# 4

# *The Mechanics of Expression Processing*

The previous chapter started with an analogy between cars and regular expressions. The bulk of the chapter discussed features, regex flavors, and other "glossy brochure" issues of regular expressions. This chapter continues with that analogy, talking about the all-important regular-expression engine, and how it goes about its work.

Why would you care how it works? As we'll see, there are several types of regex engines, and the type most commonly used—the type used by Perl, Tcl, Python, the .NET languages, Ruby, PHP, all Java packages I've seen, and more—works in such a way that *how* you craft your expression can influence *whether* it can match a particular string, *where* in the string it matches, and *how quickly* it finds the match or reports the failure. If these issues are important to you, this chapter is for you.

## *Start Your Engines!*

Let's see how much I can milk this engine analogy. The whole point of having an engine is so that you can get from Point A to Point B without doing much work. The engine does the work for you so you can relax and enjoy the sound system. The engine's primary task is to turn the wheels, and how it does that isn't really a concern of yours. Or is it?

## Two Kinds of Engines

Well, what if you had an electric car? They've been around for a long time, but they aren't as common as gas cars because they're hard to design well. If you had one, though, you would have to remember not to put gas in it. If you had a gasoline engine, well, watch out for sparks! An electric engine more or less just runs, but a gas engine might need some babysitting. You can get much better performance just by changing little things like your spark plug gaps, air filter, or brand of gas. Do it wrong and the engine's performance deteriorates, or, worse yet, it stalls.

Each engine might do its work differently, but the end result is that the wheels turn. You still have to steer properly if you want to get anywhere, but that's an entirely different issue.

## New Standards

Let's stoke the fire by adding another variable: the California Emissions Standards.† Some engines adhere to California's strict pollution standards, and some engines don't. These aren't really different kinds of engines, just new variations on what's already around. The standard regulates a result of the engine's work, the emissions, but doesn't say anything about how the engine should go about achieving those cleaner results. So, our two classes of engine are divided into four types: electric (adhering and non-adhering) and gasoline (adhering and non-adhering).

Come to think of it, I bet that an electric engine can qualify for the standard without much change—the standard just "blesses" the clean results that are already par for the course. The gas engine, on the other hand, needs some major tweaking and a bit of re-tooling before it can qualify. Owners of this kind of engine need to pay particular care to what they feed it—use the wrong kind of gas and you're in big trouble.

### The impact of standards

Better pollution standards are a good thing, but they require that the driver exercise more thought and foresight (well, at least for gas engines). Frankly, however, the standard doesn't impact most people since all the other states still do their own thing and don't follow California's standard.

So, you realize that these four types of engines can be classified into three groups (the two kinds for gas, and electric in general). You know about the differences, and that in the end they all still turn the wheels. What you don't know is what the heck this has to do with regular expressions! More than you might imagine.

† California has rather strict standards regulating the amount of pollution a car can produce. Because of this, many cars sold in America come in "California" and "non-California" models.

## *Regex Engine Types*

There are two fundamentally different types of regex engines: one called "DFA" (the electric engine of our story) and one called "NFA" (the gas engine). The details of what NFA and DFA mean follow shortly (☞156), but for now just consider them names, like Bill and Ted. Or electric and gas.

Both engine types have been around for a long time, but like its gasoline counterpart, the NFA type seems to be used more often. Tools that use an NFA engine include the .NET languages, Ruby, Perl, Python, GNU Emacs, *ed*, sed, PHP, *vi*, most versions of *grep*, and even a few versions of *egrep* and awk. On the other hand, a DFA engine is found in almost all versions of *egrep* and awk, as well as *lex* and *flex*. Some systems have a multi-engine hybrid system, using the most appropriate engine for the job (or even one that swaps between engines for different parts of the same regex, as needed to get the best combination of features and speed). Table 4-1 lists a few common programs and the regex engine that most versions use. If your favorite program is not in the list, the section "Testing the Engine Type" on the next page can help you find out which it is.

*Table 4-1: Some Tools and Their Regex Engines*

| Engine type | Programs |
|---|---|
| DFA | awk (most versions), *egrep* (most versions), flex, lex, MySQL, Procmail |
| Traditional NFA | GNU Emacs, Java, *grep* (most versions), *less*, *more*, .NET languages, PCRE library, Perl, PHP (pcre routines), Python, Ruby, sed (most versions), *vi* |
| POSIX NFA | *mawk*, Mortice Kern Systems' utilities, GNU Emacs (when requested) |
| Hybrid NFA/DFA | GNU awk, GNU *grep*/*egrep*, Tcl |

As Chapter 3 illustrated, 20 years of development with both DFAs and NFAs resulted in a lot of needless variety. Things were dirty. The POSIX standard came in to clean things up by clearly specifying not only which metacharacters and features an engine should support, as mentioned in the previous chapter, but also *exactly the results you could expect from them.* Superficial details aside, the DFAs (our electric engines) were already well suited to adhere to this new standard, but the kind of results an NFA traditionally provided were different, so changes were needed. As a result, broadly speaking, there are three types of regex engines:

- DFA (POSIX or not—similar either way)
- Traditional NFA (most common: Perl, .NET, Java, Python, . . . )
- POSIX NFA

Here, we use "POSIX" to refer to the match *semantics*—the expected operation of a regex that the POSIX standard specifies (which we'll get to later in this chapter). Don't confuse this use of "POSIX" with uses surrounding regex *features* introduced

in that same standard (☞ 125). Many programs support the features without supporting the full match semantics.

