1

Introduction to Regular Expressions

Here's the scenario: you're given the job of checking the pages on a web server for doubled words (such as "this this"), a common problem with documents subject to heavy editing. Your job is to create a solution that will:

- Accept any number of files to check, report each line of each file that has doubled words, highlight (using standard ANSI escape sequences) each doubled word, and ensure that the source filename appears with each line in the report.
- Work across lines, even finding situations where a word at the end of one line is repeated at the beginning of the next.
- Find doubled words despite capitalization differences, such as with '`The the`…', as well as allow differing amounts of *whitespace* (spaces, tabs, newlines, and the like) to lie between the words.
- Find doubled words even when separated by HTML tags. HTML tags are for marking up text on World Wide Web pages, for example, to make a word bold: '…`it is <B>very</B> very important`…'.

That's certainly a tall order! But, it's a real problem that needs to be solved. At one point while working on the manuscript for this book, I ran such a tool on what I'd written so far and was surprised at the way numerous doubled words had crept in. There are many programming languages one could use to solve the problem, but one with regular expression support can make the job substantially easier.

Regular expressions are the key to powerful, flexible, and efficient text processing. Regular expressions themselves, with a general pattern notation almost like a mini programming language, allow you to describe and parse text. With additional support provided by the particular tool being used, regular expressions can add, remove, isolate, and generally fold, spindle, and mutilate all kinds of text and data.

It might be as simple as a text editor's search command or as powerful as a full text processing language. This book shows you the many ways regular expressions can increase your productivity. It teaches you how to *think* regular expressions so that you can master them, taking advantage of the full magnitude of their power.

A full program that solves the doubled-word problem can be implemented in just a few lines of many of today's popular languages. With a single regular-expression search-and-replace command, you can find and highlight doubled words in the document. With another, you can remove all lines without doubled words (leaving only the lines of interest left to report). Finally, with a third, you can ensure that each line to be displayed begins with the name of the file the line came from. We'll see examples in Perl and Java in the next chapter.

The host language (Perl, Java, VB.NET, or whatever) provides the peripheral processing support, but the real power comes from regular expressions. In harnessing this power for your own needs, you learn how to write regular expressions to identify text you want, while bypassing text you don't. You can then combine your expressions with the language's support constructs to actually do something with the text (add appropriate highlighting codes, remove the text, change the text, and so on).

Solving Real Problems

Knowing how to wield regular expressions unleashes processing powers you might not even know were available. Numerous times in any given day, regular expressions help me solve problems both large and small (and quite often, ones that are small but would be large if not for regular expressions).

Showing an example that provides the key to solving a large and important problem illustrates the benefit of regular expressions clearly, but perhaps not so obvious is the way regular expressions can be used throughout the day to solve rather "uninteresting" problems. I use "uninteresting" in the sense that such problems are not often the subject of bar-room war stories, but quite interesting in that until they're solved, you can't get on with your real work.

As a simple example, I needed to check a lot of files (the 70 or so files comprising the source for this book, actually) to confirm that each file contained '`SetSize`' exactly as often (or as rarely) as it contained '`ResetSize`'. To complicate matters, I needed to disregard capitalization (such that, for example, '`setSIZE`' would be counted just the same as '`SetSize`'). Inspecting the 32,000 lines of text by hand certainly wasn't practical.

Even using the normal "find this word" search in an editor would have been arduous, especially with all the files and all the possible capitalization differences.

Regular expressions to the rescue! Typing just a *single*, short command, I was able to check all files and confirm what I needed to know. Total elapsed time: perhaps 15 seconds to type the command, and another 2 seconds for the actual check of all the data. Wow! (If you're interested to see what I actually used, peek ahead to page 36.)

As another example, I was once helping a friend with some email problems on a remote machine, and he wanted me to send a listing of messages in his mailbox file. I could have loaded a copy of the whole file into a text editor and manually removed all but the few header lines from each message, leaving a sort of table of contents. Even if the file wasn't as huge as it was, and even if I wasn't connected via a slow dial-up line, the task would have been slow and monotonous. Also, I would have been placed in the uncomfortable position of actually seeing the text of his personal mail.

Regular expressions to the rescue again! I gave a simple command (using the common search tool *egrep* described later in this chapter) to display the `From:` and `Subject:` line from each message. To tell *egrep* exactly which kinds of lines I wanted to see, I used the regular expression ⌜`^(From|Subject):`⌟.

Once he got his list, he asked me to send a particular (5,000-line!) message. Again, using a text editor or the mail system itself to extract just the one message would have taken a long time. Rather, I used another tool (one called *sed*) and again used regular expressions to describe exactly the text in the file I wanted. This way, I could extract and send the desired message quickly and easily.

Saving both of us a lot of time and aggravation by using the regular expression was not "exciting," but surely much more exciting than wasting an hour in the text editor. Had I not known regular expressions, I would have never considered that there was an alternative. So, to a fair extent, this story is representative of how regular expressions and associated tools can empower you to do things you might have never thought you wanted to do.

Once you learn regular expressions, you'll realize that they're an invaluable part of your toolkit, and you'll wonder how you could ever have gotten by without them.†

A full command of regular expressions is an invaluable skill. This book provides the information needed to acquire that skill, and it is my hope that it provides the motivation to do so, as well.

† If you have a TiVo, you already know the feeling!

Regular Expressions as a Language

Unless you've had some experience with regular expressions, you won't understand the regular expression ⌜`^(From|Subject):`⌟ from the last example, but there's nothing magic about it. For that matter, there is nothing magic about magic. The magician merely understands something simple which doesn't *appear* to be simple or natural to the untrained audience. Once you learn how to hold a card while making your hand look empty, you only need practice before you, too, can "do magic." Like a foreign language — once you learn it, it stops sounding like gibberish.

The Filename Analogy

Since you have decided to use this book, you probably have at least some idea of just what a "regular expression" is. Even if you don't, you are almost certainly already familiar with the basic concept.

You know that *report.txt* is a specific filename, but if you have had any experience with Unix or DOS/Windows, you also know that the pattern "`*.txt`" can be used to select multiple files. With filename patterns like this (called *file globs* or *wildcards*), a few characters have special meaning. The star means "match anything," and a question mark means "match any one character." So, with the file glob "`*.txt`", we start with a match-anything ⌜`*`⌟ and end with the literal ⌜`.txt`⌟, so we end up with a pattern that means "select the files whose names start with anything and end with `.txt`".

Most systems provide a few additional special characters, but, in general, these filename patterns are limited in expressive power. This is not much of a shortcoming because the scope of the problem (to provide convenient ways to specify groups of files) is limited, well, simply to filenames.

On the other hand, dealing with general text is a much larger problem. Prose and poetry, program listings, reports, HTML, code tables, word lists... you name it, if a particular need is specific enough, such as "selecting files," you can develop some kind of specialized scheme or tool to help you accomplish it. However, over the years, a *generalized pattern language* has developed, which is powerful and expressive for a wide variety of uses. Each program implements and uses them differently, but in general, **this powerful pattern language and the patterns themselves are called *regular expressions*.**

The Language Analogy

Full regular expressions are composed of two types of characters. The special characters (like the `*` from the filename analogy) are called *metacharacters*, while the rest are called *literal*, or normal text characters. What sets regular expressions apart from filename patterns are the advanced expressive powers that their metacharacters provide. Filename patterns provide limited metacharacters for limited needs, but a regular expression "language" provides rich and expressive metacharacters for advanced uses.

It might help to consider regular expressions as their own language, with literal text acting as the words and metacharacters as the grammar. The words are combined with grammar according to a set of rules to create an expression that communicates an idea. In the email example, the expression I used to find lines beginning with '`From:`' or '`Subject:`' was `^(From|Subject):`. The metacharacters are underlined; we'll get to their interpretation soon.

As with learning any other language, regular expressions might seem intimidating at first. This is why it seems like magic to those with only a superficial understanding, and perhaps completely unapproachable to those who have never seen it at all. But, just as 正規表現は簡単だよ![†] would soon become clear to a student of Japanese, the regular expression in

```
s!<emphasis>([0-9]+(\.[0-9]+){3})</emphasis>!<inet>$1</inet>!
```

will soon become crystal clear to you, too.

This example is from a Perl language script that my editor used to modify a manuscript. The author had mistakenly used the typesetting tag `<emphasis>` to mark Internet IP addresses (which are sets of periods and numbers that look like `209.204.146.22`). The incantation uses Perl's text-substitution command with the regular expression

```
<emphasis>([0-9]+(\.[0-9]+){3})</emphasis>
```

to replace such tags with the appropriate `<inet>` tag, while leaving other uses of `<emphasis>` alone. In later chapters, you'll learn all the details of exactly how this type of incantation is constructed, so you'll be able to apply the techniques to your own needs, with your own application or programming language.

---

† "Regular expressions are easy!" A somewhat humorous comment about this: as Chapter 3 explains, the term *regular expression* originally comes from formal algebra. When people ask me what my book is about, the answer "regular expressions" draws a blank face if they are not already familiar with the concept. The Japanese word for regular expression, 正規表現, means as little to the average Japanese as its English counterpart, but my reply in Japanese usually draws a bit more than a blank stare. You see, the "regular" part is unfortunately pronounced identically to a much more common word, a medical term for "reproductive organs." You can only imagine what flashes through their minds until I explain!

