B
Regular Expressions

ModSecurity rules rely heavily on regular expressions to allow you to specify when a rule should or shouldn't match. This appendix teaches you the basics of regular expressions so that you can better make use of them when writing ModSecurity rules. Regular expressions are also useful in a number of other areas, including programming, text processing, and Linux administration, so learning about them is definitely worth the investment in time.

What is a regular expression?

A regular expression (often abbreviated as regex or regexp) is a way to identify strings of interest. Regular expressions can be used to search through large amounts of text to find specific strings, or to make sure a particular string matches a given pattern. In the case of ModSecurity, regular expressions are used by rules to define when and how the rule matches. Although there are other operators available for use with rules (`@streq`, `@contains`, `@gt`, and others), the regular expression operator is the default one used, which goes to show how important regular expressions are and that knowledge of them is important to being able to fully use ModSecurity.

Regular expression flavors

There are many different "dialects" of regular expressions, each with slightly different nuances and supported constructs. Here are just a few of the different flavors of regex engines available:

- Perl
- PCRE
- POSIX
- .NET
- Python
- Java

As an example of the differences, the Perl regex engine supports character classes such as `[:alpha:]`, which denote an alphanumeric character. The Java regex engine, on the other hand does not support this. For a table listing regular expression features and which dialects support them see `http://www.regular-expressions.info/refflavors.html`.

The regular expression flavor used by Apache, and hence by ModSecurity since they are compiled using the same library, is called **Perl-Compatible Regular Expressions** (**PCRE**). This is a library developed by Philip Hazel and used by many open source projects which require regular expression support.

As the name implies, PCRE aims to be compatible with the Perl regular expression engine (which is so tightly integrated into the massive Perl programming language that trying to extract just the regular expression engine into a library would be near impossible). Since ModSecurity uses PCRE, this is the dialect you should be looking up online if you ever have any question about why a regex doesn't work the way you expect it to—it may be that PCRE syntax is different from what you are using.

Example of a regular expression

To get a feeling for how regular expressions are used, let's start with a real-life example so that you can see how a regex works when put to use on a common task.

Identifying an email address

Suppose you wanted to extract all email addresses from an HTML document. You'd need a regular expression that would match email addresses but not all the other text in the document. An email address consists of a username, an `@` character, and a domain name. The domain name in turn consists of a company or organization name, a dot, and a top-level domain name such as `com`, `edu`, or `de`.

Knowing this, here are the parts that we need to put together to create a regular expression:

- **User name**

  This consists of alphanumeric characters (`0-9`, `a-z`, `A-Z`) as well as dots, plus signs, dashes, and underscores. Other characters are allowed by the RFC specification for email addresses, but these are very rarely used so I have not included them here.

- **@ character**

  One of the mandatory characters in an email address, the `@` character must be present, so it is a good way to help distinguish an email address from other text.

- **Domain Name**

  For example `cnn.com`. Could also contain sub-domains, so `mail.cnn.com` would be valid.

- **Top-Level Domain**

  This is the final part of the email address, and is part of the domain name. The top-level domain usually indicates what country the domain is located in (though domains such as `.com` or `.org` are used in countries all around the world). The top-level domain is between two and four characters long (excluding the dot character that precedes it).

Putting all these parts together, we end up with the following regular expression:

```
\b[-\w.+]+@[\w.]+\.[a-zA-Z]{2,4}\b
```

The `[-\w.+]+` part corresponds to the username, the `@` character is matched literally against the `@` in the email address, and the domain name part corresponds to `[\w.]+\.\w{2,4}`. Unless you are already familiar with regular expressions, none of this will make sense to you right now, but by the time you've finished reading this appendix, you will know exactly what this regular expression does.

Let's get started learning about regular expressions, and at the end of the chapter we'll come back to this example to see exactly how it works.

The Dot character

One of the most ubiquitous characters in regular expressions is the dot. It matches any character, so the regex `d.g` will match both `dog` and `dig`. (Actually, there is one exception to "dot matches all", and that is a newline character or pair of characters—this is usually not matched by dot unless specially configured in the regex engine's options. In the case of ModSecurity and PCRE, the "dot matches all" flag *is* set at compile time, so a dot when used in a ModSecurity rule will really match any character.)

The fact that dot matches anything means that you need to be careful using it in things such as IP addresses as for example the regex `1.2.2.33` will match not only the IP address 1.2.2.33 but also the first part of addresses such as 1.222.33.45.