Old (and minimally featured) programs like *egrep*, awk, and *lex* were normally implemented with the electric DFA engine, so the new standard primarily just confirmed the status quo — no big changes. However, there *were* some gas-powered versions of these programs which had to be changed if they wanted to be POSIX-compliant. The gas engines that passed the California Emission Standards tests (POSIX NFA) were fine in that they produced results according to the standard, but the necessary changes only increased how fickle they were to proper tuning. Where before you might get by with slightly misaligned spark plugs, you now have absolutely no tolerance. Gasoline that used to be "good enough" now causes knocks and pings. But, so long as you know how to maintain your baby, the engine runs smoothly and cleanly.

## From the Department of Redundancy Department

At this point, I'll ask you to go back and review the story about engines. Every sentence there rings with some truth about regular expressions. A second reading should raise some questions. Particularly, what does it mean that an electric DFA regex engine more or less "just runs?" What affects a gas-powered NFA? How can I tune my regular expressions to run as I want on an NFA? What special concerns does an emissions-controlled POSIX NFA have? What's a "stalled engine" in the regex world?

## Testing the Engine Type

Because the type of engine used in a tool influences the type of features it can offer, and how those features appear to work, we can often learn the type of engine a tool has merely by checking to see how it handles a few test expressions. (After all, if you can't tell the difference, the difference doesn't matter.) At this point in the book, I wouldn't expect you to understand why the following test results indicate what they do, but I want to offer these tests now so that if your favorite tool is not listed in Table 4-1, you can investigate before continuing with this and the subsequent chapters.

### *Traditional NFA or not?*

The most commonly used engine is a Traditional NFA, and spotting it is easy. First, are lazy quantifiers (☞ 140) supported? If so, it's almost certainly a Traditional NFA. As we'll see, lazy quantifiers are not possible with a DFA, nor would they make any sense with a POSIX NFA. However, to make sure, simply apply the regex `⌈nfa|nfa·not⌋` to the string '`nfa·not`'—if only '`nfa`' matches, it's a Traditional NFA. If the entire '`nfa·not`' matches, it's either a POSIX NFA or a DFA.

### *DFA or POSIX NFA?*

Differentiating between a POSIX NFA and a DFA is sometimes just as simple. Capturing parentheses and backreferences are not supported by a DFA, so that can be one hint, but there are systems that are a hybrid mix between the two engine types, and so may end up using a DFA if there are no capturing parentheses.

Here's a simple test that can tell you a lot. Apply `X(.+)+X` to a string like '`=XX======================`', as with this *egrep* command:

```
echo =XX======================================== | egrep 'X(.+)+X'
```

If it takes a long time to finish, it's an NFA (and if not a Traditional NFA as per the test in the previous section, it must be a POSIX NFA). If it finishes quickly, it's either a DFA or an NFA with some advanced optimization. Does it display a warning message about a stack overflow or long match aborted? If so, it's an NFA.

# *Match Basics*

Before looking at the differences among these engine types, let's first look at their similarities. Certain aspects of the drive train are the same (or for all practical purposes appear to be the same), so these examples can cover all engine types.

## *About the Examples*

This chapter is primarily concerned with a generic, full-function regex engine, so some tools won't support exactly everything presented. In my examples, the dipstick might be to the left of the oil filter, while under your hood it might be behind the distributor cap. Your goal is to understand the concepts so that you can drive and maintain your favorite regex package (and ones you find interest in later).

I'll continue to use Perl's notation for most of the examples, although I'll occasionally show others to remind you that the *notation* is superficial and that the issues under discussion transcend any one tool or flavor. To cut down on wordiness here, I'll rely on you to check Chapter 3 (☞ 113) if I use an unfamiliar construct.

This chapter details the practical effects of how a match is carried out. It would be nice if everything could be distilled down to a few simple rules that could be memorized without needing to understand what is going on. Unfortunately, that's not the case. In fact, with all this chapter offers, I identify only two all-encompassing rules:

1. The match that begins earliest (leftmost) wins.
2. The standard quantifiers (`*`, `+`, `?`, and `{m,n}`) are greedy.

We'll look at these rules, their effects, and much more throughout this chapter. Let's start by diving into the details of the first rule.

## Rule 1: The Match That Begins Earliest Wins

This rule says that any match that begins earlier (leftmost) in the string is always preferred over any plausible match that begins later. This rule doesn't say anything about how long the winning match might be (we'll get into that shortly), merely that among all the matches possible anywhere in the string, the one that begins leftmost in the string is chosen. Actually, since more than one plausible match can start at the same earliest point, perhaps the rule should read "*a* match..." instead of "*the* match...," but that sounds odd.

Here's how the rule comes about: the match is first attempted at the very beginning of the string to be searched (just before the first character). "Attempted" means that every permutation of the entire (perhaps complex) regex is tested starting right at that spot. If all possibilities are exhausted and a match is not found, the complete expression is re-tried starting from just before the second character. This full retry occurs at each position in the string until a match is found. A "no match" result is reported only if no match is found after the full retry has been attempted at each position all the way to the end of the string (just after the last character).

Thus, when trying to match ⌈ORA⌋ against FLORAL, the first attempt at the start of the string fails (since ⌈ORA⌋ can't match FLO). The attempt starting at the second character also fails (it doesn't match LOR either). The attempt starting at the third position, however, does match, so the engine stops and reports the match: FLORAL.

If you didn't know this rule, results might sometimes surprise you. For example, when matching ⌈cat⌋ against

```
The dragging belly indicates your cat is too fat
```

the match is in indicates, not at the word cat that appears later in the line. This word cat *could* match, but the cat in indi**cat**es appears earlier in the string, so it is the one matched. For an application like *egrep*, the distinction is irrelevant because it cares only *whether* there is a match, not *where* the match might be. For other uses, such as with a search-and-replace, the distinction becomes paramount.

Here's a (hopefully simple) quiz: where does ⌈fat|cat|belly|your⌋ match in the string 'The dragging belly indicates your cat is too fat'? ❖ Turn the page to check your answer.

### The "transmission" and the bump-along

It might help to think of this rule as the car's transmission, connecting the engine to the drive train while adjusting for the gear you're in. The engine itself does the real work (turning the crank); the transmission transfers this work to the wheels.

***The transmission's main work: the bump-along***

If the engine can't find a match starting at the beginning of the string, it's the transmission that bumps the regex engine along to attempt a match at the next position in the string, and the next, and the next, and so on. Usually. For instance, if a regex begins with a start-of-string anchor, the transmission can realize that any bump-along would be futile, for only the attempt at the start of the string could possibly be successful. This and other internal optimizations are discussed in Chapter 6.

## *Engine Pieces and Parts*

An engine is made up of parts of various types and sizes. You can't possibly hope to truly understand how the whole thing works if you don't know much about the individual parts. In a regex, these parts are the individual units—literal characters, quantifiers (star and friends), character classes, parentheses, and so on, as described in Chapter 3 (☞ 113). The combination of these parts (and the engine's treatment of them) makes a regex what it is, so looking at ways they can be combined and how they interact is our primary interest. First, let's take a look at some of the individual parts:

**Literal text** (e.g., `a` `\*` `!` `枝` ... )

With a literal, non-metacharacter like ⌈`z`⌋ or ⌈`!`⌋, the match attempt is simply "Does this literal character match the current text character?" If your regex is only literal text, such as ⌈`usa`⌋, it is treated as "⌈`u`⌋ and then ⌈`s`⌋ and then ⌈`a`⌋." It's a bit more complicated if you have the engine do a case-insensitive match, where ⌈`b`⌋ matches `B` and vice-versa, but it's still pretty straightforward. (With Unicode, there are a few additional twists ☞ 109.)

**Character classes, dot, Unicode properties, and the like**

Matching dot, character classes, Unicode properties, and the like (☞ 117) is usually a simple matter: regardless of the length of the character class, it still matches just one character.†

Dot is just a shorthand for a large character class that matches almost any character (☞ 110), so its actions are simple, as are the other shorthand conveniences such as ⌈`\w`⌋, ⌈`\W`⌋, and ⌈`\d`⌋.

**Capturing parentheses**

Parentheses used only for capturing text (as opposed to those used for grouping) don't change how the match is carried out.

† Actually, as we saw in the previous chapter (☞ 126), a POSIX collating sequence *can* match multiple characters, but this is not common. Also, certain Unicode characters can match multiple characters when applied in a case-insensitive manner (☞ 109), although most implementations do not support this.

## Quiz Answer

❖ *Answer to the question on page 148.*

Remember, the regex is tried *completely* each time, so ⌈`fat|cat|belly|your`⌋ matches '`The dragging belly indicates your cat is too fat`' rather than `fat`, even though ⌈`fat`⌋ is listed first among the alternatives.

Sure, the regex could conceivably match `fat` and the other alternatives, but since they are not the *earliest* possible match (the match starting furthest to the left), they are not the one chosen. The entire regex is attempted completely from one spot before moving along the string to try again from the next spot, and in this case that means trying each alternative ⌈`fat`⌋, ⌈`cat`⌋, ⌈`belly`⌋, and ⌈`your`⌋ at each position before moving on.

**Anchors** (e.g., ⌈`^`⌋ ⌈`\Z`⌋ ⌈`(?<=\d)`⌋ ... )

There are two basic types of anchors: simple ones (`^`, `$`, `\G`, `\b`, ... ☞127) and complex ones (lookahead and lookbehind ☞132). The simple ones are indeed simple in that they test either the quality of a particular location in the target string (`^`, `\Z`, ...), or compare two adjacent characters (`\<`, `\b`, ...). On the other hand, the lookaround constructs can contain arbitrary sub-expressions, and so can be arbitrarily complex.

### *No "electric" parentheses, backreferences, or lazy quantifiers*

I'd like to concentrate here on the similarities among the engines, but as foreshadowing of what's to come in this chapter, I'll point out a few interesting differences. Capturing parentheses (and the associated backreferences and `$1` type functionality) are like a gas additive—they affect a gasoline (NFA) engine, but are irrelevant to an electric (DFA) engine. The same thing applies to lazy quantifiers. The way a DFA engine works completely precludes these concepts.† This explains why tools developed with DFAs don't provide these features. You'll notice that awk, *lex*, and *egrep* don't have backreferences or any `$1` type functionality.

You might, however, notice that GNU's version of *egrep* **does** support backreferences. It does so by having two complete engines under the hood! It first uses a DFA engine to see whether a match is likely, and then uses an NFA engine (which supports the full flavor, including backreferences) to confirm the match. Later in this chapter, we'll see why a DFA engine can't deal with backreferences or capturing, and why anyone ever would want to use such an engine at all. (It has some major advantages, such as being able to match very quickly.)

† This does not mean that there can't be some mixing of technologies to try to get the best of both worlds. This is discussed in a sidebar on page 183.

## *Rule 2: The Standard Quantifiers Are Greedy*

So far, we have seen features that are quite straightforward. They are also rather boring—you can't *do* much without involving more-powerful metacharacters such as star, plus, alternation, and so on. Their added power requires more information to understand them fully.

First, you need to know that the standard quantifiers (`?`, `*`, `+`, and `{`*min*`,`*max*`}`) are *greedy*. When one of these governs a subexpression, such as ⌈`a`⌋ in ⌈`a?`⌋, the ⌈`(`*expr*`)`⌋ in ⌈`(`*expr*`)*`⌋, or ⌈`[0-9]`⌋ in ⌈`[0-9]+`⌋, there is a minimum number of matches that are required before it can be considered successful, and a maximum number that it will ever attempt to match. This has been mentioned in earlier chapters—what's new here concerns the rule that they always attempt to match as much as possible. (Some flavors provide other types of quantifiers, but this section is concerned only with the standard, greedy ones.)

To be clear, the standard quantifiers settle for something less than the maximum number of allowed matches *if they have to*, but they always attempt to match as many times as they can, up to that maximum allowed. The only time they settle for anything less than their maximum allowed is when matching too much ends up causing some later part of the regex to fail. A simple example is using ⌈`\b\w+s\b`⌋ to match words ending with an '`s`', such as '`regexes`'. The ⌈`\w+`⌋ alone is happy to match the entire word, but if it does, it leaves nothing for the ⌈`s`⌋ to match. To achieve the overall match, the ⌈`\w+`⌋ must settle for matching only '`regexe`s', thereby allowing ⌈`s\b`⌋ (and thus the full regex) to match.

If it turns out that the only way the rest of the regex can succeed is when the greedy construct in question matches nothing, well, that's perfectly fine, if zero matches are allowed (as with star, question, and `{0,`*max*`}` intervals). However, it turns out this way only if the requirements of some later subexpression force the issue. It's because the greedy quantifiers always (or, at least, try to) take more than they minimally need that they are called greedy.

Greediness has many useful (but sometimes troublesome) implications. It explains, for example, why ⌈`[0-9]+`⌋ matches the full number in `March 1998`. Once the '`1`' has been matched, the plus has fulfilled its minimum requirement, but it's greedy, so it doesn't stop. So, it continues, and matches the '`998`' before being forced to stop by the end of the string. (Since ⌈`[0-9]`⌋ can't match the nothingness at the end of the string, the plus finally stops.)

### *A subjective example*

Of course, this method of grabbing things is useful for more than just numbers. Let's say you have a line from an email header and want to check whether it is the subject line. As we saw in earlier chapters (☞55), you simply use ⌈`^Subject:`⌋.

However, if you use ⌜^Subject:␣(.*)⌟, you can later access the text of the subject itself via the tool's after-the-fact parenthesis memory (for example, $1 in Perl).†

Before looking at why ⌜.*⌟ matches the entire subject, be sure to understand that once the ⌜^Subject:␣⌟ part matches, you're guaranteed that the entire regular expression will eventually match. You know this because there's nothing after ⌜^Subject:␣⌟ that could cause the expression to fail; ⌜.*⌟ can *never* fail, since the worst case of "no matches" is still considered successful for star.

So, why do we even bother adding ⌜.*⌟? Well, we know that because star is greedy, it attempts to match dot as many times as possible, so we use it to "fill" $1. In fact, the parentheses add nothing to the logic of what the regular expression matches—in this case we use them simply to capture the text matched by ⌜.*⌟.

Once ⌜.*⌟ hits the end of the string, the dot isn't able to match, so the star finally stops and lets the next item in the regular expression attempt to match (for even though the starred dot could match no further, perhaps a subexpression later in the regex could). Ah, but since it turns out that there is no next item, we reach the end of the regex and we know that we have a successful match.

### *Being too greedy*

Let's get back to the concept of a greedy quantifier being as greedy as it can be. Consider how the matching and results would change if we add another ⌜.*⌟: ⌜^Subject:␣(.*).*⌟. The answer is: nothing would change. The initial ⌜.*⌟ (inside the parentheses) is so greedy that it matches all the subject text, never leaving anything for the second ⌜.*⌟ to match. Again, the failure of the second ⌜.*⌟ to match something is not a problem, since the star does not require a match to be successful. Were the second ⌜.*⌟ in parentheses as well, the resulting $2 would always be empty.

Does this mean that after ⌜.*⌟, a regular expression can never have anything that is expected to actually match? No, of course not. As we saw with the ⌜\w+s⌟ example, it is possible for something later in the regex to *force* something previously greedy to give back (that is, relinquish or conceptually "unmatch") if that's what is necessary to achieve an overall match.

Let's consider the possibly useful ⌜^.*([0-9][0-9])⌟, which finds the *last* two digits on a line, wherever they might be, and saves them to $1. Here's how it works: at first, ⌜.*⌟ matches the entire line. Because the following ⌜([0-9][0-9])⌟ is *required*, its initial failure to match at the end of the line, in effect, tells ⌜.*⌟ "Hey, you took too much! Give me back something so that I can have a chance to

† This example uses capturing as a forum for presenting greediness, so the example itself is appropriate only for NFAs (because only NFAs support capturing). The lessons on greediness, however, apply to all engines, including the non-capturing DFA.

match." Greedy components first try to take as much as they can, but they always defer to the greater need to achieve an overall match. They're just stubborn about it, and only do so when forced. Of course, they'll never give up something that hadn't been optional in the first place, such as a plus quantifier's first match.

With this in mind, let's apply ⌈`^.*([0-9][0-9])`⌋ to '`about·24·characters·long`'. Once ⌈`.*`⌋ matches the whole string, the requirement for the first ⌈`[0-9]`⌋ to match forces ⌈`.*`⌋ to give up '`g`' (the last thing it had matched). That doesn't, however, allow ⌈`[0-9]`⌋ to match, so ⌈`.*`⌋ is again forced to relinquish something, this time the '`n`'. This cycle continues 15 more times until ⌈`.*`⌋ finally gets around to giving up '`4`'.

Unfortunately, even though the first ⌈`[0-9]`⌋ can then match that '`4`', the second still cannot. So, ⌈`.*`⌋ is forced to relinquish once more in an attempt fo find an overall match. This time ⌈`.*`⌋ gives up the '`2`', which the first ⌈`[0-9]`⌋ can then match. Now, the '`4`' is free for the second ⌈`[0-9]`⌋ to match, and so the entire expression matches '`about·24·char···`', with `$1` getting '`24`'.

### *First come, first served*

Consider now using ⌈`^.*([0-9]+)`⌋, ostensibly to match not just the last two digits, but the last whole number, however long it might be. When this regex is applied to '`Copyright 2003.`', what is captured? ❖ Turn the page to check your answer.

### *Getting down to the details*

I should clear up a few things here. Phrases like "the ⌈`.*`⌋ *gives up...*" and "the ⌈`[0-9]`⌋ *forces...*" are slightly misleading. I used these terms because they're easy to grasp, and the end result appears to be the same as reality. However, what really happens behind the scenes depends on the basic engine type, DFA or NFA. So, it's time to see what these really are.

# *Regex-Directed Versus Text-Directed*

The two basic engine types reflect a fundamental difference in algorithms available for applying a regular expression to a string. I call the gasoline-driven NFA engine "regex-directed," and the electric-driven DFA "text-directed."

## *NFA Engine: Regex-Directed*

Let's consider one way an engine might match ⌈`to(nite|knight|night)`⌋ against the text '`···tonight···`'. Starting with the ⌈`t`⌋, the regular expression is examined one component at a time, and the "current text" is checked to see whether it is matched by the current component of the regex. If it does, the next component is checked, and so on, until all components have matched, indicating that an overall match has been achieved.

## Quiz Answer

❖ *Answer to the question on page 153.*

**When ⌈`^.*([0-9]+)`⌋ is applied to '`Copyright 2003.`', what is captured by the parentheses?**

The desire is to get the last whole number, but it doesn't work. As before, ⌈`.*`⌋ is forced to relinquish some of what it had matched because the subsequent ⌈`[0-9]+`⌋ requires a match to be successful. In this example, that means unmatching the final period and '`3`', which then allows ⌈`[0-9]`⌋ to match. That's governed by ⌈`+`⌋, so matching just once fulfills its minimum, and now facing '`.`' in the string, it finds nothing else to match.

Unlike before, though, there's then nothing further that *must* match, so ⌈`.*`⌋ is not forced to give up the `0` or any other digits it might have matched. Were ⌈`.*`⌋ to do so, the ⌈`[0-9]+`⌋ would certainly be a grateful and greedy recipient, but nope, first come first served. Greedy constructs give up something they've matched only when forced. In the end, `$1` gets only '`3`'.

If this feels counter-intuitive, realize that ⌈`[0-9]+`⌋ is at most one match away from ⌈`[0-9]*`⌋, which is in the same league as ⌈`.*`⌋. Substituting that into ⌈`^.*([0-9]+)`⌋, we get ⌈`^.*(.*)`⌋ as our regex, which looks suspiciously like the ⌈`^Subject:␣(.*).*`⌋ example from page 152, where the second ⌈`.*`⌋ was guaranteed to match nothing.

With the ⌈`to(nite|knight|night)`⌋ example, the first component is ⌈`t`⌋, which repeatedly fails until a '`t`' is reached in the target string. Once that happens, the ⌈`o`⌋ is checked against the next character, and if it matches, control moves to the next component. In this case, the "next component" is ⌈`(nite|knight|night)`⌋ which really means "⌈`nite`⌋ or ⌈`knight`⌋ or ⌈`night`⌋." Faced with three possibilities, the engine just tries each in turn. We (humans with advanced neural nets between our ears) can see that if we're matching `tonight`, the third alternative is the one that leads to a match. Despite their brainy origins (☞ 85), a regex-directed engine can't come to that conclusion until actually going through the motions to check.

Attempting the first alternative, ⌈`nite`⌋, involves the same component-at-a-time treatment as before: "Try to match ⌈`n`⌋, then ⌈`i`⌋, then ⌈`t`⌋, and finally ⌈`e`⌋." If this fails, as it eventually does, the engine tries another alternative, and so on until it achieves a match or must report failure. Control moves within the regex from component to component, so I call it "regex-directed."

### *The control benefits of an NFA engine*

In essence, each subexpression of a regex in a regex-directed match is checked independently of the others. Other than backreferences, there's no interrelation among subexpressions, except for the relation implied by virtue of being thrown together to make a larger expression. The layout of the subexpressions and regex control structures (e.g., alternation, parentheses, and quantifiers) controls an engine's overall movement through a match.

Since the regex directs the NFA engine, the driver (the writer of the regular expression) has considerable opportunity to craft just what he or she wants to happen. (Chapters 5 and 6 show how to put this to use to get a job done correctly and efficiently.) What this really means may seem vague now, but it will all be spelled out soon.

## *DFA Engine: Text-Directed*

Contrast the regex-directed NFA engine with an engine that, while scanning the string, keeps track of all matches "currently in the works." In the `tonight` example, the moment the engine hits `t`, it adds a potential match to its list of those currently in progress:

| in string | in regex |
|---|---|
| after ···t·onight··· | possible matches: ⌈to(nite\|knight\|night)⌋ |

Each subsequent character scanned updates the list of possible matches. After a few more characters are matched, the situation becomes

| in string | in regex |
|---|---|
| after ···toni·ght··· | possible matches: ⌈to(ni·te\|knight\|ni·ght)⌋ |

with two possible matches in the works (and one alternative, `knight`, ruled out). With the `g` that follows, only the third alternative remains viable. Once the `h` and `t` are scanned as well, the engine realizes it has a complete match and can return success.

I call this "text-directed" matching because each character scanned from the text controls the engine. As in the example, a partial match might be the start of any number of different, yet possible, matches. Matches that are no longer viable are pruned as subsequent characters are scanned. There are even situations where a "partial match in progress" is also a full match. If the regex were ⌈`to(···)?`⌋, for example, the parenthesized expression becomes optional, but it's still greedy, so it's always attempted. All the time that a partial match is in progress inside those parentheses, a full match (of '`to`') is already confirmed and in reserve in case the longer matches don't pan out.

If the engine reaches a character in the text that invalidates all the matches in the works, it must revert to one of the full matches in reserve. If there are none, it must declare that there are no matches at the current attempt's starting point.

## First Thoughts: NFA and DFA in Comparison

If you compare these two engines based only on what I've mentioned so far, you might conclude that the text-directed DFA engine is generally faster. The regex-directed NFA engine might waste time attempting to match different subexpressions against the same text (such as the three alternatives in the example).

You would be right. During the course of an NFA match, the same character of the target might be checked by many different parts of the regex (or even by the same part, over and over). Even if a subexpression can match, it might have to be applied again (and again and again) as it works in concert with the rest of the regex to find a match. A local subexpression can fail or match, but you just never know about the overall match until you eventually work your way to the end of the regex. (If I could find a way to include "It's not over until the fat lady sings." in this paragraph, I would.) On the other hand, a DFA engine is *deterministic*—each character in the target is checked once (at most). When a character matches, you don't know yet if it will be part of the final match (it could be part of a possible match that doesn't pan out), but since the engine keeps track of all possible matches in parallel, it needs to be checked only once, period.

The two basic technologies behind regular-expression engines have the somewhat imposing names *Nondeterministic Finite Automaton* (NFA) and *Deterministic Finite Automaton* (DFA). With mouthfuls like this, you see why I stick to just "NFA" and "DFA." We won't be seeing these phrases spelled out again.†

### Consequences to us as users

Because of the regex-directed nature of an NFA, the details of how the engine attempts a match are very important. As I said before, the writer can exercise a fair amount of control simply by changing how the regex is written. With the `tonight` example, perhaps less work would have been wasted had the regex been written differently, such as in one of the following ways:

- ⌈`to(ni(ght|te)|knight)`⌋
- ⌈`tonite|toknight|tonight`⌋
- ⌈`to(k?night|nite)`⌋

† I suppose I could explain the underlying theory that goes into these names, if I only knew it! As I hinted, the word *deterministic* is pretty important, but for the most part the theory is not relevant, so long as we understand the practical effects. By the end of this chapter, we will.

With any given text, these all end up matching exactly the same thing, but in doing so direct the engine in different ways. At this point, we don't know enough to judge which of these, if any, are better than the others, but that's coming soon.

It's the exact opposite with a DFA — since the engine keeps track of all matches simultaneously, none of these differences in representation matter so long as in the end they all represent the same set of possible matches. There could be a hundred different ways to achieve the same result, but since the DFA keeps track of them all simultaneously (almost magically — more on this later), it doesn't matter which form the regex takes. To a pure DFA, even expressions that appear as different as ⌈`abc`⌋ and ⌈`[aa-a](b|b{1}|b)c`⌋ are utterly indistinguishable.

Three things come to my mind when describing a DFA engine:

- DFA matching is very fast.
- DFA matching is very consistent.
- Talking about DFA matching is very boring.

I'll eventually expand on all these points.

The regex-directed nature of an NFA makes it interesting to talk about. NFAs provide plenty of room for creative juices to flow. There are great benefits in crafting an expression well, and even greater penalties for doing it poorly. A gasoline engine is not the only engine that can stall and conk out completely. To get to the bottom of this, we need to look at the essence of an NFA engine: *backtracking*.

# *Backtracking*

The essence of an NFA engine is this: it considers each subexpression or component in turn, and whenever it needs to decide between two equally viable options, it selects one and remembers the other to return to later if need be.

Situations where it has to decide among courses of action include anything with a quantifier (decide whether to try another match), and alternation (decide which alternative to try, and which to leave for later).

Whichever course of action is attempted, if it's successful and the rest of the regex is also successful, the match is finished. If anything in the rest of the regex eventually causes failure, the regex engine knows it can *backtrack* to where it chose the first option, and can continue with the match by trying the other option. This way, it eventually tries all possible permutations of the regex (or at least as many as needed until a match is found).

## A Really Crummy Analogy

Backtracking is like leaving a pile of bread crumbs at every fork in the road. If the path you choose turns out to be a dead end, you can retrace your steps, giving up ground until you come across a pile of crumbs that indicates an untried path. Should that path, too, turn out to be a dead end, you can backtrack further, retracing your steps to the next pile of crumbs, and so on, until you eventually find a path that leads to your goal, or until you run out of untried paths.

There are various situations when the regex engine needs to choose between two (or more) options — the alternation we saw earlier is only one example. Another example is that upon reaching ⌈···x?···⌋, the engine must decide whether it should attempt ⌈x⌋. Upon reaching ⌈···x+···⌋, however, there is no question about trying to match ⌈x⌋ at least once — the plus requires at least one match, and that's non-negotiable. Once the first ⌈x⌋ has been matched, though, the *requirement* is lifted and it then must decide to match another ⌈x⌋. If it decides to match, it must decide if it will then attempt to match yet another... and another... and so on. At each of these many decision points, a virtual "pile of crumbs" is left behind as a reminder that another option (to match or not to match, whichever wasn't chosen at each point) remains viable at that point.

### A crummy little example

Let's look at a full example using our earlier ⌈to(nite|knight|night)⌋ regex on the string 'hot•tonic•tonight!' (silly, yes, but a good example). The first component, ⌈t⌋, is attempted at the start of the string. It fails to match h, so the entire regex fails at that point. The engine's transmission then bumps along to retry the regex from the second position (which also fails), and again at the third. This time the ⌈t⌋ matches, but the subsequent ⌈o⌋ fails to match because the text we're at is now a space. So, again, the whole attempt fails.

The attempt that eventually starts at ···▲tonic··· is more interesting. Once the to has been matched, the three alternatives become three available options. The regex engine picks one to try, remembering the others ("leaving some bread crumbs") in case the first fails. For the purposes of discussion, let's say that the engine first chooses ⌈nite⌋. That expression breaks down to "⌈n⌋ + ⌈i⌋ + ⌈t⌋ ...," which gets to ···to▲nic··· before failing. Unlike the earlier failures, this failure doesn't mean the end of the overall attempt because other options — the as-of-yet untried alternatives — still remain. (In our analogy, we still have piles of breadcrumbs we can return to.) The engine chooses one, we'll say ⌈knight⌋, but it fails right away because ⌈k⌋ doesn't match 'n'. That leaves one final option, ⌈night⌋, but it too eventually fails. Since that was the final untried option, its failure means the failure of the entire attempt starting at ···▲tonic···, so the transmission kicks in again.

Once the engine works its way to attempt the match starting at ···tonight!, it gets interesting again. This time, the night alternative successfully matches to the end (which means an overall match, so the engine can report success at that point).

## *Two Important Points on Backtracking*

The general idea of how backtracking works is fairly simple, but some of the details are quite important for real-world use. Specifically, when faced with multiple choices, which choice should be tried first? Secondly, when forced to backtrack, which saved choice should the engine use? The answer to that first question is this important principle:

> In situations where the decision is between "make an attempt" and "skip an attempt," as with items governed by quantifiers, the engine always chooses to first *make* the attempt for *greedy* quantifiers, and to first *skip* the attempt for *lazy* (non-greedy) ones.

This has far-reaching repercussions. For starters, it helps explain why the greedy quantifiers are greedy, but it doesn't explain it completely. To complete the picture, we need to know which (among possibly many) saved options to use when we backtrack. Simply put:

> The most recently saved option is the one returned to when a local failure forces backtracking. They're used LIFO (last in first out).

This is easily understood in the crummy analogy—if your path becomes blocked, you simply retrace your steps until you come back across a pile of bread crumbs. The first you'll return to is the one most recently laid. The traditional analogy for describing LIFO also holds: like stacking and unstacking dishes, the most-recently stacked will be the first unstacked.

## *Saved States*

In NFA regular expression nomenclature, the piles of bread crumbs are known as saved *states*. A state indicates where matching can restart from, if need be. It reflects both the position in the regex and the point in the string where an untried option begins. Because this is the basis for NFA matching, let me show the implications of what I've already said with some simple but verbose examples. If you're comfortable with the discussion so far, feel free to skip ahead.

### A match without backtracking

Let's look at a simple example, matching ⌈ab?c⌋ against abc. Once the ⌈a⌋ has matched, the *current state* of the match is reflected by:

| at 'a▲bc' | matching ⌈a▲b?c⌋ |
|---|---|

However, now that ⌈b?⌋ is up to match, the regex engine has a decision to make: should it attempt the ⌈b⌋, or skip it?. Well, since ? is greedy, it attempts the match. But, so that it can recover if that attempt fails or eventually leads to failure, it adds

| at 'a▲bc' | matching ⌈ab?▲c⌋ |
|---|---|

to its otherwise empty list of saved states. This indicates that the engine can later pick up the match in the regex just *after* the ⌈b?⌋, picking up in the text from just before the b (that is, where it is now). Thus, in effect, skipping the ⌈b⌋ as the question mark allows.

Once the engine carefully places that pile of crumbs, it goes ahead and checks the ⌈b⌋. With the example text, it matches, so the new current state becomes:

| at 'ab▲c' | matching ⌈ab?▲c⌋ |
|---|---|

The final ⌈c⌋ matches as well, so we have an overall match. The one saved state is no longer needed, so it is simply forgotten.

### A match after backtracking

Now, if 'ac' had been the text to match, everything would have been the same until the ⌈b⌋ attempt was made. Of course, this time it wouldn't match. This means that the path that resulted from actually attempting the ⌈···?⌋ failed. Since there is a saved state available to return to, this "local failure" does not mean overall failure. The engine backtracks, meaning that it takes the most recently saved state as its new current state. In this case, that would be the

| at 'a▲c' | matching ⌈ab?▲c⌋ |
|---|---|

state that had been saved as the untried option before the ⌈b⌋ had been attempted. This time, the ⌈c⌋ and c match up, so the overall match is achieved.

### A non-match

Now let's look at the same expression, but against 'abX'. Before the ⌈b⌋ is attempted, the question mark causes this state to be saved:

| at 'a▲bX' | matching ⌈ab?▲c⌋ |
|---|---|

The ⌈b⌋ matches, but that avenue later turns out to be a dead end because the ⌈c⌋ fails to match x. The failure results in a backtrack to the saved state. The engine next tests ⌈c⌋ against the b that the backtrack effectively "unmatched." Obviously, this test fails, too. If there were other saved states, another backtrack would occur, but since there aren't any, the overall match at the current starting position is deemed a failure.

Are we done? Nope. The engine's transmission still does its "bump along the string and retry the regex," which might be thought of as a pseudo-backtrack. The match restarts at:

| at 'a▲bx' | matching ⌈ab?c⌋ |
|---|---|

The whole match is attempted again from the new spot, and like before, all paths lead to failure. After the next two attempts (from ab▲x and abx▲) similarly fail, overall failure is finally reported.

### *A lazy match*

Let's look at the original example, but with a lazy quantifier, matching ⌈a**b??**c⌋ against 'abc'. Once the ⌈a⌋ has matched, the state of the match is reflected by:

| at 'a▲bc' | matching ⌈a▲**b??**c⌋ |
|---|---|

Now that ⌈b??⌋ is next to be applied, the regex engine has a decision to make: attempt the ⌈b⌋ or skip it? Well, since ?? is lazy, it specifically chooses to first skip the attempt, but, so that it can recover if that attempt fails or eventually leads to failure, it adds

| at 'a▲bc' | matching ⌈a▲**b**c⌋ |
|---|---|

to its otherwise empty list of saved states. This indicates that the engine can later pick up the match by making the attempt of ⌈b⌋, in the text from just before the b. (We know it will match, but the regex engine doesn't yet know that, or even know if it will ever need to get as far as making the attempt.) Once the state has been saved, it goes ahead and continues from after its skip-the-attempt decision:

| at 'a▲bc' | matching ⌈a**b??**▲c⌋ |
|---|---|

The ⌈c⌋ fails to match 'b', so indeed the engine must backtrack to its one saved state:

| at 'a▲bc' | matching ⌈a▲**b**c⌋ |
|---|---|

Of course, it matches this time, and the subsequent ⌈c⌋ matches 'c'. The same final match we got with the greedy ⌈a**b?**c⌋ is achieved, although via a different path.

## Backtracking and Greediness

For tools that use this NFA regex-directed backtracking engine, understanding how backtracking works with your regular expression is the key to writing expressions that accomplish what you want, and accomplish it quickly. We've seen how ⌜?⌟ greediness and ⌜??⌟ laziness works, so now let's look at star and plus.

### Star, plus, and their backtracking

If you consider ⌜x*⌟ to be more or less the same as ⌜x?x?x?x?x?x?···⌟ (or, more appropriately, ⌜(x(x(x(x···?)?)?)?)?⌟),† it's not too different from what we have already seen. Before checking the item quantified by the star, the engine saves a state indicating that if the check fails (or leads to failure), the match can pick up after the star. This is done repeatedly, until an attempt via the star actually does fail.

Thus, when matching ⌜[0-9]+⌟ against 'a•1234•num', once ⌜[0-9]⌟ fails to match the space after the 4, there are four saved states corresponding to locations to which the plus can backtrack:

```
a 1▲234 num
a 12▲34 num
a 123▲4 num
a 1234▲ num
```

These represent the fact that the attempt of ⌜[0-9]⌟ had been optional at each of these positions. When ⌜[0-9]⌟ fails to match the space, the engine backtracks to the most recently saved state (the last one listed), picking up at 'a•1234▲•num' in the text and at ⌜[0-9]+▲⌟ in the regex. Well, that's at the end of the regex. Now that we're actually there and notice it, we realize that we have an overall match.

Note that 'a•1234•num' is not in the list of positions, because the first match using the plus quantifier is required, not optional. Would it have been in the list had the regex been ⌜[0-9]*⌟? *(hint: it's a trick question)* ❖ Turn the page to check your answer.

### Revisiting a fuller example

With our more detailed understanding, let's revisit the ⌜^.*([0-9][0-9])⌟ example from page 152. This time, instead of just pointing to "greediness" to explain why the match turns out as it does, we can use our knowledge of NFA mechanics to explain why in precise terms.

I'll use 'CA•95472,•USA' as an example. Once the ⌜.*⌟ has successfully matched to the end of the string, there are a baker's dozen saved states accumulated from the

† Just for comparison, remember that a DFA doesn't care much about the form you use to express which matches are possible; the three examples *are* identical to a DFA.

star-governed dot matching 13 things that are (if need be) optional. These states note that the match can pick up in the regex at ⌈`^.*`▲`([0-9][0-9])`⌋, and in the string at each point where a state was created.

Now that we've reached the end of the string and pass control to the first ⌈`[0-9]`⌋, the match obviously fails. No problem: we have a saved state to try (a baker's dozen of them, actually). We backtrack, resetting the current state to the one most recently saved, to just before where ⌈`.*`⌋ matched the final `A`. Skipping that match (or "unmatching" it, if you like) gives us the opportunity to try that `A` against the first ⌈`[0-9]`⌋. But, it fails.

This backtrack-and-test cycle continues until the engine effectively unmatches the `2`, at which point the first ⌈`[0-9]`⌋ can match. The second can't, however, so we must continue to backtrack. It's now irrelevant that the first ⌈`[0-9]`⌋ matched during the previous attempt; the backtrack resets the current state to before the first ⌈`[0-9]`⌋. As it turns out, the same backtrack resets the string position to just before the `7`, so the first ⌈`[0-9]`⌋ can match again. This time, so can the second (matching the `2`). Thus, we have a match: '`CA 95472, USA`', with `$1` getting '`72`'.

A few observations: first, backtracking entails not only recalculating our position within the regex and the text, but also maintaining the status of the text being matched by the subexpression within parentheses. Each backtrack caused the match to be picked up before the parentheses, at ⌈`^.*`▲`([0-9][0-9])`⌋. As far as the simple match attempt is concerned, this is the same as ⌈`^.*`▲`[0-9][0-9]`⌋, so I used phrases such as "picks up before the first ⌈`[0-9]`⌋." However, moving in and out of the parentheses involves updating the status of what `$1` should be, which also has an impact on efficiency.

One final observation that may already be clear to you: something governed by star (or any of the greedy quantifiers) first matches as much as it can *without regard to what might follow in the regex*. In our example, the ⌈`.*`⌋ does not magically know to stop at the first digit, or the second to the last digit, or any other place until what's governed by the greedy quantifier — the dot — finally fails. We saw this earlier when looking at how ⌈`^.*([0-9]+)`⌋ would never have more than a single digit matched by the ⌈`[0-9]+`⌋ part (☞ 153).

# *More About Greediness and Backtracking*

Many concerns (and benefits) of greediness are shared by both an NFA and a DFA. (A DFA doesn't support laziness, which is why we've concentrated on greediness up to this point.) I'd like to look at some ramifications of greediness for both, but with examples explained in terms of an NFA. The lessons apply to a DFA just as well, but not for the same reasons. A DFA is greedy, period, and there's not much

## Quiz Answer

❖ *Answer to the question on page 162.*

**When matching ