q
 Supported by: JGsoft (V2); Perl; PCRE (5.0); PCRE2; PHP (5.0.5); Delphi; R; Ruby (1.9)

10. Capturing Groups and Backreferences

Feature: Capturing group
 Syntax: (regex)
 Description: Parentheses group the regex between them. They capture the text matched by the regex inside them into a numbered group that can be reused with a numbered backreference. They allow you to apply regex operators to the entire grouped regex.
 Example: (abc){3} matches abcabcabc. First group matches abc.
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex (ECMA; extended; egrep; awk); Boost (ECMA; extended; egrep; awk); Tcl ARE; POSIX ERE; GNU ERE; Oracle; XML; XPath

Feature: Capturing group
 Syntax: \ (regex\
 Description: Escaped parentheses group the regex between them. They capture the text matched by the regex inside them into a numbered group that can be reused with a numbered backreference. They allow you to apply regex operators to the entire grouped regex.
 Example: \ (abc\
 Supported by: std::regex (basic; grep); Boost (basic; grep); POSIX BRE; GNU BRE

Feature: Non-capturing group
 Syntax: (? : regex)
 Description: Non-capturing parentheses group the regex so you can apply regex operators, but do not capture anything.
 Example: (? : abc){3} matches abcabcabc. No groups.
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex (ECMA); Boost (ECMA); Tcl ARE; XPath

Feature: Backreference
 Syntax: \1 through \9
 Description: Substituted with the text matched between the 1st through 9th numbered capturing group.
 Example: (abc|def)=\1 matches abc=abc or def=def, but not abc=def or def=abc.
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex (ECMA; basic; grep); Boost (ECMA; basic; grep); Tcl ARE; POSIX BRE; GNU BRE; GNU ERE; Oracle; XPath

Feature: Backreference
 Syntax: \10 through \99
 Description: Substituted with the text matched between the 10th through 99th numbered capturing group.
 Example:
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex (ECMA); Tcl ARE; XPath

Feature: Backreference
 Syntax: \k<1> through \k<99>
 Description: Substituted with the text matched between the 1st through 99th numbered capturing group.
 Example: (abc|def)=\k<1> matches abc=abc or def=def, but not abc=def or def=abc.
 Supported by: JGsoft; .NET; XRegExp; Ruby (1.9); Boost (ECMA; 1.47–1.83)

Feature: Backreference
 Syntax: `\k'1'` through `\k'99'`
 Description: Substituted with the text matched between the 1st through 99th numbered capturing group.
 Example: `(abc|def)=\k'1'` matches `abc=abc` or `def=def`, but not `abc=def` or `def=abc`.
 Supported by: JGsoft; .NET; Ruby (1.9); Boost (ECMA; 1.47–1.83)

Feature: Backreference
 Syntax: `\g1` through `\g99`
 Description: Substituted with the text matched between the 1st through 99th numbered capturing group.
 Example: `(abc|def)=\g1` matches `abc=abc` or `def=def`, but not `abc=def` or `def=abc`.
 Supported by: Perl (5.10); PCRE (7.0); PCRE2; PHP (5.2.2); Delphi; R; Boost (ECMA; 1.42–1.83)

Feature: Backreference
 Syntax: `\g{1}` through `\g{99}`
 Description: Substituted with the text matched between the 1st through 99th numbered capturing group.
 Example: `(abc|def)=\g{1}` matches `abc=abc` or `def=def`, but not `abc=def` or `def=abc`.
 Supported by: Perl (5.10); PCRE (7.0); PCRE2; PHP (5.2.2); Delphi; R; Boost (ECMA; 1.42–1.83)

Feature: Backreference
 Syntax: `\g<1>` through `\g<99>`
 Description: Substituted with the text matched between the 1st through 99th numbered capturing group.
 Example: `(abc|def)=\g<1>` matches `abc=abc` or `def=def`, but not `abc=def` or `def=abc`.
 Supported by: Boost (ECMA; 1.47–1.83)

Feature: Backreference
 Syntax: `\g'1'` through `\g'99'`
 Description: Substituted with the text matched between the 1st through 99th numbered capturing group.
 Example: `(abc|def)=\g'1'` matches `abc=abc` or `def=def`, but not `abc=def` or `def=abc`.
 Supported by: Boost (ECMA; 1.47–1.83)

Feature: Backreference
 Syntax: `(?P=1)` through `(?P=99)`
 Description: Substituted with the text matched between the 1st through 99th numbered capturing group.
 Example: `(abc|def)=(?P=1)` matches `abc=abc` or `def=def`, but not `abc=def` or `def=abc`.
 Supported by: JGsoft

Feature: Relative Backreference
 Syntax: `\k<-1>`, `\k<-2>`, etc.
 Description: Substituted with the text matched by the capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from right to left starting at the backreference.
 Example: `(a)(b)(c)(d)\k<-3>` matches `abcdb`.
 Supported by: JGsoft (V2); Ruby (1.9); Boost (ECMA; 1.47–1.83)

Feature: Relative Backreference
 Syntax: `\k'-1'`, `\k'-2'`, etc.
 Description: Substituted with the text matched by the capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from right to left starting at the backreference.
 Example: `(a)(b)(c)(d)\k'-3'` matches `abcdb`.
 Supported by: JGsoft (V2); Ruby (1.9); Boost (ECMA; 1.47–1.83)

Feature: Relative Backreference
 Syntax: `\g-1`, `\g-2`, etc.
 Description: Substituted with the text matched by the capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from right to left starting at the backreference.
 Example: `(a)(b)(c)(d)\g-3` matches `abcdb`.
 Supported by: Perl (5.10); PCRE (7.0); PCRE2; PHP (5.2.2); Delphi; R; Boost (ECMA; 1.42–1.83)

Feature: Relative Backreference
 Syntax: `\g{-1}`, `\g{-2}`, etc.
 Description: Substituted with the text matched by the capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from right to left starting at the backreference.
 Example: `(a)(b)(c)(d)\g{-3}` matches `abcdb`.
 Supported by: Perl (5.10); PCRE (7.0); PCRE2; PHP (5.2.2); Delphi; R; Boost (ECMA; 1.42–1.83)

Feature: Relative Backreference
 Syntax: `\g<-1>`, `\g<-2>`, etc.
 Description: Substituted with the text matched by the capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from right to left starting at the backreference.
 Example: `(a)(b)(c)(d)\g<-3>` matches `abcdb`.
 Supported by: Boost (ECMA; 1.47–1.83)

Feature: Relative Backreference
 Syntax: `\g'-1'`, `\g'-2'`, etc.
 Description: Substituted with the text matched by the capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from right to left starting at the backreference.
 Example: `(a)(b)(c)(d)\g'-3'` matches `abcdb`.
 Supported by: Boost (ECMA; 1.47–1.83)

Feature: Failed backreference
 Syntax: Any numbered backreference
 Description: Backreferences to groups that did not participate in the match attempt fail to match.
 Example: `(a)?\1` matches `aa` but fails to match `b`.
 Supported by: JGsoft; .NET (non-ECMA); Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript (ignored); VBScript (ignored); XRegExp (ignored); Python; Ruby; std::regex (ECMA; basic; grep; ignored); Boost (ECMA; 1.47–1.83); Tcl ARE; POSIX BRE; GNU BRE; GNU ERE; Oracle; XPath (ignored)

Feature: Invalid backreference
 Syntax: Any numbered backreference
 Description: Backreferences to groups that do not exist at all are valid but fail to match anything.
 Example: `(a)?\2|b` matches `b` in `aab`.
 Supported by: JGsoft (error); .NET (error); Java; Perl (error); PCRE (error); PCRE2 (error); PHP (error); Delphi (error); R (error); JavaScript (error); VBScript (error); XRegExp (error); Python (error); Ruby (1.8 only); std::regex (ECMA; basic; grep; error); Boost (ECMA; basic; grep; error); Tcl ARE (error); POSIX BRE (error); GNU BRE (error); GNU ERE (error); Oracle (error); XPath (error)

Feature: Nested backreference
Syntax: Any numbered backreference
Description: Backreferences can be used inside the group they reference.
Example: `(a\1?){3}` matches `aaaaaa`.
Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript (ignored); VBScript; XRegExp (ignored); Python (error); Ruby (fail); std::regex (ECMA; basic; grep; error); Boost (ECMA; 1.78–1.83 fail); Tcl ARE (error); POSIX BRE (error); GNU BRE (error); GNU ERE (error); Oracle (error); XPath (error)

Feature: Forward reference
Syntax: Any numbered backreference
Description: Backreferences can be used before the group they reference.
Example: `(\2?(a)){3}` matches `aaaaaa`.
Supported by: JGsoft; .NET (non-ECMA); Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript (ignored); VBScript (error); XRegExp (error); Python (error); Ruby; std::regex (ECMA; basic; grep; error); Boost (ECMA; 1.78–1.83); Tcl ARE (error); POSIX BRE (error); GNU BRE (error); GNU ERE (error); Oracle (error); XPath (error)

11. Named Groups and Backreferences

Feature: Named capturing group
 Syntax: `(?<name>regex)`
 Description: Captures the text matched by “regex” into the group “name”. The name can contain letters and numbers but must start with a letter.
 Example: `(?<x>abc){3}` matches `abcabcabc`. The group `x` matches `abc`.
 Supported by: JGsoft; .NET; Java (7); Perl (5.10); PCRE (7.0); PCRE2; PHP (5.2.2); Delphi; R; JavaScript; XRegExp; Ruby (1.9); Boost (ECMA; 1.42–1.83)

Feature: Named capturing group
 Syntax: `(?'name' regex)`
 Description: Captures the text matched by “regex” into the group “name”. The name can contain letters and numbers but must start with a letter.
 Example: `(?'x' abc){3}` matches `abcabcabc`. The group `x` matches `abc`.
 Supported by: JGsoft; .NET; Perl (5.10); PCRE (7.0); PCRE2; PHP (5.2.2); Delphi; R; Ruby (1.9); Boost (ECMA; 1.42–1.83)

Feature: Named capturing group
 Syntax: `(?P<name>regex)`
 Description: Captures the text matched by “regex” into the group “name”. The name can contain letters and numbers but must start with a letter.
 Example: `(?P<x> abc){3}` matches `abcabcabc`. The group `x` matches `abc`.
 Supported by: JGsoft; Perl (5.10); PCRE; PCRE2; PHP; Delphi; R; XRegExp; Python

Feature: Duplicate named group
 Syntax: Any named group
 Description: Two named groups can share the same name.
 Example: `(?<x>a) | (?<x>b)` matches `a` or `b`.
 Supported by: JGsoft; .NET; Java (7 error); Perl (5.10); PCRE (6.7 option); PCRE2 (option); PHP (5.2.0 option); Delphi (option); R (option); JavaScript (error); XRegExp (error); Python (error); Ruby (1.9); Boost (ECMA; 1.42–1.83)

Feature: Duplicate named group
 Syntax: Any named group
 Description: Named groups that share the same name are treated as one and the same group, so there are no pitfalls when using backreferences to that name.
 Example:
 Supported by: JGsoft; .NET

Feature: Duplicate named group
 Syntax: Any named group
 Description: If a regex has multiple groups with the same name, backreferences using that name point to the leftmost group in the regex with that name.
 Example:
 Supported by: PCRE (6.7–8.33); PHP (5.2.0–5.5.9); Delphi (XE–XE6); R (2.14.0–3.0.2)

Feature: Duplicate named group
 Syntax: Any named group
 Description: If a regex has multiple groups with the same name, backreferences using that name point to the leftmost group with that name that has actually participated in the match attempt when the backreference is evaluated.
 Example:
 Supported by: Perl (5.10); PCRE (8.36); PCRE2; PHP (5.6.9); Delphi (10.2); R (3.1.3); Boost (ECMA; 1.47–1.83)

Feature: Duplicate named group
 Syntax: Any named group
 Description: If a regex has multiple groups with the same name, backreferences using that name point to the rightmost group with that name that appears to the left of the backreference in the regex.
 Example:
 Supported by: Boost (ECMA; 1.42–1.46)

Feature: Duplicate named group
 Syntax: Any named group
 Description: If a regex has multiple groups with the same name, backreferences using that name can match the text captured by any group with that name that appears to the left of the backreference in the regex.
 Example:
 Supported by: Ruby (1.9)

Feature: Named backreference
 Syntax: `\k<name>`
 Description: Substituted with the text matched by the named group “name”.
 Example: `(?<x>abc|def)=\k<x>` matches `abc=abc` or `def=def`, but not `abc=def` or `def=abc`.
 Supported by: JGsoft; .NET; Java (7); Perl (5.10); PCRE (7.0); PCRE2; PHP (5.2.2); Delphi; R; JavaScript; XRegExp; Ruby (1.9); Boost (ECMA; 1.47–1.83)