The goal of this book

The chance that *you* will ever want to replace `<emphasis>` tags with `<inet>` tags is small, but it is very likely that you will run into similar "replace *this* with *that*" problems. The goal of this book is not to teach solutions to specific problems, but rather to teach you how to *think* regular expressions so that you will be able to conquer whatever problem you may face.

The Regular-Expression Frame of Mind

As we'll soon see, complete regular expressions are built up from small building-block units. Each individual building block is quite simple, but since they can be combined in an infinite number of ways, knowing how to combine them to achieve a particular goal takes some experience. So, this chapter provides a quick overview of some regular-expression concepts. It doesn't go into much depth, but provides a basis for the rest of this book to build on, and sets the stage for important side issues that are best discussed before we delve too deeply into the regular expressions themselves.

While some examples may seem silly (because some *are* silly), they represent the kind of tasks that you will want to do — you just might not realize it yet. If each point doesn't seem to make sense, don't worry too much. Just let the gist of the lessons sink in. That's the goal of this chapter.

If You Have Some Regular-Expression Experience

If you're already familiar with regular expressions, much of this overview will not be new, but please be sure to at least glance over it anyway. Although you may be aware of the basic meaning of certain metacharacters, perhaps some of the ways of thinking about and looking at regular expressions will be new.

Just as there is a difference between playing a musical piece well and *making music*, there is a difference between knowing about regular expressions and *really understanding* them. Some of the lessons present the same information that you are already familiar with, but in ways that may be new and which are the first steps to *really understanding*.

Searching Text Files: Egrep

Finding text is one of the simplest uses of regular expressions — many text editors and word processors allow you to search a document using a regular-expression pattern. Even simpler is the utility *egrep*. Give *egrep* a regular expression and some files to search, and it attempts to match the regular expression to each line of each file, displaying only those lines in which a match is found. *egrep* is freely available

for many systems, including DOS, MacOS, Windows, Unix, and so on. See this book's web site, `http://regex.info`, for links on how to obtain a copy of *egrep* for your system.

Returning to the email example from page 3, the command I actually used to generate a makeshift table of contents from the email file is shown in Figure 1-1. *egrep* interprets the first command-line argument as a regular expression, and any remaining arguments as the file(s) to search. Note, however, that the single quotes shown in Figure 1-1 are *not* part of the regular expression, but are needed by my command shell.† When using *egrep*, I usually wrap the regular expression with single quotes. Exactly which characters are special, in what contexts, to whom (to the regular-expression, or to the tool), and in what order they are interpreted are all issues that grow in importance when you move to regular-expression use in full-fledged programming languages—something we'll see starting in the next chapter.

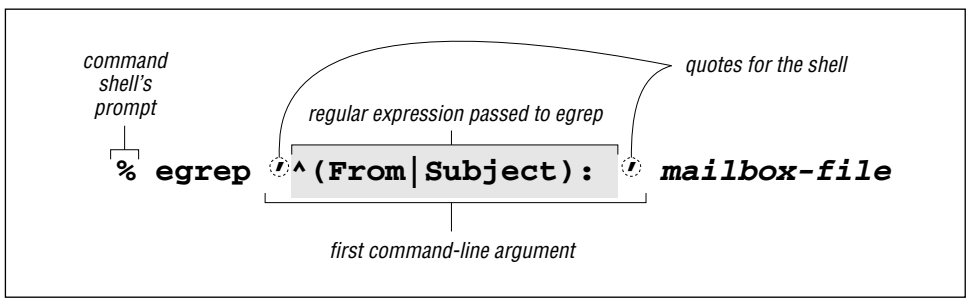


Figure 1-1: Invoking egrep from the command line

We'll start to analyze just what the various parts of the regex mean in a moment, but you can probably already guess just by looking that some of the characters have special meanings. In this case, the parentheses, the ⌜^⌟, and the ⌜|⌟ characters are regular-expression metacharacters, and combine with the other characters to generate the result I want.

On the other hand, if your regular expression doesn't use any of the dozen or so metacharacters that *egrep* understands, it effectively becomes a simple "plain text" search. For example, searching for ⌜`cat`⌟ in a file finds and displays all lines with the three letters `c·a·t` in a row. This includes, for example, any line containing `vacation`.

† The command shell is the part of the system that accepts your typed commands and actually executes the programs you request. With the shell I use, the single quotes serve to group the command argument, telling the shell not to pay too much attention to what's inside. If I didn't use them, the shell might think, for example, a '*' that I intended to be part of the regular expression was really part of a filename pattern that *it* should interpret. I don't want that to happen, so I use the quotes to "hide" the metacharacters from the shell. Windows users of COMMAND.COM or CMD.EXE should probably use double quotes instead.

Even though the line might not have the *word* `cat`, the `c·a·t` sequence in `vacation` is still enough to be matched. Since it's there, *egrep* goes ahead and displays the whole line. The key point is that regular-expression searching is not done on a "word" basis—*egrep* can understand the concept of bytes and lines in a file, but it generally has no idea of English's (or any other language's) words, sentences, paragraphs, or other high-level concepts.

Egrep Metacharacters

Let's start to explore some of the *egrep* metacharacters that supply its regular-expression power. I'll go over them quickly with a few examples, leaving the detailed examples and descriptions for later chapters.

Typographical Conventions Before we begin, please make sure to review the typographical conventions explained in the preface, on page *xix*. This book forges a bit of new ground in the area of typesetting, so some of my notations may be unfamiliar at first.

Start and End of the Line

Probably the easiest metacharacters to understand are ⌈`^`⌋ (*caret*) and ⌈`$`⌋ (*dollar*), which represent the start and end, respectively, of the line of text as it is being checked. As we've seen, the regular expression ⌈`cat`⌋ finds `c·a·t` anywhere on the line, but ⌈`^cat`⌋ matches only if the `c·a·t` is at the beginning of the line—the ⌈`^`⌋ is used to effectively *anchor* the match (of the rest of the regular expression) to the start of the line. Similarly, ⌈`cat$`⌋ finds `c·a·t` only at the end of the line, such as a line ending with `scat`.

It's best to get into the habit of interpreting regular expressions in a rather literal way. For example, don't think

> ⌈`^cat`⌋ matches a line with `cat` at the beginning

but rather:

> ⌈`^cat`⌋ matches if you have the beginning of a line, followed immediately by `c`, followed immediately by `a`, followed immediately by `t`.

They both end up meaning the same thing, but reading it the more literal way allows you to intrinsically understand a new expression when you see it. How would *egrep* interpret ⌈`^cat$`⌋, ⌈`^$`⌋, or even simply ⌈`^`⌋ alone? ❖ Turn the page to check your interpretations.

The caret and dollar are special in that they match a *position* in the line rather than any actual text characters themselves. Of course, there are various ways to actually match real text. Besides providing literal characters like ⌈`cat`⌋ in your regular expression, you can also use some of the items discussed in the next few sections.

Character Classes

Matching any one of several characters

Let's say you want to search for "grey," but also want to find it if it were spelled "gray." The regular-expression construct ⌈`[···]`⌋, usually called a *character class*, lets you list the characters you want to allow at that point in the match. While ⌈`e`⌋ matches just an `e`, and ⌈`a`⌋ matches just an `a`, the regular expression ⌈`[ea]`⌋ matches either. So, then, consider ⌈`gr[ea]y`⌋: this means to find "`g`, followed by `r`, followed by either an `e` or an `a`, all followed by `y`." Because I'm a really poor speller, I'm always using regular expressions like this against a huge list of English words to figure out proper spellings. One I use often is ⌈`sep[ea]r[ea]te`⌋, because I can never remember whether the word is spelled "seperate," "separate," "separete," or what. The one that pops up in the list is the proper spelling; regular expressions to the rescue.

Notice how outside of a class, literal characters (like the ⌈`g`⌋ and ⌈`r`⌋ of ⌈`gr[ae]y`⌋) have an implied "and then" between them — "match ⌈`g`⌋ **and then** match ⌈`r`⌋..." It's completely opposite inside a character class. The contents of a class is a list of characters that can match at that point, so the implication is "or."

As another example, maybe you want to allow capitalization of a word's first letter, such as with ⌈`[Ss]mith`⌋. Remember that this still matches lines that contain `smith` (or `Smith`) embedded within another word, such as with `blacksmith`. I don't want to harp on this throughout the overview, but this issue does seem to be the source of problems among some new users. I'll touch on some ways to handle this embedded-word problem after we examine a few more metacharacters.

You can list in the class as many characters as you like. For example, ⌈`[123456]`⌋ matches any of the listed digits. This particular class might be useful as part of ⌈`<H[123456]>`⌋, which matches `<H1>`, `<H2>`, `<H3>`, etc. This can be useful when searching for HTML headers.

Within a character class, the *character-class metacharacter* '`-`' (*dash*) indicates a range of characters: ⌈`<H[1-6]>`⌋ is identical to the previous example. ⌈`[0-9]`⌋ and ⌈`[a-z]`⌋ are common shorthands for classes to match digits and English lowercase letters, respectively. Multiple ranges are fine, so ⌈`[0123456789abcdefABCDEF]`⌋ can be written as ⌈`[0-9a-fA-F]`⌋ (or, perhaps, ⌈`[A-Fa-f0-9]`⌋, since the order in which ranges are given doesn't matter). These last three examples can be useful when processing hexadecimal numbers. You can freely combine ranges with literal characters: ⌈`[0-9A-Z_!.?]`⌋ matches a digit, uppercase letter, underscore, exclamation point, period, or a question mark.

Note that a dash is a metacharacter only within a character class — otherwise it matches the normal dash character. In fact, it is not even always a metacharacter within a character class. If it is the first character listed in the class, it can't possibly

Reading ⌜^cat$⌟, ⌜^$⌟, and ⌜^⌟

❖ *Answers to the questions on page 8.*

⌜^cat$⌟ **Literally means**: matches if the line has a beginning-of-line (which, of course, all lines have), followed immediately by `c·a·t`, and then followed immediately by the end of the line.

Effectively means: a line that consists of only `cat` — no extra words, spaces, punctuation... just '`cat`'.

⌜^$⌟ **Literally means**: matches if the line has a beginning-of-line, followed immediately by the end of the line.

Effectively means: an empty line (with nothing in it, not even spaces).

⌜^⌟ **Literally means**: matches if the line has a beginning-of-line.

Effectively *meaningless*! Since every line has a beginning, every line will match—even lines that are empty!

indicate a range, so it is not considered a metacharacter. Along the same lines, the question mark and period at the end of the class are usually regular-expression metacharacters, but only when *not* within a class (so, to be clear, the only special characters within the class in ⌜`[0-9A-Z_!.?]`⌟ are the two dashes).

> Consider character classes as their own mini language. The rules regarding which metacharacters are supported (and what they do) are completely different inside and outside of character classes.

We'll see more examples of this shortly.

Negated character classes

If you use ⌜`[^···]`⌟ instead of ⌜`[···]`⌟, the class matches any character that *isn't* listed. For example, ⌜`[^1-6]`⌟ matches a character that's *not* `1` through `6`. The leading `^` in the class "negates" the list, so rather than listing the characters you want to include in the class, you list the characters you don't want to be included.

You might have noticed that the `^` used here is the same as the start-of-line caret introduced on page 8. The character is the same, but the meaning is completely different. Just as the English word "wind" can mean different things depending on the context (sometimes a strong breeze, sometimes what you do to a clock), so can a metacharacter. We've already seen one example, the range-building dash. It is valid only inside a character class (and at that, only when not first inside the class). `^` is a line anchor outside a class, but a class metacharacter inside a class (but, only when it is immediately after the class's opening bracket; otherwise, it's

not special inside a class). Don't fear—these are the most complex special cases; others we'll see later aren't so bad.

As another example, let's search that list of English words for odd words that have `q` followed by something other than `u`. Translating that into a regular expression, it becomes ⌜`q[^u]`⌟. I tried it on the list I have, and there certainly weren't many. I did find a few, including a number of words that I didn't even know were English.

Here's what happened. (What I typed is in bold.)

```
% egrep 'q[^u]' word.list
Iraqi
Iraqian
miqra
qasida
qintar
qoph
zaqqum%
```

Two notable words not listed are "Qantas", the Australian airline, and "Iraq". Although both words are in the *word.list* file, neither were displayed by my *egrep* command. Why? ❖ Think about it for a bit, and then turn the page to check your reasoning.

Remember, a negated character class means "match a character that's not listed" and not "don't match what is listed." These might seem the same, but the `Iraq` example shows the subtle difference. A convenient way to view a negated class is that it is simply a shorthand for a normal class that includes all possible characters *except* those that are listed.

Matching Any Character with Dot

The metacharacter ⌜`.`⌟ (usually called *dot* or *point*) is a shorthand for a character class that matches any character. It can be convenient when you want to have an "any character here" placeholder in your expression. For example, if you want to search for a date such as `03/19/76`, `03-19-76`, or even `03.19.76`, you could go to the trouble to construct a regular expression that uses character classes to explicitly allow '`/`', '`-`', or '`.`' between each number, such as ⌜`03[-./]19[-./]76`⌟. However, you might also try simply using ⌜`03.19.76`⌟.

Quite a few things are going on with this example that might be unclear at first. In ⌜`03[-./]19[-./]76`⌟, the dots are *not* metacharacters because they are within a character class. (Remember, the list of metacharacters and their meanings are different inside and outside of character classes.) The dashes are also not class metacharacters *in this case* because each is the first thing after `[` or `[^`. Had they not been first, as with ⌜`[.-/]`⌟, they would be the class range metacharacter, which would be a mistake in this situation.

Quiz Answer

❖ *Answer to the question on page 11.*

Why doesn't ⌜`q[^u]`⌟ match 'Qantas' or 'Iraq'?

`Qantas` didn't match because the regular expression called for a lowercase `q`, whereas the `Q` in Qantas is uppercase. Had we used ⌜`Q[^u]`⌟ instead, we would have found it, but not the others, since they don't have an uppercase `Q`. The expression ⌜`[Qq][^u]`⌟ would have found them all.

The `Iraq` example is somewhat of a trick question. The regular expression calls for `q` *followed by a character* that's not `u`, which precludes matching `q` *at the end of the line*. Lines generally have newline characters at the very end, but a little fact I neglected to mention (sorry!) is that *egrep* strips those before checking with the regular expression, so after a line-ending `q`, there's no non-`u` to be matched.

Don't feel too bad because of the trick question.† Let me assure you that had *egrep* not automatically stripped the newlines (many other tools don't strip them), or had `Iraq` been followed by spaces or other words or whatnot, the line would have matched. It is important to eventually understand the little details of each tool, but at this point what I'd like you to come away with from this exercise is that *a character class, even negated, still requires a character to match*.

With ⌜`03.19.76`⌟, the dots *are* metacharacters — ones that match any character (including the dash, period, and slash that we are expecting). However, it is important to know that each dot can match any character at all, so it can match, say, '`lottery numbers: 19 203319 7639`'.

So, ⌜`03[-./]19[-./]76`⌟ is more precise, but it's more difficult to read and write. ⌜`03.19.76`⌟ is easy to understand, but vague. Which should we use? It all depends upon what you know about the data being searched, and just how specific you feel you need to be. One important, recurring issue has to do with balancing your knowledge of the text being searched against the need to always be exact when writing an expression. For example, if you know that with your data it would be highly unlikely for ⌜`03.19.76`⌟ to match in an unwanted place, it would certainly be reasonable to use it. Knowing the target text well is an important part of wielding regular expressions effectively.

† Once, in fourth grade, I was leading the spelling bee when I was asked to spell "miss." My answer was "`m·i·s·s`." Miss Smith relished in telling me that no, it was "`M·i·s·s`" with a capital `M`, that I should have asked for an example sentence, and that I was out. It was a traumatic moment in a young boy's life. After that, I never liked Miss Smith, and have since been a very poor speler.

Alternation

Matching any one of several subexpressions

A very convenient metacharacter is ⌜`|`⌟, which means "or." It allows you to combine multiple expressions into a single expression that matches any of the individual ones. For example, ⌜`Bob`⌟ and ⌜`Robert`⌟ are separate expressions, but ⌜`Bob|Robert`⌟ is one expression that matches either. When combined this way, the subexpressions are called *alternatives*.

Looking back to our ⌜`gr[ea]y`⌟ example, it is interesting to realize that it can be written as ⌜`grey|gray`⌟, and even ⌜`gr(a|e)y`⌟. The latter case uses parentheses to constrain the alternation. (For the record, parentheses are metacharacters too.) Note that something like ⌜`gr[a|e]y`⌟ is *not* what we want — within a class, the '`|`' character is just a normal character, like ⌜`a`⌟ and ⌜`e`⌟.

With ⌜`gr(a|e)y`⌟, the parentheses are required because without them, ⌜`gra|ey`⌟ means "⌜`gra`⌟ or ⌜`ey`⌟," which is not what we want here. Alternation reaches far, but not beyond parentheses. Another example is ⌜`(First|1st)␣[Ss]treet`⌟.† Actually, since both ⌜`First`⌟ and ⌜`1st`⌟ end with ⌜`st`⌟, the combination can be shortened to ⌜`(Fir|1)st␣[Ss]treet`⌟. That's not necessarily quite as easy to read, but be sure to understand that ⌜`(first|1st)`⌟ and ⌜`(fir|1)st`⌟ effectively mean the same thing.

Here's an example involving an alternate spelling of my name. Compare and contrast the following three expressions, which are all effectively the same:

⌜`Jeffrey|Jeffery`⌟
⌜`Jeff(rey|ery)`⌟
⌜`Jeff(re|er)y`⌟

To have them match the British spellings as well, they could be:

⌜`(Geoff|Jeff)(rey|ery)`⌟
⌜`(Geo|Je)ff(rey|ery)`⌟
⌜`(Geo|Je)ff(re|er)y`⌟

Finally, note that these three match effectively the same as the longer (but simpler) ⌜`Jeffrey|Geoffery|Jeffery|Geoffrey`⌟. They're all different ways to specify the same desired matches.

Although the ⌜`gr[ea]y`⌟ versus ⌜`gr(a|e)y`⌟ examples might blur the distinction, be careful not to confuse the concept of alternation with that of a character class. A character class can match just a *single character* in the target text. With alternation, since each alternative can be a full-fledged regular expression in and of itself, each

† Recall from the typographical conventions on page *xx* that "␣" is how I sometimes show a space character so it can be seen easily.

alternative can match an arbitrary amount of text. Character classes are almost like their own special mini-language (with their own ideas about metacharacters, for example), while alternation is part of the "main" regular expression language. You'll find both to be extremely useful.

Also, take care when using caret or dollar in an expression that has alternation. Compare ⌈`^From|Subject|Date:␣`⌋ with ⌈`^(From|Subject|Date):␣`⌋. Both appear similar to our earlier email example, but what each matches (and therefore how useful it is) differs greatly. The first is composed of three alternatives, so it matches "⌈`^From`⌋ or ⌈`Subject`⌋ or ⌈`Date:␣`⌋," which is not particularly useful. We want the leading caret and trailing ⌈`:␣`⌋ to apply to each alternative. We can accomplish this by using parentheses to "constrain" the alternation:

⌈`^(From|Subject|Date):␣`⌋

The alternation is constrained by the parentheses, so literally, this regex means "match the start of the line, then one of ⌈`From`⌋, ⌈`Subject`⌋, or ⌈`Date`⌋, and then match ⌈`:␣`⌋." Effectively, it matches:

1) start-of-line, followed by `F·r·o·m`, followed by '`:␣`'
or 2) start-of-line, followed by `S·u·b·j·e·c·t`, followed by '`:␣`'
or 3) start-of-line, followed by `D·a·t·e`, followed by '`:␣`'

Putting it less literally, it matches lines beginning with '`From:␣`', '`Subject:␣`', or '`Date:␣`', which is quite useful for listing the messages in an email file.

Here's an example:

```
% egrep  '^(From|Subject|Date): '  mailbox
From: elvis@tabloid.org (The King)
Subject: be seein' ya around
Date: Thu, 22 Aug 2002 11:04:13
From: The Prez <president@whitehouse.org>
Date: Tue, 27 Aug 2002 8:36:24
Subject: now, about your vote
   ⋮
```

Ignoring Differences in Capitalization

This email header example provides a good opportunity to introduce the concept of a *case-insensitive* match. The field types in an email header usually appear with leading capitalization, such as "Subject" and "From," but the email standard actually allows mixed capitalization, so things like "DATE" and "from" are also allowed. Unfortunately, the regular expression in the previous section doesn't match those.

One approach is to replace ⌈`From`⌋ with ⌈`[Ff][Rr][Oo][Mm]`⌋ to match any form of "from," but this is quite cumbersome, to say the least. Fortunately, there is a way to tell *egrep* to ignore case when doing comparisons, i.e., to perform the match in a *case insensitive* manner in which capitalization differences are simply ignored. It is

not a part of the regular-expression language, but is a related useful feature many tools provide. *egrep*'s command-line option "`-i`" tells it to do a case-insensitive match. Place `-i` on the command line before the regular expression:

```
% egrep -i  '^(From|Subject|Date): '  mailbox
```

This brings up all the lines we matched before, but also includes lines such as:

```
SUBJECT: MAKE MONEY FAST
```

I find myself using the `-i` option quite frequently (perhaps related to the footnote on page 12!) so I recommend keeping it in mind. We'll see other convenient support features like this in later chapters.

Word Boundaries

A common problem is that a regular expression that matches the word you want can often also match where the "word" is embedded within a larger word. I mentioned this briefly in the `cat`, `gray`, and `Smith` examples. It turns out, though, that some versions of *egrep* offer limited support for word recognition: namely the ability to match the boundary of a word (where a word begins or ends).

You can use the (perhaps odd looking) *metasequences* ⌜`\<`⌟ and ⌜`\>`⌟ if your version happens to support them (not all versions of *egrep* do). You can think of them as word-based versions of ⌜`^`⌟ and ⌜`$`⌟ that match the *position* at the start and end of a word, respectively. Like the line anchors caret and dollar, they anchor other parts of the regular expression but don't actually consume any characters during a match. The expression ⌜`\<cat\>`⌟ literally means "match if we can find a start-of-word position, followed immediately by `c·a·t`, followed immediately by an end-of-word position." More naturally, it means "find the word `cat`." If you wanted, you could use ⌜`\<cat`⌟ or ⌜`cat\>`⌟ to find words starting and ending with `cat`.

Note that ⌜`<`⌟ and ⌜`>`⌟ alone are not metacharacters — when combined with a backslash, the *sequences* become special. This is why I called them "metasequences." It's their special interpretation that's important, not the number of characters, so for the most part I use these two meta-words interchangeably.

Remember, not all versions of *egrep* support these word-boundary metacharacters, and those that do don't magically understand the English language. The "start of a word" is simply the position where a sequence of alphanumeric characters begins; "end of word" is where such a sequence ends. Figure 1-2 on the next page shows a sample line with these positions marked.

The word-starts (as *egrep* recognizes them) are marked with up arrows, the word-ends with down arrows. As you can see, "start and end of word" is better phrased as "start and end of an alphanumeric sequence," but perhaps that's too much of a mouthful.

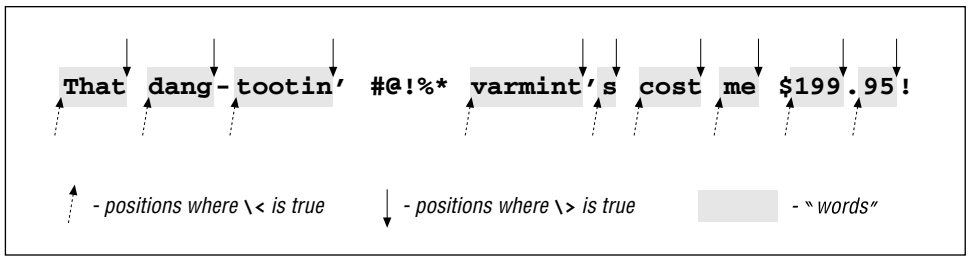


Figure 1-2: Start and end of "word" positions

In a Nutshell

Table 1-1 summarizes the metacharacters we have seen so far.

Table 1-1: Summary of Metacharacters Seen So Far

Metacharacter	Name	Matches

`.`
`[…]`
`[^…]`
	dot
	character class
	negated character class
		any one character
		any character listed
		any character not listed

`^`
`$`
`\<`
`\>`
	caret
	dollar
	backslash less-than
	backslash greater-than
		the position at the start of the line
		the position at the end of the line
		†the position at the start of a word
		†the position at the end of a word
		†*not supported by all versions of egrep*

`|` `(…)`	*or, bar* *parentheses*	matches either expression it separates used to limit scope of `|`, plus additional uses yet to be discussed

In addition to the table, important points to remember include:

- The rules about which characters are and aren't metacharacters (and exactly what they mean) are different inside a character class. For example, dot is a metacharacter outside of a class, but not within one. Conversely, a dash is a metacharacter within a class (usually), but not outside. Moreover, a caret has one meaning outside, another if specified inside a class immediately after the opening `[`, and a third if given elsewhere in the class.
- Don't confuse alternation with a character class. The class `[abc]` and the alternation `(a|b|c)` effectively mean the same thing, but the similarity in this example does not extend to the general case. A character class can match exactly one character, and that's true no matter how long or short the specified list of acceptable characters might be.

Alternation, on the other hand, can have arbitrarily long alternatives, each textually unrelated to the other: ⌈`\<(1,000,000|million|thousand•thou)\>`⌋. However, alternation can't be negated like a character class.

- A negated character class is simply a notational convenience for a normal character class that matches everything not listed. Thus, ⌈`[^x]`⌋ doesn't mean "match unless there is an `x`," but rather "match if there is something that is not `x`." The difference is subtle, but important. The first concept matches a blank line, for example, while ⌈`[^x]`⌋ does not.
- The useful `-i` option discounts capitalization during a match (☞ 15).†

What we have seen so far can be quite useful, but the real power comes from *optional* and *counting* elements, which we'll look at next.

Optional Items

Let's look at matching `color` or `colour`. Since they are the same except that one has a `u` and the other doesn't, we can use ⌈`colo`**`u?`**`r`⌋ to match either. The metacharacter ⌈`?`⌋ (*question mark*) means *optional*. It is placed after the character that is allowed to appear at that point in the expression, but whose existence isn't actually required to still be considered a successful match.

Unlike other metacharacters we have seen so far, the question mark attaches only to the immediately-preceding item. Thus, ⌈`colou?r`⌋ is interpreted as "⌈`c`⌋ then ⌈`o`⌋ then ⌈`l`⌋ then ⌈`o`⌋ then ⌈`u?`⌋ then ⌈`r`⌋."

The ⌈`u?`⌋ part is always successful: sometimes it matches a `u` in the text, while other times it doesn't. The whole point of the `?`-optional part is that it's successful either way. This isn't to say that any regular expression that contains `?` is always successful. For example, against '`semicolon`', both ⌈`colo`⌋ and ⌈`u?`⌋ are successful (matching `colo` and nothing, respectively). However, the final ⌈`r`⌋ fails, and that's what disallows `semicolon`, in the end, from being matched by ⌈`colou?r`⌋.

As another example, consider matching a date that represents July fourth, with the "July" part being either `July` or `Jul`, and the "fourth" part being `fourth`, `4th`, or simply `4`. Of course, we could just use ⌈`(July|Jul)•(fourth|4th|4)`⌋, but let's explore other ways to express the same thing.

First, we can shorten the ⌈`(July|Jul)`⌋ to ⌈`(Jul`**`y?`**`)`⌋. Do you see how they are effectively the same? The removal of the ⌈`|`⌋ means that the parentheses are no longer really needed. Leaving the parentheses doesn't hurt, but with them removed, ⌈`July?`⌋ is a bit less cluttered. This leaves us with ⌈`July?•(fourth|4th|4)`⌋.

† Recall from the typographical conventions (page *xx*) that something like "☞ 15" is a shorthand for a reference to another page of this book.

Moving now to the second half, we can simplify the ⌈4th|4⌋ to ⌈4(th)?⌋. As you can see, ⌈?⌋ can attach to a parenthesized expression. Inside the parentheses can be as complex a subexpression as you like, but "from the outside" it is considered a single unit. Grouping for ⌈?⌋ (and other similar metacharacters which I'll introduce momentarily) is one of the main uses of parentheses.

Our expression now looks like ⌈July?␣(fourth|4(th)?)⌋. Although there are a fair number of metacharacters, and even nested parentheses, it is not that difficult to decipher and understand. This discussion of two essentially simple examples has been rather long, but in the meantime we have covered tangential topics that add a lot, if perhaps only subconsciously, to our understanding of regular expressions. Also, it's given us some experience in taking different approaches toward the same goal. As we advance through this book (and through to a better understanding), you'll find many opportunities for creative juices to flow while trying to find the optimal way to solve a complex problem. Far from being some stuffy science, writing regular expressions is closer to an art.

Other Quantifiers: Repetition

Similar to the question mark are ⌈+⌋ (*plus*) and ⌈*⌋ (an asterisk, but as a regular-expression metacharacter, I prefer the term *star*). The metacharacter ⌈+⌋ means "one or more of the immediately-preceding item," and ⌈*⌋ means "any number, including none, of the item." Phrased differently, ⌈···*⌋ means "try to match it as many times as possible, but it's okay to settle for nothing if need be." The construct with plus, ⌈···+⌋, is similar in that it also tries to match as many times as possible, but different in that it fails if it can't match at least once. These three metacharacters, question mark, plus, and star, are called *quantifiers* because they influence the quantity of what they govern.

Like ⌈···?⌋, the ⌈···*⌋ part of a regular expression always succeeds, with the only issue being what text (if any) is matched. Contrast this to ⌈···+⌋, which fails unless the item matches at least once.

For example, ⌈␣?⌋ allows a single optional space, but ⌈␣*⌋ allows *any number* of optional spaces. We can use this to make page 9's <H[1-6]> example flexible. The HTML specification† says that spaces are allowed immediately before the closing >, such as with <H3␣> and <H4␣␣␣>. Inserting ⌈␣*⌋ into our regular expression where we want to allow (but not require) spaces, we get ⌈<H[1-6]␣*>⌋. This still matches <H1>, as no spaces are required, but it also flexibly picks up the other versions.

† If you are not familiar with HTML, never fear. I use these as real-world examples, but I provide all the details needed to understand the points being made. Those familiar with parsing HTML tags will likely recognize important considerations I don't address at this point in the book.

Exploring further, let's search for an HTML tag such as `<HR␣SIZE=14>`, which indicates that a line (a Horizontal Rule) 14 pixels thick should be drawn across the screen. Like the `<H3>` example, optional spaces are allowed before the closing angle bracket. Additionally, they are allowed on either side of the equal sign. Finally, one space is required between the `HR` and `SIZE`, although more are allowed. To allow more, we could just add ⌈␣*⌋ to the ⌈␣⌋ already there, but instead let's change it to ⌈␣+⌋. The plus allows extra spaces while still requiring at least one, so it's effectively the same as ⌈␣␣*⌋, but more concise. All these changes leave us with ⌈<HR␣+SIZE␣*=␣*14␣*>⌋.

Although flexible with respect to spaces, our expression is still inflexible with respect to the size given in the tag. Rather than find tags with only one particular size such as `14`, we want to find them all. To accomplish this, we replace the ⌈14⌋ with an expression to find a general number. Well, in this case, a "number" is one or more digits. A digit is ⌈[0-9]⌋, and "one or more" adds a plus, so we end up replacing ⌈14⌋ by ⌈[0-9]+⌋. (A character class is one "unit," so can be subject directly to plus, question mark, and so on, without the need for parentheses.)

This leaves us with ⌈<HR␣+SIZE␣*=␣*[0-9]+␣*>⌋, which is certainly a mouthful even though I've presented it with the metacharacters bold, added a bit of spacing to make the groupings more apparent, and am using the "visible space" symbol '␣' for clarity. (Luckily, *egrep* has the `-i` case-insensitive option, ☞15, which means I don't have to use ⌈[Hh][Rr]⌋ instead of ⌈HR⌋.) The unadorned regular expression ⌈<HR +SIZE *= *[0-9]+ *>⌋ likely appears even more confusing. This example looks particularly odd because the subjects of most of the stars and pluses are space characters, and our eye has always been trained to treat spaces specially. That's a habit you will have to break when reading regular expressions, because the space character is a normal character, no different from, say, `j` or `4`. (In later chapters, we'll see that some other tools support a special mode in which white-space is ignored, but *egrep* has no such mode.)

Continuing to exploit a good example, let's consider that the size attribute is optional, so you can simply use `<HR>` if the default size is wanted. (Extra spaces are allowed before the `>`, as always.) How can we modify our regular expression so that it matches either type? The key is realizing that the size part is *optional* (that's a hint). ❖ Turn the page to check your answer.

Take a good look at our latest expression (in the answer box) to appreciate the differences among the question mark, star, and plus, and what they really mean in practice. Table 1-2 on the next page summarizes their meanings.

Note that each quantifier has some minimum number of matches required to succeed, and a maximum number of matches that it will ever attempt. With some, the minimum number is zero; with some, the maximum number is unlimited.

Making a Subexpression Optional

❖ *Answer to the question on page 19.*

In this case, "optional" means that it is allowed once, but is not required. That means using ⌜?⌟. Since the thing that's optional is larger than one character, we must use parentheses: ⌜(···)?⌟. Inserting into our expression, we get:

⌜<HR(+SIZE *= *[0-9]+)? *>⌟

Note that the ending ⌜ *⌟ is kept outside of the ⌜(···)?⌟. This still allows something such as <HR >. Had we included it within the parentheses, ending spaces would have been allowed only when the size component was present.

Similarly, notice that the ⌜ +⌟ before SIZE *is* included within the parentheses. Were it left outside them, a space would have been required after the HR, even when the SIZE part wasn't there. This would cause '<HR>' to not match.

Table 1-2: Summary of Quantifier "Repetition Metacharacters"

	Minimum Required	Maximum to Try	Meaning
?	none	1	one allowed; none required ("*one optional*")
*	none	no limit	unlimited allowed; none required ("*any amount okay*")
+	1	no limit	unlimited allowed; one required ("*at least one*")

Defined range of matches: intervals

Some versions of *egrep* support a metasequence for providing your own minimum and maximum: ⌜···{*min*,*max*}⌟. This is called the *interval* quantifier. For example, ⌜···{3,12}⌟ matches up to 12 times if possible, but settles for three. One might use ⌜[a-zA-Z]{1,5}⌟ to match a US stock ticker (from one to five letters). Using this notation, {0,1} is the same as a question mark.

Not many versions of *egrep* support this notation yet, but many other tools do, so it's covered in Chapter 3 when we look in detail at the broad spectrum of metacharacters in common use today.

Parentheses and Backreferences

So far, we have seen two uses for parentheses: to limit the scope of alternation, ⌜|⌟, and to group multiple characters into larger units to which you can apply quantifiers like question mark and star. I'd like to discuss another specialized use that's not common in *egrep* (although GNU's popular version does support it), but which is commonly found in many other tools.

In many regular-expression flavors, parentheses can "remember" text matched by the subexpression they enclose. We'll use this in a partial solution to the doubled-word problem at the beginning of this chapter. If you knew the the specific doubled word to find (such as "the" earlier in this sentence — did you catch it?), you could search for it explicitly, such as with ⌜`the␣the`⌟. In this case, you would also find items such as `the␣theory`, but you could easily get around that problem if your *egrep* supports the word-boundary metasequences ⌜`\<···\>`⌟ mentioned on page 15: ⌜`\<the␣the\>`⌟. We could use ⌜`␣+`⌟ for the space for even more flexibility.

However, having to check for every possible pair of words would be an impossible task. Wouldn't it be nice if we could match one generic word, and then say "now match the same thing again"? If your *egrep* supports *backreferencing*, you can. Backreferencing is a regular-expression feature that allows you to match new text that is the same as some text matched earlier in the expression.

We start with ⌜`\<the␣+the\>`⌟ and replace the initial ⌜`the`⌟ with a regular expression to match a general word, say ⌜`[A-Za-z]+`⌟. Then, for reasons that will become clear in the next paragraph, let's put parentheses around it. Finally, we replace the second '`the`' by the special metasequence ⌜`\1`⌟. This yields ⌜`\<(`**`[A-Za-z]+`**`)␣+`**`\1`**`\>`⌟.

With tools that support backreferencing, parentheses "remember" the text that the subexpression inside them matches, and the special metasequence ⌜`\1`⌟ represents that text later in the regular expression, whatever it happens to be at the time.

Of course, you can have more than one set of parentheses in a regular expression. Use ⌜`\1`⌟, ⌜`\2`⌟, ⌜`\3`⌟, etc., to refer to the first, second, third, etc. sets. Pairs of parentheses are numbered by counting opening parentheses from the left, so with ⌜`([a-z])([0-9])\1\2`⌟, the ⌜`\1`⌟ refers to the text matched by ⌜`[a-z]`⌟, and ⌜`\2`⌟ refers to the text matched by ⌜`[0-9]`⌟.

With our '`the␣the`' example, ⌜`[A-Za-z]+`⌟ matches the first '`the`'. It is within the first set of parentheses, so the '`the`' matched becomes available via ⌜`\1`⌟. If the following ⌜`␣+`⌟ matches, the subsequent ⌜`\1`⌟ will require another '`the`'. If ⌜`\1`⌟ is successful, then ⌜`\>`⌟ makes sure that we are now at an end-of-word boundary (which we wouldn't be were the text '`the␣theft`'). If successful, we've found a repeated word. It's not always the case that that is an error (such as with "that" in this sentence), but that's for you to decide once the suspect lines are shown.

When I decided to include this example, I actually tried it on what I had written so far. (I used a version of *egrep* that supports both ⌜`\<···\>`⌟ and backreferencing.) To make it more useful, so that '`The␣the`' would also be found, I used the case-insensitive `-i` option mentioned on page 15.†

† Be aware that some versions of *egrep*, including the popular GNU version, have a bug with the `-i` option such that it doesn't apply to backreferences. Thus, it finds "the the" but *not* "The the."

Here's the command I ran:

```
% egrep -i '\<([a-z]+) +\1\>' files
```

I was surprised to find fourteen sets of mistakenly '`doubled doubled`' words! I corrected them, and since then have built this type of regular-expression check into the tools that I use to produce the final output of this book, to ensure none creep back in.

As useful as this regular expression is, it is important to understand its limitations. Since *egrep* considers each line in isolation, it isn't able to find when the ending word of one line is repeated at the beginning of the next. For this, a more flexible tool is needed, and we will see some examples in the next chapter.

The Great Escape

One important thing I haven't mentioned yet is how to actually match a character that a regular expression would normally interpret as a metacharacter. For example, if I searched for the Internet hostname `ega.att.com` using ⌈`ega.att.com`⌋, it could end up matching something like `megawatt computing`. Remember, ⌈`.`⌋ is a metacharacter that matches any character, including a space.

The metasequence to match an actual period is a period preceded by a backslash: ⌈`ega\.att\.com`⌋. The sequence ⌈`\.`⌋ is described as an *escaped period* or *escaped dot,* and you can do this with all the normal metacharacters, except in a character-class.†

A backslash used in this way is called an "escape" — when a metacharacter is escaped, it loses its special meaning and becomes a literal character. If you like, you can consider the sequence to be a special metasequence to match the literal character. It's all the same.

As another example, you could use ⌈`\([a-zA-Z]+\)`⌋ to match a word within parentheses, such as '`(very)`'. The backslashes in the ⌈`\(`⌋ and ⌈`\)`⌋ sequences remove the special interpretation of the parentheses, leaving them as literals to match parentheses in the text.

When used before a non-metacharacter, a backslash can have different meanings depending upon the version of the program. For example, we have already seen how some versions treat ⌈`\<`⌋, ⌈`\>`⌋, ⌈`\1`⌋, etc. as metasequences. We will see many more examples in later chapters.

† Most programming languages and tools allow you to escape characters within a character class as well, but most versions of *egrep* do not, instead treating '`\`' within a class as a literal backslash to be included in the list of characters.

Expanding the Foundation

I hope the examples and explanations so far have helped to establish the basis for a solid understanding of regular expressions, but please realize that what I've provided so far lacks depth. There's so much more out there.

Linguistic Diversification

I mentioned a number of regular expression features that most versions of *egrep* support. There are other features, some of which are not supported by all versions, which I'll leave for later chapters.

Unfortunately, the regular expression language is no different from any other in that it has various dialects and accents. It seems each new program employing regular expressions devises its own "improvements." The state of the art continually moves forward, but changes over the years have resulted in a wide variety of regular expression "flavors." We'll see many examples in the following chapters.

The Goal of a Regular Expression

From the broadest top-down view, a regular expression either matches within a lump of text (with *egrep*, each line) or it doesn't. When crafting a regular expression, you must consider the ongoing tug-of-war between having your expression match the lines you want, yet still not matching lines you don't want.

Also, while *egrep* doesn't care where in the line the match occurs, this concern is important for many other regular-expression uses. If your text is something such as

```
...zip is 44272. If you write, send $4.95 to cover postage and...
```

and you merely want to find lines matching `[0-9]+`, you don't care which numbers are matched. However, if your intent is to *do something* with the number (such as save to a file, add, replace, and such—we will see examples of this kind of processing in the next chapter), you'll care very much exactly *which* numbers are matched.

A Few More Examples

As with any language, experience is *a very good thing*, so I'm including a few more examples of regular expressions to match some common constructs.

Half the battle when writing regular expressions is getting successful matches when and where you want them. The other half is to *not* match when and where you don't want. In practice, both are important, but for the moment, I would like to concentrate on the "getting successful matches" aspect. Even though I don't take these examples to their fullest depths, they still provide useful insight.

Variable names

Many programming languages have identifiers (variable names and such) that are allowed to contain only alphanumeric characters and underscores, but which may not begin with a digit. They are matched by ⌜`[a-zA-Z_][a-zA-Z_0-9]*`⌟. The first character class matches what the first character can be, the second (with its accompanying star) allows the rest of the identifier. If there is a limit on the length of an identifier, say 32 characters, you might replace the star with ⌜`{0,31}`⌟ if the ⌜{*min*,*max*}⌟ notation is supported. (This construct, the interval quantifier, was briefly mentioned on page 20.)

A string within double quotes

A simple solution to matching a string within double quotes might be: ⌜`"[^"]*"`⌟

The double quotes at either end are to match the opening and closing double quotes of the string. Between them, we can have anything... except another double quote! So, we use ⌜`[^"]`⌟ to match all characters except a double quote, and apply using a star to indicate we can have any number of such non double-quote characters.

A more useful (but more complex) definition of a double-quoted string allows double quotes within the string if they are escaped with a backslash, such as in `"nail␣the␣2\"x4\"␣plank"`. We'll see this example several times in future chapters while covering the many details of how a match is actually carried out.

Dollar amount (with optional cents)

One approach to matching a dollar amount is: ⌜`\$[0-9]+(\.[0-9][0-9])?`⌟

From a top-level perspective, this is a simple regular expression with three parts: ⌜`\$`⌟ and ⌜⋯`+`⌟ and ⌜`(`⋯`)?`⌟, which might be loosely paraphrased as "a literal dollar sign, a bunch of one thing, and finally perhaps another thing." In this case, the "one thing" is a digit (with a bunch of them being a number), and "another thing" is the combination of a decimal point followed by two digits.

This example is a bit naïve for several reasons. For example, it considers dollar amounts like `$1000`, but not `$1,000`. It does allow for optional cents, but frankly, that's not really very useful when applied with *egrep*. *egrep* never cares exactly *how much* is matched, but merely *whether* there is a match. Allowing something optional at the end never changes whether there's an overall match to begin with.

But, if you need to find lines that contain *just* a price, and nothing else, you can wrap the expression with ⌜`^`⋯`$`⌟. In this case, the optional cents part becomes important since it might or might not come between the dollar amount and the

end of the line, and allowing or disallowing it makes the difference in achieving an overall match.

One type of value our expression doesn't match is '`$.49`'. To solve this, you might be tempted to change the plus to a star, but that doesn't work. As to why, I'll leave it as a teaser until we look at this example again in Chapter 5 (☞194).

An HTTP/HTML URL

The format of web URLs can be complex, so constructing a regular expression to match any possible URL can be equally complex. However, relaxing your standards slightly can allow you to match most common URLs with a fairly simple expression. One common reason I might do this, for example, would be to search my email archive for a URL that I vaguely remember having received, but which I think I might recognize when I see it.

The general form of a common HTTP/HTML URL is along the lines of

```
http://hostname/path.html
```

although ending with `.htm` is common as well.

The rules about what can and can't be a *hostname* (computer name, such as `www.yahoo.com`) are complex, but for our needs we can realize that if it follows '`http://`', it's probably a hostname, so we can make do with something simple, such as `⌈[-a-z0-9_.]+⌋`. The *path* part can be even more varied, so we'll use `⌈[-a-z0-9_:@&?=+,.!/~*'%$]*⌋` for that. Notice that these classes have the dash first, to ensure that it's taken as a literal character and included in the list, as opposed to part of a range (☞9).

Putting these all together, we might use as our first attempt something like:

```
% egrep -i '\<http://[-a-z0-9_.:]+/[-a-z0-9_:@&?=+,.!/~*'%$]*\.html?\>' files
```

Again, since we've taken liberties and relaxed what we'll match, we could well match something such as '`http://..../foo.html`', which is certainly not a valid URL. Do we care about this? It all depends on what you're trying to do. For my scan of my email archive, it doesn't really matter if I get a few false matches. Heck, I could probably get away with even something as simple as:

```
% egrep -i '\<http://[^ ]*\.html?\>' files...
```

As we'll learn when getting deeper into how to craft an expression, knowing the data you'll be searching is an important aspect of finding the balance between complexity and completeness. We'll visit this example again, in more detail, in the next chapter.

An HTML tag

With a tool like *egrep*, it doesn't seem particularly common or useful to simply match lines with HTML tags. But, exploring a regular expression that matches HTML tags exactly can be quite fruitful, especially when we delve into more advanced tools in the next chapter.

Looking at simple cases like '`<TITLE>`' and '`<HR>`', we might think to try ⌈`<.*>`⌋. This simplistic approach is a frequent first thought, but it's certainly incorrect. Converting ⌈`<.*>`⌋ into English reads "match a '<', followed by as much of anything as can be matched, followed by '>'." Well, when phrased that way, it shouldn't be surprising that it can match more than just one tag, such as the marked portion of '`this <I>short</I> example`'.

This might have been a bit surprising, but we're still in the first chapter, and our understanding at this point is only superficial. I have this example here to highlight that regular expressions are not a difficult subject, but they can be tricky if you don't truly understand them. Over the next few chapters, we'll look at all the details required to understand and solve this problem.

Time of day, such as "9:17 am" or "12:30 pm"

Matching a time can be taken to varying levels of strictness. Something such as

⌈`[0-9]?[0-9]:[0-9][0-9] (am|pm)`⌋

picks up both `9:17 am` and `12:30 pm`, but also allows something nonsensical like `99:99 pm`.

Looking at the hour, we realize that if it is a two-digit number, the first digit must be a one. But, ⌈`1?[0-9]`⌋ still allows an hour of `19` (and also an hour of `0`), so maybe it is better to break the hour part into two possibilities: ⌈`1[012]`⌋ for two-digit hours and ⌈`[1-9]`⌋ for single-digit hours. The result is ⌈`(1[012]|[1-9])`⌋.

The minute part is easier. The first digit should be ⌈`[0-5]`⌋. For the second, we can stick with the current ⌈`[0-9]`⌋. This gives ⌈`(1[012]|[1-9]):[0-5][0-9] (am|pm)`⌋ when we put it all together.

Using the same logic, can you extend this to handle 24-hour time with hours from `0` through `23`? As a challenge, allow for a leading zero, at least through to `09:59`. ❖ Try building your solution, and then turn the page to check mine.

Regular Expression Nomenclature

Regex

As you might guess, using the full phrase "regular expression" can get a bit tiring, particularly in writing. Instead, I normally use "regex." It just rolls right off the tongue (it rhymes with "FedEx," with a hard *g* sound like "regular" and not a soft one like in "Regina") and it is amenable to a variety of uses like "when you regex...," "budding regexers," and even "regexification."† I use the phrase "regex engine" to refer to the part of a program that actually does the work of carrying out a match attempt.

Matching

When I say a regex "matches" a string, I really mean that it matches *in* a string. Technically, the regex ⌜`a`⌟ doesn't *match* `cat`, but matches the `a` *in* `cat`. It's not something that people tend to confuse, but it's still worthy of mention.

Metacharacter

Whether a character is a metacharacter (or "metasequence"—I use the words interchangeably) depends on exactly where in the regex it's used. For example, ⌜`*`⌟ is a metacharacter, but only when it's not within a character class and when not escaped. "Escaped" means that it has a backslash in front of it—usually. The star is escaped in ⌜`\*`⌟, but not in ⌜`\\*`⌟ (where the first backslash escapes the second), although the star "has a backslash in front of it" in both examples.

Depending upon the regex flavor, there are various situations when certain characters are and aren't metacharacters. Chapter 3 discusses this in more detail.

Flavor

As I've hinted, different tools use regular expressions for many different things, and the set of metacharacters and other features that each support can differ. Let's look at word boundaries again as an example. Some versions of *egrep* support the `\<`⋯`\>` notation we've seen. However, some do not support the separate word-start and word-end, but one catch-all ⌜`\b`⌟ metacharacter (which we haven't seen yet—we'll see it in the next chapter). Still others support both, and many others support neither.

I use the term "flavor" to describe the sum total of all these little implementation decisions. In the language analogy, it's the same as a dialect of an individual speaker. Superficially, this concept refers to which metacharacters are and aren't

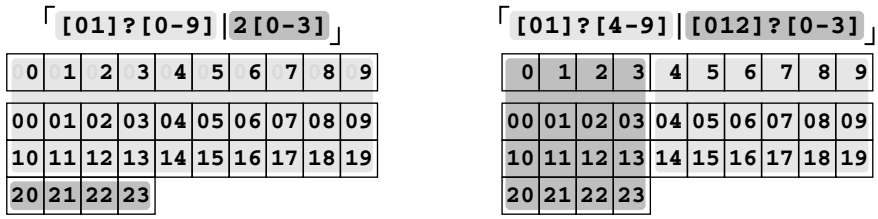
† You might also come across the decidedly unsightly "regexp." I'm not sure how one would pronounce that, but those with a lisp might find it a bit easier.

Extending the Time Regex to Handle a 24-Hour Clock

❖ *Answer to the question on page 26.*

There are various solutions, but we can use similar logic as before. This time, I'll break the task into three groups: one for the morning (hours `00` through `09`, with the leading zero being optional), one for the daytime (hours `10` through `19`), and one for the evening (hours `20` through `23`). This can be rendered in a pretty straightforward way: `0?[0-9]|1[0-9]|2[0-3]`.

Actually, we can combine the first two alternatives, resulting in the shorter `[01]?[0-9]|2[0-3]`. You might need to think about it a bit to convince yourself that they'll really match exactly the same text, but they do. The figure below might help, and it shows another approach as well. The shaded groups represent numbers that can be matched by a single alternative.



supported, but there's much more to it. Even if two programs both support `\<···\>`, they might disagree on exactly what they do and don't consider to be a word. This concern is important when you *use* the tool.

Don't confuse "flavor" with "tool." Just as two people can speak the same dialect, two completely different programs can support exactly the same regex flavor. Also, two programs with the same name (and built to do the same task) often have slightly (and sometimes not-so-slightly) different flavors. Among the various programs called *egrep*, there is a wide variety of regex flavors supported.

In the late 1990s, the particularly expressive flavor offered by the Perl programming language was widely recognized for its power, and soon other languages were offering Perl-inspired regular expressions (many even acknowledging the inspirational source by labeling themselves "Perl-compatible"). The adopters include Python, many Java regex packages, Microsoft's .NET Framework, Tcl, and a variety of C libraries, to name a few. Yet, all are different in important respects. On top of this, Perl's regular expressions themselves are evolving and growing (sometimes, now, in response to advances seen with other tools). As always, the overall landscape continues to become more varied and confusing.

Subexpression

The term "subexpression" simply refers to part of a larger expression, although it often refers to some part of an expression within parentheses, or to an alternative of ⌈|⌋. For example, with ⌈`^(Subject|Date):•`⌋, the ⌈`Subject|Date`⌋ is usually referred to as a subexpression. Within that, the alternatives ⌈`Subject`⌋ and ⌈`Date`⌋ are each referred to as subexpressions as well. But technically, ⌈`S`⌋ is a subexpression, as is ⌈`u`⌋, and ⌈`b`⌋, and ⌈`j`⌋, . . .

Something such as `1-6` isn't considered a subexpression of ⌈`H[1-6]•*`⌋, since the '`1-6`' is part of an unbreakable "unit," the character class. But, ⌈`H`⌋, ⌈`[1-6]`⌋, and ⌈`•*`⌋ are all subexpressions of ⌈`H[1-6]•*`⌋.

Unlike alternation, quantifiers (star, plus, and question mark) always work with the smallest immediately-preceding subexpression. This is why with ⌈`mis+pell`⌋, the `+` governs the ⌈`s`⌋, not the ⌈`mis`⌋ or ⌈`is`⌋. Of course, when what immediately precedes a quantifier is a parenthesized subexpression, the entire subexpression (no matter how complex) is taken as one unit.

Character

The word "character" can be a loaded term in computing. The character that a byte represents is merely a matter of interpretation. A byte with such-and-such a value has that same value in any context in which you might wish to consider it, but which *character* that value *represents* depends on the encoding in which it's viewed. As a concrete example, two bytes with decimal values `64` and `53` represent the characters "@" and "5" respectively, if considered in the ASCII encoding, yet on the other hand are completely different if considered in the EBCDIC encoding (they are a space and some kind of a control character).

On the third hand, if those two bytes are considered in one of the popular encodings for Japanese characters, together they represent the single character 正. Yet, to represent this same character in another of the Japanese encodings requires two completely different bytes. Those two different bytes, by the way, yield the two characters "Àµ" in the popular *Latin-1* encoding, but yield the one Korean character 삵 in one of the Unicode encodings.† The point is this: how bytes are to be interpreted is a matter of perspective (called an *encoding*), and to be successful, you've got to make sure that your perspective agrees with the perspective taken by the tool you're using.

† The definitive book on multiple-byte encodings is Ken Lunde's *CJKV Information Processing*, also published by O'Reilly & Associates. The CJKV stands for *Chinese*, *Japanese*, *Korean*, and *Vietnamese*, which are languages that tend to require multiple-byte encodings. Ken and Adobe kindly provided many of the special fonts used in this book.

Until recently, text-processing tools generally treated their data as a bunch of ASCII bytes, without regard to the encoding you might be intending. Recently, however, more and more systems are using some form of Unicode to process data internally (Chapter 3 includes an introduction to Unicode ☞105). On such systems, if the regular-expression subsystem has been implemented properly, the user doesn't normally have to pay much attention to these issues. That's a big "if," which is why Chapter 3 looks at this issue in depth.

Improving on the Status Quo

When it comes down to it, regular expressions are not difficult. But, if you talk to the average user of a program or language that supports them, you will likely find someone that understands them "a bit," but does not feel secure enough to really use them for anything complex or with any tool but those they use most often.

Traditionally, regular expression documentation tends to be limited to a short and incomplete description of one or two metacharacters, followed by a table of the rest. Examples often use meaningless regular expressions like `a*((ab)*|b*)`, and text like '`a•xxx•ce•xxxxxx•ci•xxx•d`'. They also tend to completely ignore subtle but important points, and often claim that their flavor is the same as some other well-known tool, almost always forgetting to mention the exceptions where they inevitably differ. The state of regex documentation needs help.

Now, I don't mean to imply that this chapter fills the gap for all regular expressions, or even for *egrep* regular expressions. Rather, this chapter merely provides the foundation upon which the rest of this book is built. It may be ambitious, but I hope this book does fill the gaps for you. I received many gratifying responses to the first edition, and have worked very hard to make this one even better, both in breadth and in depth.

Perhaps because regular-expression documentation has traditionally been so lacking, I feel the need to make the extra effort to make things particularly clear. Because I want to make sure you can use regular expressions to their fullest potential, I want to make sure you really, *really* understand them.

This is both good and bad.

It is good because you will learn how to *think* regular expressions. You will learn which differences and peculiarities to watch out for when faced with a new tool with a different flavor. You will know how to express yourself even with a weak, stripped-down regular expression flavor. You will understand what makes one expression more efficient than another, and will be able to balance tradeoffs among complexity, efficiency, and match results. When faced with a particularly complex task, you will know how to work through an expression the way the

program would, constructing it as you go. In short, you will be comfortable using regular expressions to their fullest.

The problem is that the learning curve of this method can be rather steep, with three separate issues to tackle:

- **How regular expressions are used** Most programs use regular expressions in ways that are more complex than *egrep*. Before we can discuss in detail how to write a really useful expression, we need to look at the ways regular expressions can be used. We start in the next chapter.
- **Regular expression features** Selecting the proper tool to use when faced with a problem seems to be half the battle, so I don't want to limit myself to only using one utility throughout this book. Different programs, and often even different versions of the same program, provide different features and metacharacters. We must survey the field before getting into the details of using them. This is the subject of Chapter 3.
- **How regular expressions really work** Before we can learn from useful (but often complex) examples, we need to "look under the hood" to understand just how a regular expression search is conducted. As we'll see, the order in which certain metacharacters are checked can be very important. In fact, regular expression engines can be implemented in different ways, so different programs sometimes do different things with the same expression. We examine this meaty subject in Chapters 4, 5, and 6.

This last point is the most important and the most difficult to address. The discussion is unfortunately sometimes a bit dry, with the reader chomping at the bit to get to the fun part — tackling real problems. However, understanding how the regex engine really works is the key to *really understanding*.

You might argue that you don't want to be taught how a car works when you simply want to know how to drive. But, learning to drive a car is a poor analogy for learning about regular expressions. My goal is to teach you how to solve problems with regular expressions, and that means constructing regular expressions. The better analogy is not how to drive a car, but how to build one. Before you can build a car, you have to know how it works.

Chapter 2 gives more experience with driving. Chapter 3 takes a short look at the history of driving, and a detailed look at the bodywork of a regex flavor. Chapter 4 looks at the all-important engine of a regex flavor. Chapter 5 shows some extended examples, Chapter 6 shows you how to tune up certain kinds of engines, and the chapters after that examine some specific makes and models. Particularly in Chapters 4, 5, and 6, we'll spend a lot of time under the hood, so make sure to have your coveralls and shop rags handy.

Summary

Table 1-3 summarizes the *egrep* metacharacters we've looked at in this chapter.

Table 1-3: Egrep Metacharacter Summary

Metacharacter		Matches
Items to Match a Single Character		
`.`	*dot*	Matches any one character
`[…]`	*character class*	Matches any one character listed
`[^…]`	*negated character class*	Matches any one character not listed
`\`*char*	*escaped character*	When *char* is a metacharacter, or the escaped combination is not otherwise special, matches the literal *char*
Items Appended to Provide "Counting" : The Quantifiers		
`?`	*question*	One allowed, but it is optional
`*`	*star*	Any number allowed, but all are optional
`+`	*plus*	At least one required; additional are optional
`{`*min*`,`*max*`}`	*specified range*†	*Min* required, *max* allowed
Items That Match a Position		
`^`	*caret*	Matches the position at the start of the line
`$`	*dollar*	Matches the position at the end of the line
`\<`	*word boundary*†	Matches the position at the start of a word
`\>`	*word boundary*†	Matches the position at the end of a word
Other		
`|`	alternation	Matches either expression it separates
`(…)`	*parentheses*	Limits scope of alternation, provides grouping for the quantifiers, and "captures" for backreferences
`\1`, `\2`, ...	*backreference*†	Matches text previously matched within first, second, etc., set of parentheses.

†*not supported by all versions of egrep*

In addition, be sure that you understand the following points:

- Not all *egrep* programs are the same. The metacharacters supported, as well as their exact meanings, are often different — see your local documentation (☞ 23).
- Three reasons for using parentheses are constraining alternation (☞ 13), grouping (☞ 14), and capturing (☞ 21).
- Character classes are special, and have their own set of metacharacters totally distinct from the "main" regex language (☞ 10).

- Alternation and character classes are fundamentally different, providing unrelated services that appear, in only one limited situation, to overlap (☞ 13).
- A negated character class is still a "positive assertion"—even negated, a character class must match a character to be successful. Because the listing of characters to match is negated, the matched character must be one of those *not* listed in the class (☞ 12).
- The useful `-i` option discounts capitalization during a match (☞ 15).
- There are three types of escaped items:
  1. The pairing of ⌜\⌟ and a metacharacter is a metasequence to match the literal character (for example, ⌜*⌟ matches a literal asterisk).
  2. The pairing of ⌜\⌟ and selected non-metacharacters becomes a metasequence with an implementation-defined meaning (for example, ⌜\<⌟ often means "start of word").
  3. The pairing of ⌜\⌟ and any other character defaults to simply matching the character (that is, the backslash is ignored).

  Remember, though, that a backslash within a character class is not special at all with most versions of *egrep*, so it provides no "escape services" in such a situation.
- Items governed by a question mark or star don't need to actually match any characters to "match successfully." They are *always* successful, even if they don't match anything (☞ 17).

Personal Glimpses

The doubled-word task at the start of this chapter might seem daunting, yet regular expressions are so powerful that we could solve much of the problem with a tool as limited as *egrep*, right here in the first chapter. I'd like to fill this chapter with flashy examples, but because I've concentrated on the solid foundation for the later chapters, I fear that someone completely new to regular expressions might read this chapter, complete with all the warnings and cautions and rules and such, and feel "why bother?"

Recently, my brothers were teaching some friends how to play *schaffkopf*, a card game that's been in my family for generations. It is much more exciting than it appears at first glance, but has a rather steep learning curve. After about half an hour, my sister-in-law Liz, normally the quintessence of patience, got frustrated with the seemingly complex rules and said "Can't we just play rummy?" Yet, as it turned out, they ended up playing late into the night. Once they were able to get

over the initial hump of the learning curve, a first-hand taste of the excitement was all it took to hook them. My brothers knew it would, but it took some time and work to get to the point where Liz and the others new to the game could appreciate what they were getting into.

It might take some time to become acclimated to regular expressions, so until you get a real taste of the excitement by using them to solve *your* problems, it might all feel just a bit too academic. If so, I hope you will resist the desire to "play rummy." Once you understand the power that regular expressions provide, the small amount of work spent learning them will feel trivial indeed.