The solution is to *escape* the dot by prefixing it with a backslash. The backslash means that the next character should be interpreted literally, and hence the dot will only match an actual dot when preceded by a backslash. So to match only the IP address 1.2.2.33 and nothing else, you would use the regex `1\.2\.2\.33` which will avoid any unpleasant surprises.

Quantifiers—star, plus, and question mark

When you want to match a character or pattern more than once, or want to make characters or patterns optional, the star, plus, and question mark sign are exactly what's needed. These characters are called quantifiers. They are also known as metacharacters since for example the plus sign (as we will see) does not match a literal plus sign in a string, but instead means something else.

Question Mark

The question mark is used to match something zero or one time, or phrased differently, it makes a match optional.

A simple example is the regex `colou?r`, which matches both the US spelling of color as well as the British colour. The question mark after the `u` makes the `u` optional, meaning the regex will match both with the `u` present and with it absent.

Star

The star means that something should match *zero or more* times. For example, this regex can be used to match the string `Once upon a time`:

```
Once .* a time
```

The dot matches anything, and the star means that the match should be made zero or more times, so this will end up matching the string `Once upon a time`. (And any other string beginning with "`Once `" and ending with "` a time`".)

Plus sign

The plus sign is similar to the star—it means "match one or more times", so the difference to the star is that the plus must match at least one character. In the previous example, if the regex had instead been `Once upon a time.+` then the string `Once upon a time` would not match any longer, as one or more additional characters would be required.

Grouping

If you wanted to create a regex that allows a word to appear one or more times in a row, for example to match both `A really good day` and `A really really good day` then you would need a way to specify that the word in question (`really` in this case) could be repeated. Using the regex `A really+ good day` would not work, as the plus quantifier would only apply to the character `y`. What we'd want is a way to make the quantifier apply to the whole word (including the space that follows it). The solution is to use grouping to indicate to the regex engine that the plus should apply to the whole word. Grouping is achieved by using standard parentheses, just as in the alternation example above.

Knowing this, we can change the regex so that `really` is allowed more than once:

```
A (really )+good day
```

This will now match both `A really good day` and `A really really good day`. Grouping can be used with any of the quantifiers—question mark, star, the plus sign, and also with ranges, which is what we'll be learning about next.

Ranges

The star and plus quantifiers are a bit crude in that they match an unlimited number of items. What if you wanted to specify *exactly* how many times something should match. This is where the *interval quantifier* comes in handy—it allows you to specify a range that defines exactly how many times something should match.

Suppose that you wanted to match both the string `Hungry Hippos` as well as `Hungry Hungry Hippos`. You could of course make the second Hungry optional by using the regex `Hungry (Hungry )?Hippos`, but with the interval quantifier the same effect can be achieved by using the regex `(Hungry ){1,2}Hippos`.

The word `Hungry` is matched either one or two times, as defined by the interval quantifier `{1,2}`. The range could easily have been something else, such as `{1,5}`, which would have made the hippos very hungry indeed, as it would match `Hungry` up to five times.

Note that the parentheses are required in this case—using `Hungry{1,2}` without the parentheses would have been incorrect as that would have matched only the character `y` one or two times. The parentheses are required to group the word `Hungry` so that the `{1,2}` range is applied to the whole word.

You can also specify just a single number, like so:

```
(Hungry ){2} Hippos
```

This matches "Hungry" exactly twice, and hence will match the phrase `Hungry Hungry Hippos` and nothing else.

The following table summarizes the quantifiers we have discussed so far:

Quantifier	Meaning
`*`	Match the preceding character or sequence 0 or more times.
`?`	Match the preceding character or sequence 0 or 1 times.
`+`	Match the preceding character or sequence 1 or more times.
`{min,max}`	Match the preceding character or sequence at least *min* times and at most *max* times.
`{num}`	Match the preceding character or sequence exactly *num* times.

Alternation

Sometimes you want to match one of several phrases. For example, maybe you want to match against `Monday` written in one of several languages. The pipe character `|` can be used for this purpose, in the following manner:

```
Monday|Montag|Lundi
```

This regex matches either one of `Monday`, `Montag`, and `Lundi`. The pipe character is what makes each of the words an alternative—it can be thought of as an "or" construct if you are familiar with programming.

So how far does alternation reach? In the regex `I remember the day, it was a Monday|Montag|Lundi`, does the first alternative refer to `Monday`, `it was a Monday`, or something else? The answer is that the first alternative will be the entire first part of the sentence, namely `I remember the day, it was a Monday`.

This is obviously not what we want from this regex, so we need a way to constrain what the alternation matches. This is done by using parentheses, in the following way:

```
I remember the day, it was a (Monday|Montag|Lundi)
```

The parentheses in this regex make sure that the alternation only applies within the parentheses, so the first alternative will be restricted to `Monday` and not anything without the parentheses. Similarly, the last alternative will be `Lundi` only and will not include anything following it.

Backreferences

Backreferences are used to capture a part of a regular expression so that it can be referred to later. The regex `Hello, my name is (.+)` will capture the name into a variable that can be referred to later. The reason for this is that the `.+` construct is surrounded by parentheses.

The name of the variable that the matched text is captured into will differ depending on what regex flavor you are working with. In Perl, for example, regex backreferences are captured into variables named `$1`, `$2`, `$3`, and so on.

Captures are made left-to-right, with the text within the first parentheses captured into the variable `$1`, the second into `$2`, and so forth. Capturing can even be made within a set of parenthesis, so the regex `My full name is ((\w+) \w+)` would store the complete name (first and last) into `$1` and the first name only into `$2`.

These are the same kind of parenthesis used for grouping, so grouping using standard parenthesis will also create a backreference. We will however shortly see how to achieve grouping without capturing backreferences.

Captures and ModSecurity

To use captured backreferences in a ModSecurity rule, you specify the `capture` action in the rule, which makes the captured backreferences available in the transaction variables `TX:1` through `TX:9`.

The following rule uses a regex that looks for a browser name and version number in the request headers. If found, the version number is captured into the transaction variable `TX:1` (which is accessed using the syntax `%{TX.1}`) and is subsequently logged to the error log file:

```
SecRule REQUEST_HEADERS:User-Agent "Firefox/(\d\.\d\.\d)" "pass,phase:
2,capture,log,logdata:%{TX.1}"
```

Up to nine captures can be made this way. The transaction variable `TX:0` is used to capture the entire regex match, so in the above example, it would contain something like `Firefox/3.0.9`.

Non-capturing parentheses

Parentheses are used to capture backreferences and also to group strings together (as in the regex `(one|two|three)`). This means that any grouping using parentheses also creates a backreference. Sometimes you want to avoid this and not create a backreference. In this case, *non-capturing parentheses* come in handy. In the example we just saw, the following would group the words, but would *not* create a backreference:

```
(?:one|two|three)
```

The construct `(?:)` is what is used to create a non-capturing set of parenthesis. To further show the difference between the two, consider the following regex:

```
It is (?:hard|difficult) to say goodbye to (.*)
```

When matched against the string `It is hard to say goodbye to you`, this will create a single backreference, which will contain the string `you`. The first, non-capturing parentheses also allow strings beginning with `It is difficult to say goodbye` to match, but they do not create a backreference.

Non-capturing parentheses are sometimes referred to as "grouping-only parentheses".

Character classes

Character classes provide a way to specify that exactly one of a group of characters should be matched against. Character classes are denoted by square brackets — `[]` — that contain characters or ranges of characters to match against.

As an example, the character class `[abc]` will match either `a`, `b`, or `c` exactly once, so the first match when matching against the string `Brothers in arms` would be the `a` in `arms`.

Character classes can contain *ranges* of characters, specified by using a hyphen. One example is the common character class `[a-z]`, which denotes any character between `a` and `z`. A similar class is `[a-zA-Z]` which means any character between `a` and `z`, or `A` and `Z`, matching characters regardless of their case. Note how the first range is `a-z`, and the second range `A-Z` is specified immediately following it without any space or other character in-between.

Ranges work equally well for digits, and `[0-9]` means any digit, whereas `[0-3]` means only the digits `0`, `1`, `2`, and `3`.

Negated matching

You can negate a character class by specifying a caret immediately following the opening bracket. This means that the character class should match only characters not present inside the brackets. For example, the character class `[^a-z]` matches anything that *isn't* a letter from `a` through `z`.

Another thing to keep in mind is that there is no regular expression construct to negate matching of anything more than a single character. So for example it's not possible to have a regex that specifies that any other color than `red` should match in the string `Roses are red`, unless you want to resort to the regex `Roses are [^r][^e][^d].*`. (There *is* something called negative lookahead which can be handy if you really do want to assert that something is not present at a particular position in a regex, but lookaheads are beyond the scope of this book. A simple Google search will enlighten you if you really need this sort of regex.)

ModSecurity does have inverted rule matching using the exclamation mark operator, and this allows you to specify that a rule should match when a regex *isn't* present in the variable being matched against. The following rule, for example, will match if the string Firefox isn't in the user-agent string:

```
SecRule REQUEST_HEADERS:User-Agent "!Firefox" "phase,2"
```

Shorthand notation

There are a number of shorthand notations for common character classes, such as whitespace or digits. These consist of a backslash followed by a letter, and provide a simple way to use a character class without having to type it out in full each time. For example, the class shorthand `\w` means "part-of-word character" and is equivalent to `[a-zA-Z0-9_]`, and will thus match a single letter, digit, or underscore character.

The following table lists the most common shorthand notations available.

Shorthand	Description
`\d`	Matches any digit.
	Equivalent to `[0-9]`.
`\D`	Matches any character that is *not* a digit.
	Equivalent to `[^0-9]`.
`\w`	Matches a word character.
	Equivalent to `[a-zA-Z0-9_]`.

Shorthand	Description
`\W`	Matches anything that is not a word character. Equivalent to `[^a-zA-Z0-9_]`.
`\s`	Matches whitespace (space, tab, newline, form feed, and so on.)
`\S`	Matches anything that is not whitespace. Equivalent to `[^\s]`.

Note that the character class shorthands are case sensitive, and how the upper-case version usually means negation of the character class—for example `\d` means a digit whereas `\D` means any *non*-digit.

Anchors

If you wanted to make sure that a regex matched only if a certain string was present at the start of a line, what would you do? The regex constructs we have seen so far do not provide any way of doing that. You could check for a newline followed by the string, but that would not work if the string was present in the first line of text, as it wouldn't be preceded by a newline. The solution is to use something called *anchors*, which are able to ascertain that the regex matches at a certain position in the string.

Start and end of string

Two special characters are used in regexes to match "start of line or string" and "end of line or string". These are the caret (`^`) and dollar sign (`$`). The caret matches the start of any line or string, so the following regex makes a good example:

```
^Subject:
```

Since the regex starts with a caret, it will match only those instances of `Subject:` which are at the start of a line or string. Note how the caret does not actually match any character in a string—it only matches a *position* in a string. In essence, the caret means "make sure that at this position we are at the start of a line or string".

Similarly, the dollar sign matches "end of line or string". If we were to modify the regex so that it reads as follows, can you figure out what it will match?

```
^Subject:$
```

That's right, since there is now a dollar sign at the end of the regex, it will match only those lines or strings that consist of only the string `Subject:` and nothing else.

You may wonder why I keep saying "line or string". The reason is that the caret and dollar sign behave differently depending on how the regular expression library was compiled. In the case of PCRE, which is the library that ModSecurity uses, the way it works by default means that the caret and dollar sign only match at the beginning or end of the string being examined. So for example when examining a HTML response body by using the `RESPONSE_BODY` variable in a string, the dollar sign will only match at the very end of the string, and not at each linebreak within the HTML document.

Line anchors are often used in ModSecurity rules to ascertain that we are not matching against substrings. For example, if you wanted to have a rule trigger on the IP address `1.2.3.4` then you may be tempted to use the following:

```
SecRule REMOTE_ADDR 1\.2\.3\.4
```

However, this will also match the latter part of an IP address such as `121.2.3.4`. Using the caret and dollar sign anchors solves the problem since it makes sure nothing else can come before or after the string we are matching against:

```
SecRule REMOTE_ADDR ^1\.2\.3\.4$
```

Make sure you get into the habit of using these anchors to avoid mishaps such as additional IP addresses matching.

Word Boundary

Another anchor is the *word boundary* anchor, specified by using `\b` in a regex. Like the start-of-line and end-of-line anchors, it does not match a specific character in a string, but rather a *position* in a string. In this case the position is at a word boundary—just before or after a word.

Say you wanted to match against the word `magic`, but only if it appears as a stand-alone word. You could then use the regex `\bmagic\b`, which would match the last word in the sentence `A lot like magic`, but not against the `magical` in `A magical thing`.

As with `\w` (a word character) and its inverse `\W`, which means any non-word character, the non-word boundary `\B` is available, and means any position that is not a word boundary. So the regex `A.?\Btest` would match `Atest`, `Attest`,and others, but not `A test`, since in the latter, the position before the `t` in `test` is at a word boundary.

Lazy quantifiers

By default, regex engines will try to match as much as possible when applying a regex. If you matched `The number is \d+` against the string `The number is 108`, then the entire string would match, as `\d+` would be "greedy" and try to match as much as possible (hence matching `\d+` against the entire number `108` and not just the first digit).

Sometimes you want to match as little as possible, and that is where *lazy* quantifiers come in. A lazy quantifier will cause the regex engine to only include the minimum text possible so that a match can be achieved. You make a quantifier lazy by putting a question mark after it. So for example to make the plus quantifier lazy, you write it as `+?`. The lazy version of our regex would thus be `The number is \d+?` and when matched against `The number is 108`, the resulting match would be `The number is 1`, as the lazy version of `\d+` would be satisfied with a single digit, since that achieves the requirement of the plus quantifier of "one or more".

The following table lists the lazy quantifiers that are available for use.

Quantifier	Description
`+?`	Lazy plus.
`*?`	Lazy star.
`??`	Lazy question mark.
`{min,max}?`	Lazy range.

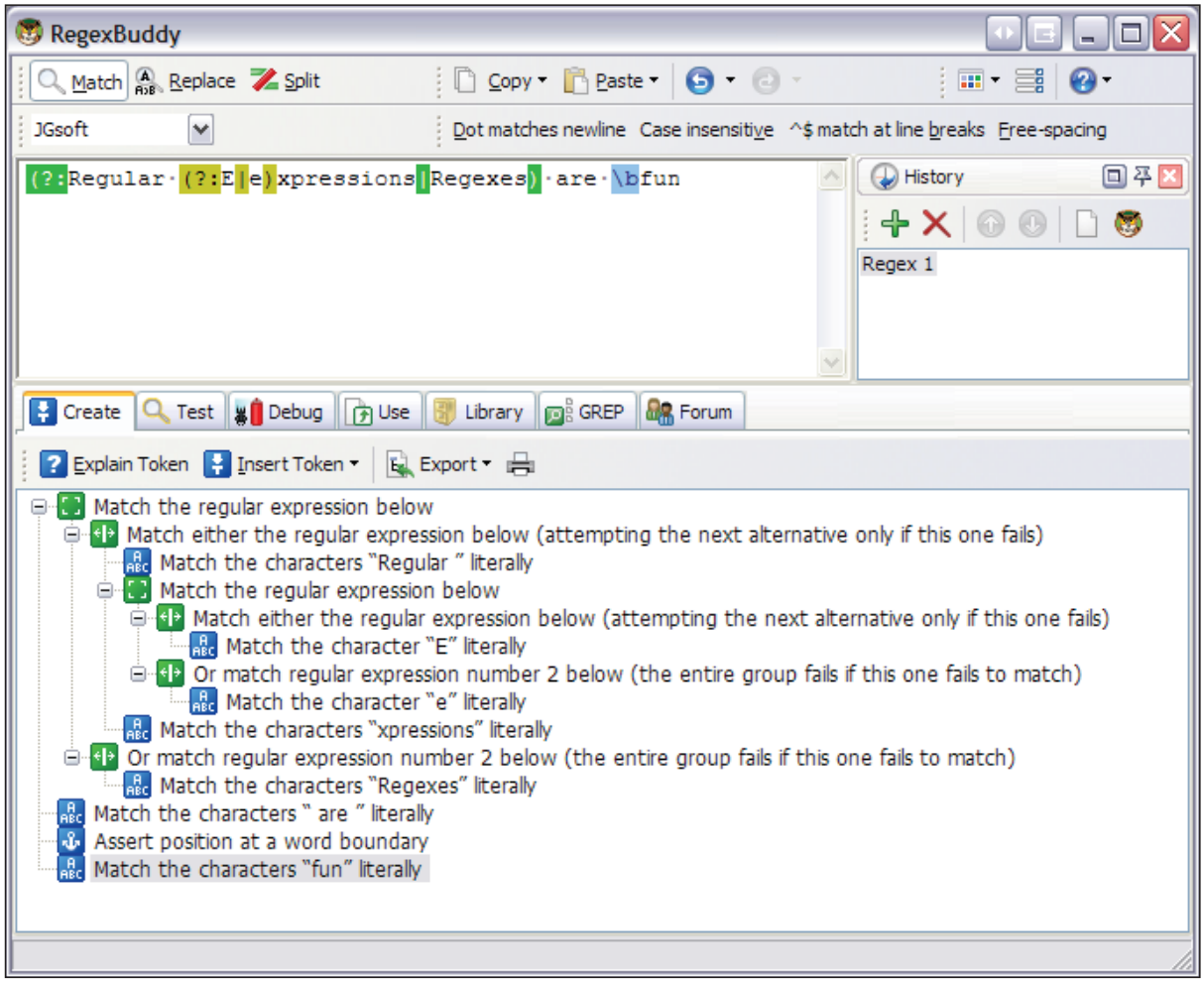
So when are lazy quantifiers needed? One example is if you're trying to extract the first HTML tag from the string `This is <b>an example</b> of using bold text`. If you use the regex `<.+>` then the resulting match will be `<b>an example</b>`, since the regex engine tries to be greedy and match as much as possible. In this case that causes it to keep trying to match after encountering the first `>` character, and when it finds the second `>`, it concludes that it has matched as much as it can and returns the match.

The solution in this case is to use the lazy version of the plus quantifier, which turns the regex into `<.+?>`. This will stop as soon as the first match is found, and so will return `<b>`, which is exactly what we wanted.

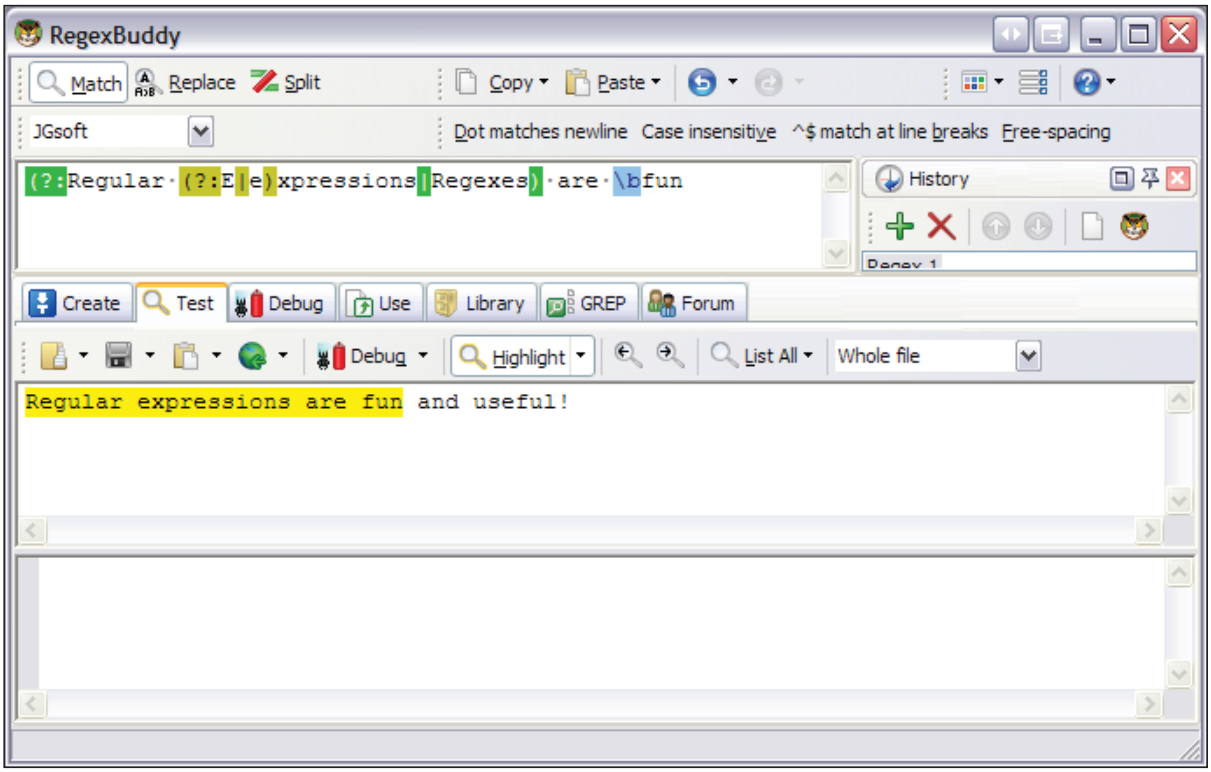
Debugging regular expressions

When a regular expression is not working as you expect, it can be handy to have a tool available that is able to tell you what a regular expression does and why something isn't matching the way it should. If you are using Windows, one such tool is RegexBuddy, available from `http://www.regexbuddy.com/`. It lets you enter a regular expression, and will explain in plain English how the regular expression works. After entering the regular expression, you can type text in an input box, and RegexBuddy will highlight the parts of the text that matches the regular expression.

The following screenshot shows RegexBuddy after the regex `(?:Regular (?:E|e)xpressions|Regexes) are \bfun` has been entered into the program. Note how the lower part of the program window explains the regex in plain English.



This next screenshot shows the "Test" tab, in which a string has been entered to see if it matches the regex created previously. You can see that the part of the string that matches has been highlighted:



If you regularly find yourself creating regexes then a tool such as RegexBuddy can save you a lot of time as you will be able to get regexes right the first time as opposed to spending needless time debugging them or not finding out until much later that they are not working as expected.

RegexBuddy is commercial software, but there are also a number of free alternatives available, such as Regex Coach (`http://weitz.de/regex-coach/`) and Expresso (`http://www.ultrapico.com/Expresso.htm`). The latter is a free download, but users are encouraged to donate some money if they find the tool useful.

Additional resources

If reading this introduction to regular expressions got you interested in learning even more, then take a look at the following resources if you want to delve deeper into the subject:

- The book Mastering Regular Expressions (O'Reilly) by Jeffrey E.F. Friedl covers the subject extremely thoroughly and is the definitive guide to the subject. At the time of this writing, the latest edition was the 3rd edition, released in 2006.
- The web site `http://www.regular-expressions.info/`, which is maintained by Jan Goyvaerts (author of RegexBuddy) contains a tutorial on regular expressions, examples, and much more.
- Jan Goyvaerts is also co-author of the book *Regular Expressions Cookbok* (O'Reilly, 2009), which is marketed as a source of practical examples of regular expressions as they are used in real life.

Our email address regex

At the beginning of the chapter I introduced a regular expression for extracting email addresses from web pages. As promised, let's use our newfound knowledge of regexes to see exactly how it works. Here, again, is the regular expression as it was presented in the beginning of the chapter:

```
\b[-\w.+]+@[\w.]+\.[a-zA-Z]{2,4}\b
```

We noted that an email address consists of a *username*, @ *character*, and *domain name*. The first part of the regex is `\b`, which makes sure that the email address starts at a word boundary. Following that, we see that the `[-\w.+]` character class allows for a word character as well as a dash, dot, or a plus sign. In this case, the dot does not need to be escaped as it is contained within a character class. Also worth noting is that the plus sign inside the character class is also interpreted as a literal plus and not as a repetition quantifier. There is another plus sign immediately following the character class, and this is an actual plus quantifier that is used to match against one or more occurrences of the characters within the character class.

Following this, the `@` character is matched literally, as it is a requirement for it to be present in an email address. After this the same character class as before, `[\w.]+` is used to allow an arbitrary number of sub-domains (for example, `misec.net` and `support.misec.net` are both allowed using this construct).

The second-to-last part of the regular expression is `\.[a-zA-Z]{2,4}`, and this corresponds to the top-level domain in the email address (such as `.com`). We see how the dot is required (and is escaped, so that it only matches the dot and not any character). Following this, a letter is required from two up to four times—this allows it to match top-level domains such as `de` and `com` and also four-letter domains such as `info`. Finally, the last part of the regex is another `\b` word-boundary assertion, to make sure the email address precedes a space or similar word-boundary marker.

Summary

This appendix showed you the basics of regular expressions—what they're used for, how to use the most common regular expression features, and the need to be aware of the differences between various regex flavors. Regular expressions are a very powerful tool, and I urge you to learn as much about them as you can—the investment in time will pay itself back many times over as you will be able to quickly solve problems using regexes that would take a lot of hard work to solve in other ways. Many interesting features of regular expressions such as look-ahead and look-behind matching, mode modifiers, and possessive quantifiers have not been covered here, but those are definitely things you'd want to read about if you get a book on regular expressions.