Feature: Named backreference
 Syntax: `\k' name '`
 Description: Substituted with the text matched by the named group “name”.
 Example: `(?'x'abc|def)=\k'x'` matches `abc=abc` or `def=def`, but not `abc=def` or `def=abc`.
 Supported by: JGsoft; .NET; Perl (5.10); PCRE (7.0); PCRE2; PHP (5.2.2); Delphi; R; Ruby (1.9); Boost (ECMA; 1.47–1.83)

Feature: Named backreference
 Syntax: `\k{name}`
 Description: Substituted with the text matched by the named group “name”.
 Example: `(?'x'abc|def)=\k{x}` matches `abc=abc` or `def=def`, but not `abc=def` or `def=abc`.
 Supported by: Perl (5.10); PCRE (7.2); PCRE2; PHP (5.2.4); Delphi; R; Boost (ECMA; 1.47–1.83)

Feature: Named backreference
 Syntax: `\g{name}`
 Description: Substituted with the text matched by the named group “name”.
 Example: `(?'x'abc|def)=\g{x}` matches `abc=abc` or `def=def`, but not `abc=def` or `def=abc`.
 Supported by: Perl (5.10); PCRE (7.2); PCRE2; PHP (5.2.4); Delphi; R; Boost (ECMA; 1.42–1.83)

- Feature:** Named backreference
Syntax: (?P=name)
Description: Substituted with the text matched by the named group “name”.
Example: (?P<x>abc|def)=(?P=x) matches abc=abc or def=def, but not abc=def or def=abc.
Supported by: JGsoft; Perl (5.10); PCRE; PCRE2; PHP; Delphi; R; Python
- Feature:** Failed backreference
Syntax: Any named backreference
Description: Backreferences to groups that did not participate in the match attempt fail to match.
Example: (?<x>a)?\k<x> matches aa but fails to match b.
Supported by: JGsoft; .NET (non-ECMA); Java (7); Perl (5.10); PCRE; PCRE2; PHP; Delphi; R; JavaScript (ignored); XRegExp (ignored); Python; Ruby (1.9); Boost (ECMA; 1.47–1.83)
- Feature:** Nested backreference
Syntax: Any named backreference
Description: Backreferences can be used inside the group they reference.
Example: (?<x>a\k<x>?)\{3} matches aaaaaa.
Supported by: JGsoft; .NET; Java (7); Perl (5.10); PCRE (6.5); PCRE2; PHP (5.1.3); Delphi; R; JavaScript (ignored); XRegExp (ignored); Python (2.4–3.4 fail); Ruby (1.9 fail); Boost (ECMA; 1.78–1.83 fail)
- Feature:** Forward reference
Syntax: Any named backreference
Description: Backreferences can be used before the group they reference.
Example: (\k<x>?(?<x>a))\{3} matches aaaaaa.
Supported by: JGsoft; .NET; Java (7 error); Perl (5.10); PCRE (6.7); PCRE2; PHP (5.2.0); Delphi; R; JavaScript (ignored); XRegExp (error); Python (error); Ruby (1.9 error); Boost (ECMA; 1.42–1.83 error)
- Feature:** Named capturing group
Syntax: Any named capturing group
Description: A number is a valid name for a capturing group.
Example: (?<17>abc)\{3} matches abcabcabc. The group named “17” matches abc.
Supported by: JGsoft; .NET; Java (7 error); Perl (5.10 error); PCRE (4.0–8.33); PCRE2 (error); PHP (5.0.0–5.1.2); Delphi (XE–XE6); R (2.14.0–3.0.2); JavaScript (error); XRegExp (error); Python (error); Ruby (1.9 error); Boost (ECMA; 1.42–1.83)
- Feature:** Named capturing group
Syntax: Any capturing group with a number as its name
Description: If the name of the group is a number, that becomes the group’s name and the group’s number.
Example: (?<17>abc|def)=\17 matches abc=abc or def=def, but not abc=def or def=abc.
Supported by: .NET
- Feature:** Named backreference
Syntax: Any named backreference
Description: A number is a valid name for a backreference which then points to a group with that number as its name.
Example: (?<17>abc|def)=\k<17> matches abc=abc or def=def, but not abc=def or def=abc.
Supported by: JGsoft; .NET; PCRE (4.0–8.33); PHP (5.0.0–5.1.2); Delphi (XE–XE6); R (2.14.0–3.0.2); Boost (ECMA; 1.42–1.83 error)

Feature: Named capturing group
Syntax: Any named capturing group
Description: A negative number is a valid name for a capturing group.
Example: `(?<-17>abc){3}` matches `abcabcabc`. The group named “-17” matches `abc`.
Supported by: JGsoft (error); .NET (error); Java (7 error); Perl (5.10 error); PCRE (error); PCRE2 (error); PHP (error); Delphi (error); R (error); JavaScript (error); XRegExp (error); Python (error); Ruby (1.9 error); Boost (ECMA; 1.42–1.83)

Feature: Named backreference
Syntax: Any named backreference
Description: A negative number is a valid name for a backreference which then points to a group with that negative number as its name.
Example:
Supported by: Boost (ECMA; 1.42–1.83 error)

12. Special Groups

Feature: Comment
 Syntax: `(?#comment)`
 Description: Everything between `(?#` and `)` is ignored by the regex engine.
 Example: `a(?#foobar)b` matches `ab`
 Supported by: JGsoft; .NET; Perl; PCRE; PCRE2; PHP; Delphi; R; XRegExp; Python; Ruby; Boost (ECMA); Tcl ARE

Feature: Branch reset group
 Syntax: `(?|regex)`
 Description: If the regex inside the branch reset group has multiple alternatives with capturing groups, then the capturing group numbers are the same in all the alternatives.
 Example: `(x)(?|(a)|(bc)|(def))\2` matches `xaa`, `xbcbc`, or `xdefdef` with the first group capturing `x` and the second group capturing `a`, `bc`, or `def`
 Supported by: JGsoft (V2); Perl (5.10); PCRE (7.2); PCRE2; PHP (5.2.4); Delphi; R; Boost (ECMA; 1.42–1.83)

Feature: Atomic group
 Syntax: `(?>regex)`
 Description: Atomic groups prevent the regex engine from backtracking back into the group after a match has been found for the group. If the remainder of the regex fails, the engine may backtrack over the group if a quantifier or alternation makes it optional. But it will not backtrack into the group to try other permutations of the group.
 Example: `a(?>bc|b)c` matches `abcc` but not `abc`
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; Ruby; Boost (ECMA)

Feature: Positive lookahead
 Syntax: `(?=regex)`
 Description: Matches at a position where the pattern inside the lookahead can be matched. Matches only the position. It does not consume any characters or expand the match. In a pattern like `one(?=two)three`, both `two` and `three` have to match at the position where the match of `one` ends.
 Example: `t(?=s)` matches the second `t` in `streets`.
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex (ECMA); Boost (ECMA); Tcl ARE

Feature: Negative lookahead
 Syntax: `(?!regex)`
 Description: Similar to positive lookahead, except that negative lookahead only succeeds if the regex inside the lookahead fails to match.
 Example: `t(?!s)` matches the first `t` in `streets`.
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python; Ruby; std::regex (ECMA); Boost (ECMA); Tcl ARE

Feature: Positive lookbehind
 Syntax: `(?<=regex)`
 Description: Matches at a position if the pattern inside the lookbehind can be matched ending at that position.
 Example: `(?<=s)t` matches the first `t` in `streets`.
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; XRegExp; Python; Ruby (1.9); Boost (ECMA)

Feature: Negative lookbehind
 Syntax: `(?<!regex)`
 Description: Matches at a position if the pattern inside the lookbehind cannot be matched ending at that position.
 Example: `(?<!s)t` matches the second `t` in `streets`.
 Supported by: JGsoft; .NET; Java; Perl; PCRE; PCRE2; PHP; Delphi; R; JavaScript; XRegExp; Python; Ruby (1.9); Boost (ECMA)

Feature: Lookbehind
 Syntax: `(?<=regex|longer regex)`
 Description: Alternatives inside lookbehind can differ in length.
 Example: `(?<=is|e)t` matches the second and fourth `t` in `twisty streets`.
 Supported by: JGsoft; .NET; Java; Perl (5.30); PCRE; PCRE2; PHP; Delphi; R; JavaScript; XRegExp; Ruby (1.9); Boost (ECMA; 1.38–1.43)

Feature: Lookbehind
 Syntax: `(?<=x{n,m})`
 Description: Quantifiers with a finite maximum number of repetitions can be used inside lookbehind.
 Example: `(?<=s\w{1,7})t` matches only the fourth `t` in `twisty streets`.
 Supported by: JGsoft; .NET; Java (6; 4 fail); Perl (5.30 fail); JavaScript; XRegExp

Feature: Lookbehind
 Syntax: `(?<=regex)`
 Description: The full regular expression syntax can be used inside lookbehind.
 Example: `(?<=s\w+)` matches only the fourth `t` in `twisty streets`.
 Supported by: JGsoft; .NET; Java (13); JavaScript; XRegExp

Feature: Lookbehind
 Syntax: `(group)?<=\1`
 Description: Backreferences can be used inside lookbehind. Syntax prohibited in lookbehind is also prohibited in the referenced capturing group.
 Example: `(\w).+(?<=\1)` matches `twisty street` in `twisty streets`.
 Supported by: JGsoft; .NET; PCRE2 (10.23); PHP (7.3.0); R (4.0.0); JavaScript; XRegExp; Python (3.5)

Feature: Keep text out of the regex match
 Syntax: `\K`
 Description: The text matched by the part of the regex to the left of the `\K` is omitted from the overall regex match. Other than that the regex is matched normally from left to right. Capturing groups to the left of the `\K` capture as usual.
 Example: `s\Kt` matches only the first `t` in `streets`.
 Supported by: JGsoft (V2); Perl (5.10); PCRE (7.2); PCRE2; PHP (5.2.4); Delphi; R; Ruby (2.0); Boost (ECMA; 1.42–1.83)

Feature: Lookaround conditional
 Syntax: `(?(?=regex)then|else)` where `(?=regex)` is any valid lookaround and `then` and `else` are any valid regexes
 Description: If the lookaround succeeds, the “then” part must match for the overall regex to match. If the lookaround fails, the “else” part must match for the overall regex to match. The lookaround is zero-length. The “then” and “else” parts consume their matches like normal regexes.
 Example: `(?(?<=a)b|c)` matches the second `b` and the first `c` in `babxcac`
 Supported by: JGsoft; .NET; Perl; PCRE; PCRE2; PHP; Delphi; R; Boost (ECMA)

Feature: Implicit lookahead conditional
 Syntax: `(?(regex)then|else)` where `regex`, `then`, and `else` are any valid regexes and `regex` is not the name of a capturing group
 Description: If “`regex`” is not the name of a capturing group, then it is interpreted as a lookahead as if you had written `(?(?=regex)then|else)`. 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. The lookaround is zero-length. The “then” and “else” parts consume their matches like normal regexes.
 Example: `(?(\d{2})7|c)` matches the first `7` and the `c` in `747c`
 Supported by: .NET

Feature: Named conditional
 Syntax: `(?(name)then|else)` where `name` is the name of a capturing group and `then` and `else` are any valid regexes
 Description: If the capturing group with the given name took part in the match attempt thus far, the “then” part must match for the overall regex to match. If the capturing group did not take part in the match thus far, the “else” part must match for the overall regex to match.
 Example: `(?<one>a)?(?<(one)b|c)` matches `ab`, the first `c`, and the second `c` in `babxcac`
 Supported by: JGsoft; .NET; PCRE (6.7); PCRE2; PHP (5.2.0); Delphi; R; Python

Feature: Named conditional
 Syntax: `(?(<name>)then|else)` where `name` is the name of a capturing group and `then` and `else` are any valid regexes
 Description: If the capturing group with the given name took part in the match attempt thus far, the “then” part must match for the overall regex to match. If the capturing group did not take part in the match thus far, the “else” part must match for the overall regex to match.
 Example: `(?<one>a)?(?<(<one>)b|c)` matches `ab`, the first `c`, and the second `c` in `babxcac`
 Supported by: JGsoft (V2); Perl (5.10); PCRE (7.0); PCRE2; PHP (5.2.2); Delphi; R; Ruby (2.0); Boost (ECMA; 1.42–1.83)

Feature: Named conditional
 Syntax: `(?('name')then|else)` where `name` is the name of a capturing group and `then` and `else` are any valid regexes
 Description: If the capturing group with the given name took part in the match attempt thus far, the “then” part must match for the overall regex to match. If the capturing group did not take part in the match thus far, the “else” part must match for the overall regex to match.
 Example: `(?('one')a)?(?('one')b|c)` matches `ab`, the first `c`, and the second `c` in `babxcac`
 Supported by: JGsoft (V2); Perl (5.10); PCRE (7.0); PCRE2; PHP (5.2.2); Delphi; R; Ruby (2.0); Boost (ECMA; 1.42–1.83)

Feature: Conditional
 Syntax: `(?(1)then|else)` where 1 is the number of a capturing group and then and else are any valid regexes
 Description: If the referenced capturing group took part in the match attempt thus far, the “then” part must match for the overall regex to match. If the capturing group did not take part in the match thus far, the “else” part must match for the overall regex to match.
 Example: `(a)?(?(1)b|c)` matches `ab`, the first `c`, and the second `c` in `babxcac`
 Supported by: JGsoft; .NET; Perl; PCRE; PCRE2; PHP; Delphi; R; Python; Ruby (2.0); Boost (ECMA)

Feature: Relative conditional
 Syntax: `(?(-1)then|else)` where -1 is a negative integer and then and else are any valid regexes
 Description: Conditional that tests the capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from right to left starting immediately before the conditional. If the referenced capturing group took part in the match attempt thus far, the “then” part must match for the overall regex to match. If the capturing group did not take part in the match thus far, the “else” part must match for the overall regex to match.
 Example: `(a)?(?(-1)b|c)` matches `ab`, the first `c`, and the second `c` in `babxcac`
 Supported by: JGsoft (V2); PCRE (7.2); PCRE2; PHP (5.2.4); Delphi; R

Feature: Forward conditional
 Syntax: `(?(+1)then|else)` where +1 is a positive integer and then and else are any valid regexes
 Description: Conditional that tests the capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from left to right starting at the “then” part of conditional. If the referenced capturing group took part in the match attempt thus far, the “then” part must match for the overall regex to match. If the capturing group did not take part in the match thus far, the “else” part must match for the overall regex to match.
 Example: `((?(+1)b|c)(d)?){2}` matches `cc` and `cdb` in `bdbdcccxcxcdb`
 Supported by: JGsoft (V2); PCRE (7.2); PCRE2; PHP (5.2.4); Delphi; R

Feature: Conditional
 Syntax: `(?(+1)then|else)` where 1 is the number of a capturing group and then and else are any valid regexes
 Description: The + is ignored and the number is taken as an absolute reference to a capturing group. If the referenced capturing group took part in the match attempt thus far, the “then” part must match for the overall regex to match. If the capturing group did not take part in the match thus far, the “else” part must match for the overall regex to match.
 Example: `(a)?(?(+1)b|c)` matches `ab`, the first `c`, and the second `c` in `babxcac`
 Supported by: Python

13. Balancing Groups, Recursion, and Subroutines

Feature: Balancing group
 Syntax: (?<capture-subtract>regex) where “capture” and “subtract” are group names and “regex” is any regex
 Description: The name “subtract” must be used as the name of a capturing group elsewhere in the regex. If this group has captured matches that haven’t been subtracted yet, then the balancing group subtracts one capture from “subtract”, attempts to match “regex”, and stores its match into the group “capture”. If “capture” is omitted, the same happens without storing the match. If “regex” is omitted, the balancing group succeeds without advancing through the string. If the group “subtract” has no matches to subtract, then the balancing group fails to match, regardless of whether “regex” is specified or not.
 Example: `^(?<1>\w)+\w?`
`(\k<1>(?!<1>))+`
`(? (1) (?!))$` matches any palindrome word
 Supported by: JGsoft (V2); .NET

Feature: Balancing group
 Syntax: (?'capture-subtract'regex) where “capture” and “subtract” are group names and “regex” is any regex
 Description: The name “subtract” must be used as the name of a capturing group elsewhere in the regex. If this group has captured matches that haven’t been subtracted yet, then the balancing group subtracts one capture from “subtract”, attempts to match “regex”, and stores its match into the group “capture”. If “capture” is omitted, the same happens without storing the match. If “regex” is omitted, the balancing group succeeds without advancing through the string. If the group “subtract” has no matches to subtract, then the balancing group fails to match, regardless of whether “regex” is specified or not.
 Example: `^(?'1'\w)+\w?`
`(\k'1'(?!'1'))+`
`(? (1) (?!))$` matches any palindrome word
 Supported by: JGsoft (V2); .NET

Feature: Recursion
 Syntax: (?R)
 Description: Recursion of the entire regular expression.
 Example: `a(?R)?z` matches `az`, `aazz`, `aaazzz`, etc.
 Supported by: JGsoft (V2); Perl (5.10); PCRE; PCRE2; PHP; Delphi; R; Boost (ECMA; 1.42–1.83)

Feature: Recursion
 Syntax: (?0)
 Description: Recursion of the entire regular expression.
 Example: `a(?0)?z` matches `az`, `aazz`, `aaazzz`, etc.
 Supported by: JGsoft (V2); Perl (5.10); PCRE; PCRE2; PHP; Delphi; R; Boost (ECMA; 1.42–1.83)

Feature: Recursion
 Syntax: \g<0>
 Description: Recursion of the entire regular expression.
 Example: `a\g<0>?z` matches `az`, `aazz`, `aaazzz`, etc.
 Supported by: JGsoft (V2); PCRE (7.7); PCRE2; PHP (5.2.7); Delphi; R; Ruby (2.0)

- Feature: Recursion
 Syntax: `\g'0'`
 Description: Recursion of the entire regular expression.
 Example: `a\g'0'?z` matches `az`, `aazz`, `aaazzz`, etc.
 Supported by: JGsoft (V2); PCRE (7.7); PCRE2; PHP (5.2.7); Delphi; R; Ruby (2.0)
- Feature: Subroutine call
 Syntax: `(?1)` where 1 is the number of a capturing group
 Description: Recursion of a capturing group or subroutine call to a capturing group.
 Example: `a(b(?1)?y)z` matches `abyz`, `abbyyz`, `abbbyyyz`, etc.
 Supported by: JGsoft (V2); Java (4 only); Perl (5.10); PCRE; PCRE2; PHP; Delphi; R; Boost (ECMA; 1.42–1.83)
- Feature: Subroutine call
 Syntax: `\g<1>` where 1 is the number of a capturing group
 Description: Recursion of a capturing group or subroutine call to a capturing group.
 Example: `a(b\g<1>?y)z` matches `abyz`, `abbyyz`, `abbbyyyz`, etc.
 Supported by: JGsoft (V2); PCRE (7.7); PCRE2; PHP (5.2.7); Delphi; R; Ruby (1.9)
- Feature: Subroutine call
 Syntax: `\g'1'` where 1 is the number of a capturing group
 Description: Recursion of a capturing group or subroutine call to a capturing group.
 Example: `a(b\g'1'?y)z` matches `abyz`, `abbyyz`, `abbbyyyz`, etc.
 Supported by: JGsoft (V2); PCRE (7.7); PCRE2; PHP (5.2.7); Delphi; R; Ruby (1.9)
- Feature: Relative subroutine call
 Syntax: `(?-1)` where -1 is a negative integer
 Description: Recursion of or subroutine call to a capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from right to left starting at the subroutine call.
 Example: `a(b(?-1)?y)z` matches `abyz`, `abbyyz`, `abbbyyyz`, etc.
 Supported by: JGsoft (V2); Perl (5.10); PCRE (7.2); PCRE2; PHP (5.2.4); Delphi; R; Boost (ECMA; 1.42–1.83)
- Feature: Relative subroutine call
 Syntax: `\g<-1>` where -1 is a negative integer
 Description: Recursion of or subroutine call to a capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from right to left starting at the subroutine call.
 Example: `a(b\g<-1>?y)z` matches `abyz`, `abbyyz`, `abbbyyyz`, etc.
 Supported by: JGsoft (V2); PCRE (7.7); PCRE2; PHP (5.2.7); Delphi; R; Ruby (1.9)
- Feature: Relative subroutine call
 Syntax: `\g'-1'` where -1 is a negative integer
 Description: Recursion of or subroutine call to a capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from right to left starting at the subroutine call.
 Example: `a(b\g'-1'?y)z` matches `abyz`, `abbyyz`, `abbbyyyz`, etc.
 Supported by: JGsoft (V2); PCRE (7.7); PCRE2; PHP (5.2.7); Delphi; R; Ruby (1.9)

Feature: Forward subroutine call
 Syntax: `(?+1)` where +1 is a positive integer
 Description: Recursion of or subroutine call to a capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from left to right starting at the subroutine call.
 Example: `(?+1)x([ab])` matches `axa`, `axb`, `bxa`, and `bx b`
 Supported by: JGsoft (V2); Perl (5.10); PCRE (7.2); PCRE2; PHP (5.2.4); Delphi; R; Boost (ECMA; 1.42–1.83)

Feature: Forward subroutine call
 Syntax: `\g<+1>` where +1 is a positive integer
 Description: Recursion of or subroutine call to a capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from left to right starting at the subroutine call.
 Example: `\g<+1>x([ab])` matches `axa`, `axb`, `bxa`, and `bx b`
 Supported by: JGsoft (V2); PCRE (7.7); PCRE2; PHP (5.2.7); Delphi; R; Ruby (2.0)

Feature: Forward subroutine call
 Syntax: `\g'+1'` where +1 is a positive integer
 Description: Recursion of or subroutine call to a capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from left to right starting at the subroutine call.
 Example: `\g'+1'x([ab])` matches `axa`, `axb`, `bxa`, and `bx b`
 Supported by: JGsoft (V2); PCRE (7.7); PCRE2; PHP (5.2.7); Delphi; R; Ruby (2.0)

Feature: Named subroutine call
 Syntax: `(?&name)` where “name” is the name of a capturing group
 Description: Recursion of a capturing group or subroutine call to a capturing group.
 Example: `a(?<x>b(?&x)?y)z` matches `abyz`, `abbyyz`, `abbbyyyz`, etc.
 Supported by: JGsoft (V2); Perl (5.10); PCRE (7.0); PCRE2; PHP (5.2.2); Delphi; R; Boost (ECMA; 1.42–1.83)

Feature: Named subroutine call
 Syntax: `(?P>name)` where “name” is the name of a capturing group
 Description: Recursion of a capturing group or subroutine call to a capturing group.
 Example: `a(?P<x>b(?P>x)?y)z` matches `abyz`, `abbyyz`, `abbbyyyz`, etc.
 Supported by: JGsoft (V2); Perl (5.10); PCRE; PCRE2; PHP; Delphi; R; Boost (ECMA; 1.42–1.83)

Feature: Named subroutine call
 Syntax: `\g<name>` where “name” is the name of a capturing group
 Description: Recursion of a capturing group or subroutine call to a capturing group.
 Example: `a(?<x>b\g<x>?y)z` matches `abyz`, `abbyyz`, `abbbyyyz`, etc.
 Supported by: JGsoft (V2); PCRE (7.7); PCRE2; PHP (5.2.7); Delphi; R; Ruby (1.9)

Feature: Named subroutine call
 Syntax: `\g'name'` where “name” is the name of a capturing group
 Description: Recursion of a capturing group or subroutine call to a capturing group.
 Example: `a(?'x'b\g'x'?y)z` matches `abyz`, `abbyyz`, `abbbyyyz`, etc.
 Supported by: JGsoft (V2); PCRE (7.7); PCRE2; PHP (5.2.7); Delphi; R; Ruby (1.9)

Feature: Subroutine definitions
 Syntax: `(?(DEFINE) regex)` where “regex” is any regex
 Description: The DEFINE group does not take part in the matching process. Subroutine calls can be made to capturing groups inside the DEFINE group.
 Example: `(?(DEFINE) ([ab]))`
`x(?1)y(?1)z` matches `xayaz`, `xaybz`, `xbyaz`, and `xbybz`
 Supported by: JGsoft (V2); Perl (5.10); PCRE (7.0); PCRE2; PHP (5.2.2); Delphi; R; Boost (ECMA; 1.42–1.83)

Feature: Subroutine calls capture
 Syntax: Subroutine call using Ruby-style `\g` syntax
 Description: A subroutine call to a capturing group makes that capturing group store the text matched during the subroutine call.
 Example: When `([ab])\g'1'` matches `ab` the first capturing group holds `b` after the match.
 Supported by: JGsoft (V2); Ruby (1.9)

Feature: Subroutine calls capture
 Syntax: Subroutine call using syntax other than `\g`
 Description: A subroutine call to a capturing group makes that capturing group store the text matched during the subroutine call.
 Example: When `([ab])(?1)` matches `ab` the first capturing group holds `b` after the match.
 Supported by:

Feature: Recursion isolates capturing groups
 Syntax: Any recursion or subroutine call
 Description: Each subroutine call has its own separate storage space for capturing groups. Backreferences inside the call cannot see text matched before the call, and backreferences after the call cannot see text matched inside the call. Backreferences inside recursion cannot see text matched at other recursion levels.
 Example: `(a)(\1)(?2)` never matches anything because `\1` always fails during the call made by `(?2)`.
 Supported by: Perl (5.10–5.18)

Feature: Recursion reverts capturing groups
 Syntax: Recursion or subroutine call using Ruby-style `\g` syntax
 Description: When the regex engine exits from recursion or a subroutine call, it reverts all capturing groups to the text they had matched prior to entering the recursion or subroutine call.
 Example: When `(a)(([bc])\1)\g'2'` matches `abaca` the third group stores `b` after the match
 Supported by: PCRE (7.7); PCRE2; PHP (5.2.7); Delphi; R

Feature: Recursion reverts capturing groups
 Syntax: Recursion or subroutine call using syntax other than `\g`
 Description: When the regex engine exits from recursion or a subroutine call, it reverts all capturing groups to the text they had matched prior to entering the recursion or subroutine call.
 Example: When `(a)(([bc])\1)(?2)` matches `abaca` the third group stores `b` after the match
 Supported by: JGsoft (V2); Perl (5.10); PCRE; PCRE2; PHP; Delphi; R; Boost (ECMA; 1.42–1.83)

Feature: Recursion does not isolate or revert capturing groups
 Syntax: Recursion or subroutine call using Ruby-style `\g` syntax
 Description: Capturing groups are not given any special treatment by recursion and subroutine calls, except perhaps that subroutine calls capture. Backreferences always see the text most recently matched by each capturing group, regardless of whether they are inside the same level of recursion or not.
 Example: When `(a)(([bc])\1)\g'2'` matches `abaca` the third group stores `c` after the match
 Supported by: JGsoft (V2); Ruby (1.9)

Feature: Recursion does not isolate or revert capturing groups
 Syntax: Recursion or subroutine call using syntax other than `\g`
 Description: Capturing groups are not given any special treatment by recursion and subroutine calls, except perhaps that subroutine calls capture. Backreferences always see the text most recently matched by each capturing group, regardless of whether they are inside the same level of recursion or not.
 Example: When `(a)(([bc])\1)(?)` matches `abaca` the third group stores `c` after the match
 Supported by: Java (4 only)

Feature: Recursion is atomic
 Syntax: Recursion or subroutine call using `(?P>...)`
 Description: Recursion and subroutine calls are atomic. Once the regex engine exits from them, it will not backtrack into it to try different permutations of the recursion or subroutine call.
 Example: `(a+)(?P>1)(?P>1)` can never match anything because the first `(?P>1)` matches all remaining a's and the regex engine won't backtrack into the first `(?P>1)` when the second one fails
 Supported by: JGsoft (V2); PCRE (6.5); PCRE2 (10.00–10.23); PHP (5.1.3–7.2.34); Delphi; R (2.14.0–3.6.3)

Feature: Recursion is atomic
 Syntax: Recursion of the whole regex using syntax other than `(?P>0)`
 Description: Recursion of the whole regex is atomic. Once the regex engine exits from recursion, it will not backtrack into it to try different permutations of the recursion.
 Example: `aa$a(?R)a|a` matches `a` in `aaa` when recursion is atomic; otherwise it would match the whole string.
 Supported by: PCRE (6.5); PCRE2; PHP (5.1.3–7.2.34); Delphi; R; Boost (ECMA; 1.42–1.83)

Feature: Recursion is atomic
 Syntax: Subroutine call using syntax other than `(?P>...)`
 Description: Subroutine calls are atomic. Once the regex engine exits from them, it will not backtrack into it to try different permutations of the subroutine call.
 Example: `(a+)(?1)(?1)` can never match anything because the first `(?1)` matches all remaining a's and the regex engine won't backtrack into the first `(?1)` when the second one fails
 Supported by: Java (4 only); PCRE (6.5); PCRE2 (10.00–10.23); PHP (5.1.3–7.2.34); Delphi; R (2.14.0–3.6.3)

14. Replacement String Characters

Feature: Backslash
 Syntax: Backslash followed by any character that does not form a token
 Description: A backslash that is followed by any character that does not form a replacement string token in combination with the backslash inserts the escaped character literally.
 Example: Replacing with `\!` yields `!`
 Supported by: Java; Perl; PCRE2 (extended); R; Python (3.7–3.10 error); `std::regex` (sed); Boost; XPath (error)

Feature: Backslash
 Syntax: A backslash that does not form a token
 Description: A backslash that is not part of a replacement string token is a literal backslash.
 Example: Replacing with `\!` yields `\!`
 Supported by: JGsoft; .NET; PCRE2 (default); PHP; Delphi; JavaScript; VBScript; XRegExp; Python; Ruby; `std::regex` (default); Tcl ARE; Oracle; XPath (error)

Feature: Backslash
 Syntax: Trailing backslash
 Description: A backslash at the end of the replacement string is a literal backslash.
 Example: Replacing with `\` yields `\`
 Supported by: JGsoft; .NET; Java (error); Perl (error); PCRE2 (default); PHP; Delphi; JavaScript; VBScript; XRegExp; Python (error); Ruby; `std::regex` (default); Boost; Tcl ARE; Oracle; XPath (error)

Feature: Backslash
 Syntax: `\\`
 Description: A backslash escapes itself.
 Example: Replacing with `\\` yields `\`
 Supported by: JGsoft; Java; Perl; PCRE2 (extended); PHP; Delphi; R; Python; Ruby; `std::regex` (sed); Boost; Tcl ARE; Oracle; XPath

Feature: Dollar
 Syntax: A dollar that does not form a token
 Description: A dollar sign that does not form a replacement string token is a literal dollar sign.
 Example: Replacing with `$!` yields `$!`
 Supported by: JGsoft; .NET; Java (error); Perl (error); PCRE2 (error); PHP; Delphi; R; JavaScript; VBScript; XRegExp (error); Python; Ruby; `std::regex`; Boost; Tcl ARE; Oracle; XPath (error)

Feature: Dollar
 Syntax: Trailing dollar
 Description: A dollar sign at the end of the replacement string is a literal dollar sign.
 Example: Replacing with `$` yields `$`
 Supported by: JGsoft; .NET; Java (error); Perl (error); PCRE2 (error); PHP; Delphi; R; JavaScript; VBScript; XRegExp (2–4); Python; Ruby; `std::regex` (default VC'15–VC'22; sed VC'08–VC'22); Boost; Tcl ARE; Oracle; XPath (error)

Feature: Dollar
 Syntax: `$$`
 Description: A dollar sign escapes itself.
 Example: Replacing with `$$` yields `$`
 Supported by: JGsoft; .NET; Java (error); Perl (error); PCRE2; Delphi; JavaScript; VBScript; XRegExp; std::regex (default); Boost (all; default); XPath (error)

Feature: Dollar
 Syntax: `\$`
 Description: A backslash escapes a dollar sign.
 Example: Replacing with `\$` yields `$`
 Supported by: JGsoft; Java; Perl; PCRE2 (extended); PHP; Delphi; R; Python (3.7–3.10 error); std::regex (sed); Boost; XPath

Feature: Hexadecimal escape
 Syntax: `\xFF` where FF are 2 hexadecimal digits
 Description: Inserts the character at the specified position in the code page
 Example: `\xA9` inserts `©` when using the Latin-1 code page
 Supported by: JGsoft; Perl; PCRE2 (extended); PHP (string); R (string); JavaScript (string); XRegExp (string); Python (string); Ruby (string); std::regex (string); Boost; Tcl ARE (string); XPath (error)

Feature: Unicode escape
 Syntax: `\uFFFF` where FFFF are 4 hexadecimal digits
 Description: Inserts a specific Unicode code point.
 Example: `\u00E0` inserts `à` encoded as U+00E0 only. `\u00A9` inserts `©`
 Supported by: JGsoft; Java (string); PCRE2 (extended error); R (string); JavaScript (string); XRegExp (string); Python (string); Ruby (1.9 string); std::regex (string); Tcl ARE (string); XPath (error)

Feature: Unicode escape
 Syntax: `\u{FFFF}` where FFFF are 1 to 4 hexadecimal digits
 Description: Inserts a specific Unicode code point.
 Example: `\u{E0}` inserts `à` encoded as U+00E0 only. `\u{A9}` inserts `©`
 Supported by: JGsoft (V2); PCRE2 (extended error); PHP (7.0.0 string); R (string); JavaScript (string); XRegExp (string); Python (3.7–3.10 error); Ruby (1.9 string); XPath (error)

Feature: Unicode escape
 Syntax: `\x{FFFF}` where FFFF are 1 to 4 hexadecimal digits
 Description: Inserts a specific Unicode code point.
 Example: `\x{E0}` inserts `à` encoded as U+00E0 only. `\x{A9}` inserts `©`
 Supported by: JGsoft; Perl; PCRE2 (extended); Python (3.7–3.10 error); Boost; XPath (error)

Feature: Character escape
 Syntax: `\n`, `\r` and `\t`
 Description: Insert an LF character, CR character and a tab character respectively
 Example: `\r\n` inserts a Windows CRLF line break
 Supported by: JGsoft; Java (string); Perl; PCRE2 (extended); PHP (string); R (string); JavaScript (string); XRegExp (string); Python; Ruby (string); std::regex (string); Boost; Tcl ARE (string); XPath (error)

- Feature: Character escape
 Syntax: `\a`
 Description: Insert the “alert” or “bell” control character (ASCII 0x07)
 Example:
 Supported by: Perl; PCRE2 (extended); Python; Boost; XPath (error)
- Feature: Character escape
 Syntax: `\b`
 Description: Insert the “backspace” control character (ASCII 0x08)
 Example:
 Supported by: Perl; PCRE2 (extended error); Python; XPath (error)
- Feature: Character escape
 Syntax: `\e`
 Description: Insert the “escape” control character (ASCII 0x1B)
 Example:
 Supported by: Perl; PCRE2 (extended); Python (3.7–3.10 error); Boost; XPath (error)
- Feature: Character escape
 Syntax: `\f`
 Description: Insert the “form feed” control character (ASCII 0x0C)
 Example:
 Supported by: Perl; PCRE2 (extended); Python; Boost; XPath (error)
- Feature: Character escape
 Syntax: `\v`
 Description: Insert the “vertical tab” control character (ASCII 0x0B)
 Example:
 Supported by: PCRE2 (extended error); Python; Boost; XPath (error)
- Feature: Control character escape
 Syntax: `\cA` through `\cZ`
 Description: Insert an ASCII character Control+A through Control+Z, equivalent to `\x01` through `\x1A`
 Example: `\cm\cj` inserts a Windows CRLF line break
 Supported by: Perl; PCRE2 (extended); Boost
- Feature: Control character escape
 Syntax: `\ca` through `\cz`
 Description: Insert an ASCII character Control+A through Control+Z, equivalent to `\x01` through `\x1A`
 Example: `\cm\cj` inserts a Windows CRLF line break
 Supported by: Perl; PCRE2 (extended); Boost
- Feature: NULL escape
 Syntax: `\0`
 Description: Insert the NULL character
 Example:
 Supported by: Perl; PCRE2 (extended error); Python; Boost (all; default); XPath (error)

Feature: Octal escape
 Syntax: `\o{7777}` where 7777 is any octal number
 Description: Inserts the character at the specified position in the active code page
 Example: `\o{20254}` inserts € when using Unicode
 Supported by: JGsoft (V2); Perl (5.14); PCRE2 (extended); Python (3.7–3.10 error); XPath (error)

Feature: Octal escape
 Syntax: `\10` through `\77`
 Description: Inserts the character at the specified position in the ASCII table
 Example: `\77` inserts ?
 Supported by: Perl; PCRE2 (extended error); XPath (error)

Feature: Octal escape
 Syntax: `\100` through `\177`
 Description: Inserts the character at the specified position in the ASCII table
 Example: `\100` inserts @
 Supported by: Perl; PCRE2 (extended error); Python; XPath (error)

Feature: Octal escape
 Syntax: `\200` through `\377`
 Description: Inserts the character at the specified position in the active code page
 Example: `\377` inserts ÿ when using the Latin-1 code page
 Supported by: Perl; PCRE2 (extended error); Python; XPath (error)

Feature: Octal escape
 Syntax: `\400` through `\777`
 Description: Inserts the character at the specified position in the active code page
 Example: `\777` inserts ø when using Unicode
 Supported by: Perl; PCRE2 (extended error); XPath (error)

Feature: Octal escape
 Syntax: `\01` through `\07`
 Description: Inserts the character at the specified position in the ASCII table
 Example: `\07` inserts the “bell” character
 Supported by: Perl; PCRE2 (extended error); Python; Boost (all; default); XPath (error)

Feature: Octal escape
 Syntax: `\010` through `\077`
 Description: Inserts the character at the specified position in the ASCII table
 Example: `\077` inserts ?
 Supported by: Perl; PCRE2 (extended error); Python; Boost (all; default); XPath (error)

15. Matched Text and Backreferences in Replacement Strings

- Feature: Ampersand
 Syntax: `\&`
 Description: Insert a literal ampersand.
 Example: Replacing with `\&` yields `&`
 Supported by: Java; Perl; PCRE2 (extended); R; std::regex (sed); Boost; Tcl ARE; XPath (error)
- Feature: Whole match
 Syntax: `\&`
 Description: Insert the whole regex match.
 Example: Replacing `\d+` with `[\&]` in `1a2b` yields `[1]a[2]b`
 Supported by: JGsoft; Delphi; Ruby; XPath (error)
- Feature: Whole match
 Syntax: `$$`
 Description: Insert the whole regex match.
 Example: Replacing `\d+` with `[$$]` in `1a2b` yields `[1]a[2]b`
 Supported by: JGsoft; .NET; Java (error); Perl; PCRE2 (error); Delphi; JavaScript; VBScript; XRegExp; std::regex (default); Boost (all; default); XPath (error)
- Feature: Whole match
 Syntax: `&`
 Description: Insert the whole regex match.
 Example: Replacing `\d+` with `[&]` in `1a2b` yields `[1]a[2]b`
 Supported by: std::regex (sed); Boost (sed); Tcl ARE
- Feature: Whole match
 Syntax: `\0`
 Description: Insert the whole regex match.
 Example: Replacing `\d+` with `[\0]` in `1a2b` yields `[1]a[2]b`
 Supported by: JGsoft; PHP; Delphi; Ruby; std::regex (sed); Boost (sed); Tcl ARE; XPath (error)
- Feature: Whole match
 Syntax: `$0`
 Description: Insert the whole regex match.
 Example: Replacing `\d+` with `[$0]` in `1a2b` yields `[1]a[2]b`
 Supported by: JGsoft; .NET; Java; Perl (error); PCRE2; PHP; Delphi; XRegExp; std::regex (default; VC'08–VC'13); Boost (all; default); XPath
- Feature: Whole match
 Syntax: `\g<0>`
 Description: Insert the whole regex match.
 Example: Replacing `\d+` with `[\g<0>]` in `1a2b` yields `[1]a[2]b`
 Supported by: JGsoft; Python; XPath (error)

Feature: Whole match
 Syntax: `$MATCH` and `${^MATCH}`
 Description: Insert the whole regex match.
 Example: Replacing `\d+` with `[$MATCH]` in `1a2b` yields `[1]a[2]b`
 Supported by: Java (error); Perl (error); PCRE2 (error); XRegExp (error); Boost (all; default; 1.42–1.83); XPath (error)

Feature: Backreference
 Syntax: `\1` through `\9`
 Description: Insert the text matched by one of the first 9 capturing groups.
 Example: Replacing `(a)(b)(c)` with `\3\3\1` in `abc` yields `cca`
 Supported by: JGsoft; Perl; PHP; Delphi; R; Python; Ruby; std::regex (sed); Boost; Tcl ARE; Oracle; XPath (error)

Feature: Backreference
 Syntax: `\10` through `\99`
 Description: Insert the text matched by capturing groups 10 through 99.
 Example:
 Supported by: JGsoft; PHP; Delphi; Python

Feature: Backreference and literal
 Syntax: `\10` through `\99`
 Description: When there are fewer capturing groups than the 2-digit number, treat this as a single-digit backreference followed by a literal number instead of as an invalid backreference.
 Example: Replacing `(a)(b)(c)` with `\39\38\17` in `abc` yields `c9c8a7`
 Supported by: JGsoft; Delphi

Feature: Backreference
 Syntax: `$1` through `$9`
 Description: Insert the text matched by one of the first 9 capturing groups.
 Example: Replacing `(a)(b)(c)` with `$3$3$1` in `abc` yields `cca`
 Supported by: JGsoft; .NET; Java; Perl; PCRE2; PHP; Delphi; JavaScript; VBScript; XRegExp; std::regex (default); Boost (all; default); XPath

Feature: Backreference
 Syntax: `$10` through `$99`
 Description: Insert the text matched by capturing groups 10 through 99.
 Example:
 Supported by: JGsoft; .NET; Java; Perl; PCRE2; PHP; Delphi; JavaScript; VBScript; XRegExp; std::regex (default); Boost (all; default); XPath

Feature: Backreference and literal
 Syntax: `$10` through `$99`
 Description: When there are fewer capturing groups than the 2-digit number, treat this as a single-digit backreference followed by a literal number instead of as an invalid backreference.
 Example: Replacing `(a)(b)(c)` with `$39$38$17` in `abc` yields `c9c8a7`
 Supported by: JGsoft; .NET (ECMA); Java; Delphi; JavaScript; VBScript; XPath

Feature: Backreference
 Syntax: `#{1}` through `#{99}`
 Description: Insert the text matched by capturing groups 1 through 99.
 Example: Replacing `(a)(b)(c)` with `#{3}#{3}#{1}` in `abc` yields `cca`
 Supported by: JGsoft; .NET; Java (error); Perl; PCRE2; PHP; Delphi; XRegExp; Boost (all; default); XPath (error)

Feature: Backreference
 Syntax: `\g<1>` through `\g<99>`
 Description: Insert the text matched by capturing groups 1 through 99.
 Example: Replacing `(a)(b)(c)` with `\g<3>\g<3>\g<1>` in `abc` yields `cca`
 Supported by: JGsoft; Python; XPath (error)

Feature: Named backreference
 Syntax: `#{name}`
 Description: Insert the text matched by the named capturing group “name”.
 Example: Replacing `(?'one'a)(?'two'b)` with `#{two}#{one}` in `ab` yields `ba`
 Supported by: JGsoft; .NET; Java (7); Perl (error); PCRE2; Delphi; XRegExp; XPath (error)

Feature: Named backreference
 Syntax: `#{+name}`
 Description: Insert the text matched by the named capturing group “name”.
 Example: Replacing `(?'one'a)(?'two'b)` with `#{+two}#{+one}` in `ab` yields `ba`
 Supported by: Java (error); Perl (5.10); PCRE2 (error); XRegExp (error); Boost (all; default; 1.42–1.83); XPath (error)

Feature: Named backreference
 Syntax: `$name`
 Description: Insert the text matched by the named capturing group “name”.
 Example: Replacing `(?'one'a)(?'two'b)` with `twoone` in `ab` yields `ba`
 Supported by: Java (error); Perl (error); PCRE2; XRegExp (error); XPath (error)

Feature: Named backreference
 Syntax: `\g<name>`
 Description: Insert the text matched by the named capturing group “name”.
 Example: Replacing `(?P<one>a)(?P<two>b)` with `\g<two>\g<one>` in `ab` yields `ba`
 Supported by: JGsoft; Delphi; Python; XPath (error)

Feature: Invalid backreference
 Syntax: Any supported backreference syntax
 Description: A backreference that indicates a number greater than the highest-numbered group or a name of a group that does not exist is replaced with the empty string.
 Example:
 Supported by: JGsoft (V1 only); Java (error); Perl; PCRE2 (error); PHP; Delphi; R; XRegExp (error); Python (error); Ruby; std::regex (default VC'08–VC'22; sed VC'15–VC'22); Boost; Tcl ARE; Oracle; XPath

- Feature: Invalid backreference
 Syntax: Any supported backreference syntax
 Description: A backreference that indicates a number greater than the highest-numbered group or a name of a group that does not exist is treated as literal text that is inserted as such in the replacement.
 Example:
 Supported by: JGsoft (V2 error); .NET; Java (error); PCRE2 (error); JavaScript; VBScript; XRegExp (error); Python (error); std::regex (sed; VC'08–VC'13 error)
- Feature: Backreference to non-participating group
 Syntax: Any supported backreference syntax
 Description: A backreference to a non-participating capturing group is replaced with the empty string.
 Example:
 Supported by: JGsoft; .NET; Java; Perl; PCRE2 (error); PHP; Delphi; R; JavaScript; VBScript; XRegExp; Python (3.5); Ruby; std::regex; Boost; Tcl ARE; Oracle; XPath
- Feature: Last backreference
 Syntax: `\+`
 Description: Insert the text matched by the highest-numbered capturing group that actually participated in the match.
 Example: Replacing `(a)(z)?` with `[\+]` in `ab` yields `[a]b`
 Supported by: JGsoft; Delphi; Ruby; XPath (error)
- Feature: Last backreference
 Syntax: `\+`
 Description: Insert the text matched by the highest-numbered capturing group, regardless of whether it participated in the match.
 Example: Replacing `(a)(z)?` with `[\+]` in `ab` yields `[]b`
 Supported by: XPath (error)
- Feature: Last backreference
 Syntax: `$+`
 Description: Insert the text matched by the highest-numbered capturing group that actually participated in the match.
 Example: Replacing `(a)(z)?` with `[$+]` in `ab` yields `[a]b`
 Supported by: JGsoft; Java (error); Perl (5.18); PCRE2 (error); Delphi; XRegExp (error); XPath (error)
- Feature: Last backreference
 Syntax: `$+`
 Description: Insert the text matched by the highest-numbered capturing group, regardless of whether it participated in the match.
 Example: Replacing `(a)(z)?` with `[$+]` in `ab` yields `[]b`
 Supported by: .NET; Java (error); Perl (5.8–5.16); PCRE2 (error); VBScript; XRegExp (error); Boost (all; default; 1.42–1.83); XPath (error)

Feature: Last backreference
 Syntax: `$$^N`
 Description: Insert the text matched by the highest-numbered capturing group that actually participated in the match.
 Example: Replacing `(a)(z)?` with `[$^N]` in `ab` yields `[a]b`
 Supported by: Java (error); Perl; PCRE2 (error); XRegExp (error); Boost (all; default; 1.42–1.83); XPath (error)

Feature: Last backreference
 Syntax: `$LAST_SUBMATCH_RESULT` and `${^LAST_SUBMATCH_RESULT}`
 Description: Insert the text matched by the highest-numbered capturing group that actually participated in the match.
 Example: Replacing `(a)(z)?` with `[$LAST_SUBMATCH_RESULT]` in `ab` yields `[a]b`
 Supported by: Java (error); Perl (error); PCRE2 (error); XRegExp (error); Boost (all; default; 1.42–1.83); XPath (error)

Feature: Last backreference
 Syntax: `$LAST_PAREN_MATCH` and `${^LAST_PAREN_MATCH}`
 Description: Insert the text matched by the highest-numbered capturing group, regardless of whether it participated in the match.
 Example: Replacing `(a)(z)?` with `[$LAST_PAREN_MATCH]` in `ab` yields `[]b`
 Supported by: Java (error); Perl (error); PCRE2 (error); XRegExp (error); Boost (all; default; 1.42–1.83); XPath (error)

16. Context and Case Conversion in Replacement Strings

- Feature: Match Context
 Syntax: `\`` (backslash backtick)
 Description: Insert the part of the subject string to the left of the regex match
 Example: Replacing `b` with `\`` in `abc` yields `aac`
 Supported by: JGsoft; Delphi; Ruby; XPath (error)
- Feature: Match Context
 Syntax: `$`` (dollar backtick)
 Description: Insert the part of the subject string to the left of the regex match
 Example: Replacing `b` with `$`` in `abc` yields `aac`
 Supported by: JGsoft; .NET; Java (error); Perl; PCRE2 (error); Delphi; JavaScript; VBScript; XRegExp; std::regex (default); Boost (all; default); XPath (error)
- Feature: Match Context
 Syntax: `$PREMATCH` and `${^PREMATCH}`
 Description: Insert the part of the subject string to the left of the regex match
 Example: Replacing `b` with `$PREMATCH` in `abc` yields `aac`
 Supported by: Java (error); Perl (error); PCRE2 (error); XRegExp (error); Boost (all; default; 1.42–1.83); XPath (error)
- Feature: Match Context
 Syntax: `\'` (backslash quote)
 Description: Insert the part of the subject string to the right of the regex match
 Example: Replacing `b` with `\'` in `abc` yields `acc`
 Supported by: JGsoft; Delphi; Ruby; XPath (error)
- Feature: Match Context
 Syntax: `$'` (dollar quote)
 Description: Insert the part of the subject string to the right of the regex match
 Example: Replacing `b` with `$'` in `abc` yields `acc`
 Supported by: JGsoft; .NET; Java (error); Perl; PCRE2 (error); Delphi; JavaScript; VBScript; XRegExp; std::regex (default); Boost (all; default); XPath (error)
- Feature: Match Context
 Syntax: `$POSTMATCH` and `${^POSTMATCH}`
 Description: Insert the part of the subject string to the right of the regex match
 Example: Replacing `b` with `$POSTMATCH` in `abc` yields `acc`
 Supported by: Java (error); Perl (error); PCRE2 (error); XRegExp (error); Boost (all; default; 1.42–1.83); XPath (error)
- Feature: Match Context
 Syntax: `_`
 Description: Insert the whole subject string
 Example: Replacing `b` with `_` in `abc` yields `aabcc`
 Supported by: XPath (error)

Feature: Match Context
 Syntax: `$`
 Description: Insert the whole subject string
 Example: Replacing `b` with `$_` in `abc` yields `aabcc`
 Supported by: JGsoft; .NET; Java (error); Perl (error); PCRE2 (error); Delphi; VBScript; XRegExp (error); XPath (error)

Feature: Case Conversion
 Syntax: `\U0` and `\U1` through `\U99`
 Description: Insert the whole regex match or the 1st through 99th backreference with all letters in the matched text converted to uppercase.
 Example: Replacing `.+` with `\U0` in `HeLL10 WoRlD` yields `HELLO WORLD`
 Supported by: JGsoft; Delphi; Python (3.7–3.10 error); XPath (error)

Feature: Case Conversion
 Syntax: `\L0` and `\L1` through `\L99`
 Description: Insert the whole regex match or the 1st through 99th backreference with all letters in the matched text converted to lowercase.
 Example: Replacing `.+` with `\L0` in `HeLL10 WoRlD` yields `hello world`
 Supported by: JGsoft; Delphi; Python (3.7–3.10 error); XPath (error)

Feature: Case Conversion
 Syntax: `\F0` and `\F1` through `\F99`
 Description: Insert the whole regex match or the 1st through 99th backreference with the first letter in the matched text converted to uppercase and the remaining letters converted to lowercase.
 Example: Replacing `.+` with `\F0` in `HeLL10 WoRlD` yields `Hello world`
 Supported by: JGsoft; PCRE2 (extended error); Delphi; Python (3.7–3.10 error); XPath (error)

Feature: Case Conversion
 Syntax: `\I0` and `\I1` through `\I99`
 Description: Insert the whole regex match or the 1st through 99th backreference with the first letter of each word in the matched text converted to uppercase and the remaining letters converted to lowercase.
 Example: Replacing `.+` with `\I0` in `HeLL10 WoRlD` yields `Hello World`
 Supported by: JGsoft; PCRE2 (extended error); Delphi; Python (3.7–3.10 error); XPath (error)

Feature: Case Conversion
 Syntax: `\U`
 Description: All literal text and all text inserted by replacement text tokens after `\U` up to the next `\E` or `\L` is converted to uppercase.
 Example: Replacing `(\w+)` `(\w+)` with `\U$1 CrUeL \E$2` in `HeLL10 WoRlD` yields `HELLO CRUEL WoRlD`
 Supported by: Perl; PCRE2 (extended); Python (3.7–3.10 error); Boost (all; default); XPath (error)

Feature: Case Conversion
 Syntax: `\L`
 Description: All literal text and all text inserted by replacement text tokens after `\L` up to the next `\E` or `\U` is converted to lowercase.
 Example: Replacing `(\w+)` `(\w+)` with `\L$1 CrUeL \E$2` in `HeLL10 WoRlD` yields `hello cruel WoRlD`
 Supported by: Perl; PCRE2 (extended); Python (3.7–3.10 error); Boost (all; default); XPath (error)

Feature: Case Conversion
 Syntax: `\u`
 Description: The first character after `\u` that is inserted into the replacement text as a literal or by a token is converted to uppercase.
 Example: Replacing `(\w+)` `(\w+)` with `\u$1 \uCrUeL \u$2` in `hELlO wOrLd` yields `HELLO CRuE1 wOrLd`
 Supported by: Perl; PCRE2 (extended); Python (3.7–3.10 error); Boost (all; default); XPath (error)

Feature: Case Conversion
 Syntax: `\l`
 Description: The first character after `\l` that is inserted into the replacement text as a literal or by a token is converted to lowercase.
 Example: Replacing `(\w+)` `(\w+)` with `\l$1 \lCrUeL \l$2` in `HeLlO WoRlD` yields `heLlO crUeL woRlD`
 Supported by: Perl; PCRE2 (extended); Python (3.7–3.10 error); Boost (all; default); XPath (error)

Feature: Case Conversion
 Syntax: `\u\L`
 Description: The first character after `\u\L` that is inserted into the replacement text as a literal or by a token is converted to uppercase and the following characters up to the next `\E` or `\U` are converted to lowercase.
 Example: Replacing `(\w+)` `(\w+)` with `\u\L$1 \uCrUeL \E\u$2` in `HeLlO wOrLd` yields `Hello Cruel wOrLd`
 Supported by: Perl; Python (3.7–3.10 error); XPath (error)

Feature: Case Conversion
 Syntax: `\l\U`
 Description: The first character after `\l\U` that is inserted into the replacement text as a literal or by a token is converted to lowercase and the following characters up to the next `\E` or `\L` are converted to uppercase.
 Example: Replacing `(\w+)` `(\w+)` with `\l\U$1 \lCrUeL \E\l$2` in `HeLlO WoRlD` yields `hELLO CRUEL woRlD`
 Supported by: Perl; Python (3.7–3.10 error); XPath (error)

Feature: Case Conversion
 Syntax: `\L\u`
 Description: The first character after `\L\u` that is inserted into the replacement text as a literal or by a token is converted to uppercase and the following characters up to the next `\E` or `\U` are converted to lowercase.
 Example: Replacing `(\w+)` `(\w+)` with `\L\u$1 \uCrUeL \E\u$2` in `HeLlO wOrLd` yields `Hello Cruel wOrLd`
 Supported by: Python (3.7–3.10 error); Boost (all; default); XPath (error)

Feature: Case Conversion
 Syntax: `\U\l`
 Description: The first character after `\U\l` that is inserted into the replacement text as a literal or by a token is converted to lowercase and the following characters up to the next `\E` or `\L` are converted to uppercase.
 Example: Replacing `(\w+)` `(\w+)` with `\U\l$1 \lCrUeL \E\l$2` in `HeLlO WoRlD` yields `hELLO CRUEL woRlD`
 Supported by: Python (3.7–3.10 error); Boost (all; default); XPath (error)

17. Conditionals in Replacement Strings

Feature: Conditional
 Syntax: `?1yes:no` through `?99yes:no`
 Description: Conditional referencing a numbered capturing group. Inserts the “yes” part if the group participated or the “no” part if it didn’t.
 Example: Replacing all matches of `(y)?|n` in `yyn!` with `?1yes:no` yields `yesyesno!`
 Supported by: Boost (all)

Feature: Conditional
 Syntax: `(?1yes:no)` through `(?99yes:no)`
 Description: Conditional referencing a numbered capturing group. Inserts the “yes” part if the group participated or the “no” part if it didn’t.
 Example: Replacing all matches of `(y)?|n` in `yyn!` with `(?1yes:no)` yields `yesyesno!`
 Supported by: JGsoft (V2); Boost (all)

Feature: Conditional
 Syntax: `(?10yes:no)` through `(?99yes:no)`
 Description: When there are fewer capturing groups than the 2-digit number, treat this as a single-digit conditional with the “yes” part starting with a literal number instead of as an invalid conditional.
 Example: Replacing all matches of `(y)?|n` in `yyn!` with `(?19yes:no)` yields `9yes9yesno!`
 Supported by: JGsoft (V2)

Feature: Conditional
 Syntax: `{1}yes:no` through `{99}yes:no`
 Description: Conditional referencing a numbered capturing group. Inserts the “yes” part if the group participated or the “no” part if it didn’t.
 Example: Replacing all matches of `(y)?|n` in `yyn!` with `{1}yes:no` yields `yesyesno!`
 Supported by: Boost (all; 1.42–1.83)

Feature: Conditional
 Syntax: `(?{1}yes:no)` through `(?{99}yes:no)`
 Description: Conditional referencing a numbered capturing group. Inserts the “yes” part if the group participated or the “no” part if it didn’t.
 Example: Replacing all matches of `(y)?|n` in `yyn!` with `(?{1}yes:no)` yields `yesyesno!`
 Supported by: JGsoft (V2); Boost (all; 1.42–1.83)

Feature: Conditional
 Syntax: `{1:+yes:no}` through `{99:+yes:no}`
 Description: Conditional referencing a numbered capturing group. Inserts the “yes” part if the group participated or the “no” part if it didn’t.
 Example: Replacing all matches of `(y)?|n` in `yyn!` with `{1:+yes:no}` yields `yesyesno!`
 Supported by: JGsoft (V2); PCRE2 (extended)

Feature: Conditional
 Syntax: `#{1:-no}` through `#{99:-no}`
 Description: Conditional referencing a numbered capturing group. Inserts the text captured by the group if it participated or the contents of the conditional if it didn't.
 Example: Replacing all matches of `(y)?|n` in `yn!` with `#{1:-no}` yields `ynno!`
 Supported by: JGsoft (V2); PCRE2 (extended)

Feature: Conditional
 Syntax: Any numbered conditional
 Description: A conditional that references the number of a capturing group that does not exist acts as a conditional to a group that never participates.
 Example: Replacing all matches of `(y)?|n` in `yn!` with `(?9yes:no)` yields `nonono!`
 Supported by: JGsoft (V2 error); PCRE2 (extended error); Boost (all)

Feature: Conditional
 Syntax: `?{name}yes:no`
 Description: Conditional referencing a named capturing group. Inserts the “yes” part if the group participated or the “no” part if it didn't.
 Example: Replacing all matches of `(?'one'y)?|n` in `yn!` with `?{one}yes:no` yields `yesyesno!`
 Supported by: Boost (all; 1.42–1.83)

Feature: Conditional
 Syntax: `(?{name}yes:no)`
 Description: Conditional referencing a named capturing group. Inserts the “yes” part if the group participated or the “no” part if it didn't.
 Example: Replacing all matches of `(?'one'y)?|n` in `yn!` with `(?{one}yes:no)` yields `yesyesno!`
 Supported by: JGsoft (V2); Boost (all; 1.42–1.83)

Feature: Conditional
 Syntax: `#{name:+yes:no}`
 Description: Conditional referencing a named capturing group. Inserts the “yes” part if the group participated or the “no” part if it didn't.
 Example: Replacing all matches of `(?'one'y)?|n` in `yn!` with `#{one:+yes:no}` yields `yesyesno!`
 Supported by: JGsoft (V2); PCRE2 (extended)

Feature: Conditional
 Syntax: `#{name:-no}`
 Description: Conditional referencing a named capturing group. Inserts the text captured by the group if it participated or the contents of the conditional if it didn't.
 Example: Replacing all matches of `(?'one'y)?|n` in `yn!` with `#{one:-no}` yields `ynno!`
 Supported by: JGsoft (V2); PCRE2 (extended)

Feature: Conditional
 Syntax: Any named conditional
 Description: A conditional that references the name of a capturing group that does not exist is treated as literal text.
 Example: Replacing all matches of `(y)?|n` in `yn!` with `(?{name}yes:no)` yields `?{name}yes:no?{name}yes:no!`
 Supported by: JGsoft (V2 error); PCRE2 (extended error); Boost (all; 1.42–1.83)

Index

- \$. *see* dollar sign, *see* dollar sign
- .rba files, 152
- .rbg files, 118
- .rbl files, 114
- 3rd party integration, 147
- .rbl files, 153
- .rbg files, 153
- \. *see* backslash
- ^. *see* caret
- .. *see* dot
- |. *see* vertical bar
- ?. *see* question mark
- *. *see* star
- +. *see* plus
- (. *see* parenthesis
-). *see* parenthesis
- [. *see* square brackets
-]. *see* square brackets
- {. *see* curly braces
- }. *see* curly braces
- \t. *see* tab
- \r. *see* carriage return
- \n. *see* line feed
- \a. *see* bell
- \e. *see* escape
- \f. *see* form feed
- \v. *see* vertical tab
- \d. *see* digit
- \D. *see* digit
- \w. *see* word character
- \W. *see* word character
- \s. *see* whitespace
- \S. *see* whitespace
- \c. *see* control characters *or* XML names
- \C. *see* control characters *or* XML names
- \i. *see* XML names
- \I. *see* XML names
- \. *see* start file
- \'. *see* end file
- \b. *see* word boundary
- \y. *see* word boundary
- \m. *see* word boundary
- \<. *see* word boundary
- \>. *see* word boundary
- {. *see* curly braces
- \1. *see* backreference
- \K. *see* keep
- \G. *see* previous match
- \. *see* backslash
- &. *see* ampersand
- \t. *see* tab
- \n. *see* line feed
- \a. *see* bell
- \f. *see* form feed
- \v. *see* vertical tab
- .bak files, 137
- ^\$ **don't match at line breaks**, 20, 59
- ^\$ **match at line breaks**, 19, 59
- 8-bit code page, 27, 64
- actions, 15
- ActionScript, 109
- Add button, 112
- Add empty strings**, 21
- Add groups**, 21
- Advanced Regular Expressions, 410, 431, 439
- alert, 25, 63
- Allow duplicate names**, 20
- Allow zero-length matches**, 20
- alnum, 293
- alpha, 293
- alternation, 47, 219
 - POSIX, 409
- anchor, 213, 256, 298
- anchors, 43
- ANSI, 27, 64
- any character, 210
- any line break, 26
- API, 148
- application integration, 147
- applications, 11
 - compare, 78
- Arabic, 140
- ARE, 410, 431, 439
- arguments
 - command line, 147
- array, 17, 104
- ASCII, 27, 64, 196, 293
- assertion, 256
- asterisk. *see* star
- atomic
 - recursion, 288
- atomic group, 55

- automatic line breaks, 88
- automatic save, 111
- awk, 377
- Awk**, 19
- \b. *see* word boundary
- backreference
 - in a character class, 228
 - number, 226
 - recursion, 285
 - relative, 234
 - repetition, 354
- backreferences, 50, 68
- backslash, 108, 194, 195, 301
 - in a character class, 201
- backspace, 25, 63
- backtick, 214
- backtracking, 45, 223, 340
 - recursion, 288
- backup, 117
- backup files, 137
- bak files, 137
- balanced constructs, 51, 267, 271
- balancing groups, 267
- Basic**, 19, 108
- Basic Regular Expressions, 377, 408, 431
- begin file, 214
- begin line, 213
- begin string, 213
- Beginning of a line**, 43
- Beginning of a word**, 43
- Beginning of the string**, 43
- bell, 25, 63, 196, 303
- benchmark, 95
- bidirectional, 141
- binary data, 86
- blank, 293
- blinking, 146
- block
 - Unicode, 32
- Borland, 428
- braces. *see* curly braces
- bracket expressions, 292
- brackets. *see* square brackets *or* parenthesis
- branch reset, 235
- BRE, 377, 408, 431
- brief, 75, 79
- browsers, 384
- \c. *see* control characters *or* XML names
- C, 108
- \C. *see* control characters *or* XML names
- C#, 108, 147, *see* .NET
- C/C++, 396, 398
- C++, 108, 428
- C++11, 108, 428
- callback function, 104
- canonical equivalence
 - Java, 247
- capturing group, 48, 225
 - recursion, 273, 281
- capturing group subtraction, 267
- caret, 194, 213
 - in a character class, 201
- carriage return, 20, 25, 63, 196, 211
- case insensitive**, 59
 - Perl, 402
- Case insensitive**, 19
- case sensitive, 59
- Case sensitive**, 19
- Case sensitivity**, 19
- catastrophic backtracking, 340
- char, 108
- character class, 38, 201
 - intersection, 39, 206
 - negated, 201
 - negated shorthand, 208
 - repeating, 202
 - shorthand, 36, 208
 - special characters, 201
 - subtract, 204
 - subtraction, 39
 - XML names, 209
- character equivalents, 294
- character range, 201
- character set. *see* character class
- characters, 25, 63, 194, 301
 - ASCII, 196
 - categories, 30, 241
 - control, 196
 - digit, 208
 - in a character class, 201
 - invisible, 25, 63, 196
 - metacharacters, 194
 - non-printable, 25, 63, 196
 - non-word, 208, 216
 - special, 25, 194, 301
 - Unicode, 34, 66, 196, 240
 - whitespace, 208
 - word, 208, 216
- check, 85
- choice, 47, 219
- Chrome, 108
- class, 201
- Clear button, 118

- clipboard, 86, 107
- clipboard export, 81
- closing bracket, 243
- closing quote, 243
- cntrl, 293
- code editor, 101
- code page, 27, 64, 356, 357
- code point, 241
- code snippets, 101
- collapse, 93
- collating sequences, 293
- collect information, 413
- colors, 138
- COM Automation interface, 154
- combining character, 242
- combining mark*, 240
- combining multiple regexes, 219, 336
- command line examples, 147
- command line parameters, 149
- comment, 19, 59, 60, 337
- comments, 83, 238
- compare, 78
- compatibility, 191
- complex script, 140
- condition
 - version, 398
- conditional, 264
 - replacement, 310
- conditionals, 54, 71
- conditions
 - many in one regex, 260
- configure RegexBuddy, 133
- Consecutive Character in XML Name**, 36
- continue
 - from previous match, 298
- control characters, 196, 243
- Convert panel, 82
- copy, 107
- copy all searched files, 117
- copy only modified files, 117
- count matches, 89
- CR only**, 20, 88
- CR, LF, or CRLF**, 20
- Create panel, 23, 62, 75, 81
- CRLF pair, 88, 197
- CRLF pairs only**, 20
- cross. *see plus*
- Ctrl+C, 107
- Ctrl+V, 107
- curly braces, 194, 222
- currency sign, 242
- cursor, 141
- cursor configuration, 144
- \d. *see digit*
- \D. *see digit*
- dash, 243
- data, 191
- database
 - MySQL, 387
 - Oracle, 393
 - PostgreSQL, 410
- date, 327, 329
 - to text, 329
 - validate, 327
- Debug button, 92
- Debug panel, 93
- debugger, 92
- Default flavor**, 19
- Default line breaks**, 20
- default value for parameters, 113
- delete backup files, 118
- Delphi, 108, 147
- denial of service, 350
- description, 112
- detailed, 75, 79
- details of a match, 103
- diff, 78
- digit, 208, 242, 293
- Digit**, 36
- Dinkumware, 428
- distance, 339
- documentation, 81
- dollar sign, 194, 213, 301
- Don't add empty strings**, 21
- Don't add groups**, 21
- dot, 194, 210, 302
 - misuse, 341
- ^\$ match at line breaks**, 20
- dot doesn't match line breaks, 59
- Dot doesn't match line breaks**, 19
- ^\$ don't match at line breaks**, 20
- dot matches line breaks**, 59
- Dot matches line breaks**, 19
- dot net. *see .NET*
- double quote, 108, 195, 302
- duplicate items, 335
- duplicate lines, 335
- Duplicate names**, 20
- eager, 200, 219
- ECMA-262, 109
- ECMAScript**, 19, 109, 384
- ed, 377
- EditPad Lite, 370

- EditPad Pro, 192, 372
- efficiency, 95
- egrep, 375, 377
- EGrep**, 19
- else, 264
 - replacement, 310
- emacs, 377
- email address, 320
- emulation, 13
- enclosing mark, 242
- encoding, 27, 64, 86
- end file, 214
- end line, 213
- End of a line**, 43
- End of a word**, 43
- end of line, 196, 303
- End of the previous match**, 43
- End of the string**, 43
- End of the string or before the final line break**, 43
- end string, 213
- endless recursion, 277
- engine, 191, 199
- entire string, 213
- entirely matching, 103
- ERE, 377, 408, 431
- ereg, 109, 406
- escape, 25, 63, 108, 194, 195, 196
 - in a character class, 201
- euro, 356
- exact spacing, 59, 83, 107
- Exact spacing**, 19
- example
 - combining multiple regexes, 336
 - date, 327, 329
 - duplicate items, 335
 - duplicate lines, 335
 - exponential number, 319, 337
 - floating point number, 319
 - HTML tags, 315
 - integer number, 337
 - keywords, 338
 - multi-line comment, 337
 - not meeting a condition, 333
 - number, 337
 - numeric ranges, 317
 - prepend lines, 295
 - programming languages, 336
 - quoted string, 212, 337
 - reserved words, 338
 - scientific number, 319, 337
 - single-line comment, 337
 - source code, 336
 - trimming whitespace, 315
 - whole line, 333
- examples
 - command line, 147
- exception handling, 101, *see* exception handling, *see* exception handling
- exclude files, 137
- exclude folders, 138
- execute, 118
- expand, 93
- explain token, 75
- explicit capture, 20
- Explicit capture**, 20
- export, 81
- Export button, 118
- Extended**, 19
- Extended Regular Expressions, 377, 408, 431
- feeds, 126
- file
 - exclude, 137
 - hide, 137
- file mask, 116
- files
 - hidden, 137
 - system, 137
- find, 15
- find first, 90, 103
- Find First button, 90
- find next, 90
- Find Next button, 90
- flag, 145
- flavor, 191
- Flavor**, 19
- flavors
 - compare, 78
- floating point number, 319
- fold, 93
- folder
 - exclude, 138
 - hide, 138
- folders, 116
- font, 142
- form feed, 20, 25, 26, 36, 63, 196, 211, 303
- free-spacing**, 59, 83, 107, 237
- Free-spacing**, 19
- Free-spacing [...]**, 19
- Free-spacing mode**, 19
- full stop. *see* dot
- generate regex, 61
- GNU, 377

- GNU grep, 375
- Gnulib, 378
- graph, 293
- grapheme, 33, 240
- greedy, 45, 221, 222
- Greedy quantifiers**, 20
- grep, 375, 377
 - multi-line, 412
 - PowerGREP, 412
- Grep**, 19
- GREP button, 118
- GREP panel, 116
- Groovy, 109, 379
- group, 48, 55, 225
 - atomic, 55
 - capturing, 48, 225
 - in a character class, 228
 - named, 48, 231
 - nested, 340
 - non-capturing, 55
 - numbered, 48
 - repetition, 354
- handle exceptions, 101
- Hebrew, 140
- Helpful, 13
- Henry Spencer, 410, 431, 439
- hexadecimal, 86
- Hexadecimal Digit**, 36
- hidden files, 137
- hidden history, 117
- hide files, 137
- hide folders, 138
- highlight, 89
- Highlight button, 89, 138
- Horizontal Space Character**, 36
- horizontal whitespace, 208
- HTML file export, 81
- HTML tags, 315
- hyphen, 243
 - in a character class, 201
- \i. *see* XML names
- \I. *see* XML names
- IDE, 101
- identifiers, 165
- if-then-else, 264
 - replacement, 310
- ignore whitespace, 19, 59
- implementation, 101
- include binary files, 116
- Indic, 140
- infinite recursion, 277
- information
 - collecting, 413
- Initial Character in XML Name**, 36
- insert regex, 61
- insert token, 23, 62
- integer number, 337
- integration with other tools, 147
- intersect character classes, 206
- intersected character class, 39
- invert mask, 116
- invert results, 116
- invisible characters, 25, 63, 196
- Java, 109, 381
 - literal strings, 383
 - Matcher class, 382
 - Pattern class, 382
 - String class, 381
- java.util.regex, 381
- JavaScript, 109, 384
- JDK 1.4, 381
- JScript, 109
- keep, 262
- keep comments, 83
- keyboard shortcuts, 127, 131
- keywords, 338
- language
 - C/C++, 396, 398
 - ECMAScript, 384
 - Groovy, 379
 - Java, 381
 - JavaScript, 384
 - Perl, 402
 - PHP, 404
 - Python, 417
 - Ruby, 426
 - Tcl, 431
 - VBScript, 435
 - Visual Basic, 438
- languages, 11, 101, 107
- last backreference, 69
- launch RegexBuddy, 147
- lazy, 45, 223
 - better alternative, 223
- Lazy quantifiers**, 20
- leftmost match, 199
- left-to-right, 140, 145
- legend, 451
- letter, 242, *see* word character
- LF only**, 20, 88
- libraries, 101
- library, 111
 - parameters, 113

- Library panel, 114
- line, 213
 - begin, 213
 - duplicate, 335
 - end, 213
 - not meeting a condition, 333
 - prepend, 295
- line break, 26, 196, 197, 210, 303
- Line break handling**, 20
- line breaks, 133
- line by line, 87
- line feed, 20, 25, 63, 196, 303
- line separator, 20, 26, 36, 211, 242
- line terminator, 196, 303
- line-based, 116
- Linux, 377
- Linux grep, 375
- list all, 90
- List All button, 90
- literal characters, 25, 63, 194, 301
- literal text, 25, 63
- locale, 292
- longest, 220
- lookahead, 256
- lookaround, 57, 256
 - many conditions in one regex, 260
- lookbehind, 257, 262
 - limitations, 258
- lower, 293
- lowercase letter, 242
- Lowercase Letter**, 36
- \m. *see* word boundary
- many conditions in one regex, 260
- mark, 242
- match, 15, 101, 191
- match details, 103
- match line breaks, 19, 59
- match mode, 249
- matched text, 68
- matching entirely, 103
- mathematical symbol, 242
- maximum, 45
- mb_ereg, 406
- metacharacters, 25, 194
 - in a character class, 201
- Microsoft .NET. *see* .NET
- minimum, 45
- mixed, 88
- mode modifier, 249
- mode modifiers
 - PostgreSQL, 431
 - Tcl, 431
- mode span, 250
- modes, 18, 135
- modifier, 249
- modifier span, 250
- modify libraries, 111
- monospaced, 140
- More applications and languages, 12
- multi .bak, 117
- multi backup N of, 117
- multi-line comment, 337
- multi-line grep, 412
- multi-line mode, 20, 59, 214
- Multi-line mode**, 19
- multiple regexes, 336
- multiple regexes combined, 219
- MySQL, 109, 387
- .NET, 389
 - Core, 389
- n/a, 91
- named backreference, 50, 68
- Named capture only**, 20
- named conditional, 54, 71
- named group, 231
- named subroutine call, 52
- Names must be unique**, 20
- near, 339
- negated character class, 201
- negated shorthand, 208
- negative lookahead, 256
- negative lookbehind, 257
- nested constructs, 267, 271
- nested grouping, 340
- nested pairs, 51
- newline, 210
- news feeds, 126
- Next button, 89
- next line, 20, 26, 36
- no backups, 117
- Non a word boundary**, 43
- non-capturing group, 55, 225
- non-participating groups, 90
- non-printable characters, 25, 196
- non-spacing mark, 242
- NULL, 91
- number, 208, 242, 337
 - backreference, 226
 - exponential, 319, 337
 - floating point, 319
 - range, 317
 - scientific, 319, 337
- number of matches, 89

- numbered backreference, 50, 68
- Numbered capture**, 20
- numbered capturing group, 48, 225
- numbered conditional, 54, 71
- numbered subroutine call, 52
- numeric ranges, 317
- object with regex, 105
- octal escapes, 197
- once or more, 222
- Open button, 111, 114, 118
- open file, 86
- open libraries, 111, 114
- opening bracket, 243
- opening quote, 243
- operator, 109
- optimization, 95
- option, 47, 219, 221, 222
- options, 18, 133, 135
- or
 - one character or another, 201
 - one regex or another, 47, 219
- Oracle, 393
- other character class tokens, 38
- Oxygene, 108
- page break, 87
- page by page, 87
- palindrome, 270, 283, 289
- paragraph separator, 20, 26, 36, 211, 242
- parameterize a regex, 113
- parentheses, 225
- parser, 336
- Pascal, 108
- paste, 107
- paste subject, 86
- pattern, 191
- PCRE, 396
- PCRE_NOTEMPTY, 295
- PCRE2, 398
- period. *see* dot
- Perl, 109, 402
- Perl-compatible, 396
- permanent exclusion, 137
- PHP, 109, 404
 - ereg, 406
 - mb_ereg, 406
 - preg, 404
 - split, 406
- pipe symbol. *see* vertical bar
- plus, 194, 222
 - possessive quantifiers, 253
- position, 43
- positive lookahead, 256
- positive lookbehind, 257
- POSIX, 41, 292, 408
- possessive, 45, 253
- PostgreSQL, 109, 410
- PowerGREP, 412
- PowerShell, 109
- precedence, 219, 225
- preferences, 133
- preg, 109, 404
- prepend lines, 295
- preserve state, 136
- preview, 118
- Previous button, 89
- previous match, 298
- print, 293
- Prism, 108, 369
- programming
 - Groovy, 379
 - Java, 381
 - MySQL, 387
 - Oracle, 393
 - Perl, 402
 - PostgreSQL, 410
 - Tcl, 431
 - wxWidgets, 439
- programming languages, 11, 101, 107, 336
- properties
 - Unicode, 30, 241
- punct, 293
- punctuation, 243
- Python, 109, 417
- quantifier, 45
 - backreference, 354
 - backtracking, 223
 - curly braces, 222
 - greedy, 222
 - group, 354
 - lazy, 223
 - nested, 340
 - once or more, 222
 - once-only, 253
 - plus, 222
 - possessive, 253
 - question mark, 221
 - reluctant, 223
 - specific amount, 222
 - star, 222
 - ungreedy, 223
 - zero or more, 222
 - zero or once, 221
- question mark, 194, 221

- common mistake, 319
- lazy quantifiers, 223
- quick execute, 118
- quirks, 384
- quote, 108, 195, 302
- quoted string, 212, 337
- raise. *see* exception handling
- range of characters, 201
- raw string, 109
- rba files, 152
- rbg files, 118, 153
- rbl files, 114, 153
- read only, 111
- REALbasic, 108
- recursion, 51, 271
 - atomic, 288
 - backreference, 285
 - backtracking, 288
 - capturing group, 273, 281
- ReDoS, 350
- regenerate regex, 61
- regex engine, 199
- regex flavor, 11
- regex flavors
 - compare, 78
- regex modes, 18, 135
- regex object, 105
- regex options, 18, 135
- regex structure, 75
- regex syntax, 11
- Regex syntax only**, 21
- regex template, 113
- regex tool, 412
- regex tree, 75, 81
- regex variants, 113
- regex with parameters, 113
- RegexBuddy Library file, 111
- RegexBuddy4.rbl, 114
- regex-directed engine, 199
- RegexMagic, 61, 424
- regular expression, 191
- reluctant, 223
- remember state, 136
- repetition, 45
 - backreference, 354
 - backtracking, 223
 - curly braces, 222
 - greedy, 222
 - group, 354
 - lazy, 223
 - nested, 340
 - once or more, 222
 - once-only, 253
 - plus, 222
 - possessive, 253
 - question mark, 221
 - reluctant, 223
 - specific amount, 222
 - star, 222
 - ungreedy, 223
 - zero or more, 222
 - zero or once, 221
- replace, 16, 101, 104
- replace all, 91
- Replace All button, 91
- replacement
 - last backreference, 69
 - matched text, 68
- replacement flavor, 11
- requirements
 - many in one regex, 260
- reserved characters, 25, 194, 301
- reset, 235
- Reset button, 22
- reuse, 111
 - part of the match, 226
- right-to-left, 140, 145
- round bracket, 194
- round brackets, 225
- RSS feeds, 126
- Ruby, 109, 426
- \s. *see* whitespace
- \S. *see* whitespace
- same file name, 117
- save, 81
- Save As button, 112
- Save button, 118
- save libraries, 111
- save one file for each searched file, 116
- save regexes, 111
- save results into a single file, 116
- save state, 136
- sawtooth, 98, 346
- Scala, 109
- script, 31, 243
- search, 15
- search and replace, 192, 413
 - preview, 413
 - text editor, 371, 373
- sed, 377
- send, 107
- separator, 242
- several conditions in one regex, 260

- share regexes, 111
- shortcuts, 131
- shorthand character class, 36, 208
 - negated, 208
 - XML names, 209
- show non-participating groups, 90
- show spaces, 133
- single `~??`, 117
- single `.bak`, 117
- single line mode, 19, 59
- single quote, 108, 195, 214, 302
- single-line comment, 337
- single-line mode, 210
- Single-line mode**, 19
- Skip zero-length matches**, 20
- software integration, 147
- source code, 101, 336
- space, 293
- space separator, 208, 242
- spaces, 133
 - ignore, 59
- spacing, 19, 83
- spacing combining mark, 242
- special characters, 25, 194, 301
 - in a character class, 201
 - in programming languages, 195, 302
- specific amount, 222
- split, 17, 91, 101, 104
- Split button, 91
- Split capture**, 21
- Split empty**, 21
- split flavor, 11
- Split limit**, 21
- Split with limit:**, 21
- Split without limit**, 21
- SQL, 109, 387, 393, 410
- square bracket, 201
- square brackets, 194
- star, 194, 222
 - common mistake, 319
- start file, 214
- start line, 213
- Start of the match attempt**, 43
- start string, 213
- state, 136
- statistics, 413
- statusbar, 136
- stopwatch, 95
- store regexes, 111
- Strict, 13
- string, 191, 337
 - begin, 213
 - end, 213
 - matching entirely, 213
 - quoted, 212
- String syntax**, 20
- strip comments, 83
- structure, 75
- subroutine call, 52
- subroutine calls, 273
- subtract character class, 204
- subtracted character class, 39
- Support string syntax**, 20
- surrogate, 243
- symbol, 242
- syntax coloring, 138, 373
- system files, 137
- System.Text.RegularExpressions, 389
- tab, 25, 63, 196, 303
- tabs, 133
- target, 116
- target application, 11
- task, 101
- Tcl, 110, 431
 - word boundaries, 217
- teaching materials, 81
- template regex, 113
- terminate lines, 196, 303
- test, 85
- Test panel, 86
- text, 25, 63, 191
 - encoding, 86
- text cursor configuration, 144
- text editor, 192, 370, 372
- text encoding, 356
- text file export, 81
- text layout, 139
- text mode, 88
- text-directed engine, 199
- throw. *see* **exception handling**
- titlecase letter, 242
- token, 23, 62, 75
- tool
 - EditPad Lite, 370
 - EditPad Pro, 372
 - egrep, 375
 - GNU grep, 375
 - grep, 375
 - Linux grep, 375
 - PowerGREP, 412
 - RegexMagic, 424
 - specialized regex tool, 412
 - text editor, 370, 372
- tool integration, 147

- TR1, 428
- transfer, 107
- tree, 75
- trimming whitespace, 315
- turn off modes, 59
- tutorial, 191
- type library, 154
- underscore, 208
- undo GREP, 118
- unfold, 93
- ungreedy, 223
- Unicode, 240
 - block, 32
 - blocks, 244
 - canonical equivalence, 247
 - categories, 30, 241
 - characters, 34, 66, 240
 - code point, 241
 - combining mark*, 240
 - grapheme, 33, 240
 - normalization, 247
 - properties, 30, 241
 - ranges, 244
 - scripts, 31, 243
- Unicode line breaks**, 20
- Unix, 377
- UNIX grep, 375
- UNIX_LINES, 211
- unlimited, 45
- update automatically, 91
- upper, 293
- uppercase letter, 242
- Uppercase Letter**, 36
- use, 101
- Use button, 114
- variants of regexes, 113
- VB, 438
- VBScript, 435
- verbatim string, 108
- verify, 85
- version
 - PCRE2, 398
- vertical bar, 47, 194, 219
 - POSIX, 409
- Vertical Space Character**, 36
- vertical tab, 20, 25, 26, 36, 63, 196, 211, 303
- Visual Basic, 108, 147, 438
- Visual Basic.NET. *see* .NET
- Visual Studio, 147
- visualize spaces, 133
- \w. *see* word character
- \W. *see* word character
- W3C, 443
- wchar_t, 108
- whitespace, 19, 83, 208, 242, 315
 - ignore, 59
- Whitespace Character**, 36
- whole file, 86
- whole line, 213, 333
- whole word, 216, 217
- whole words only, 44
- word, 142, 216, 217, 293
- word boundary, 216
 - Tcl, 217
- Word boundary**, 43
- word character, 208, 216
- Word Character**, 36
- words
 - keywords, 338
- workflow, 147
- working copy, 137
- wrap, 133
- wxWidgets, 439
- xdigit, 293
- XML, 110, 443
- XML names, 209
- Xojo, 108
- \y. *see* word boundary
- zero or more, 222
- zero or once, 221
- zero-length, 213, 256
- zero-length match, 295
- Zero-length matches**, 20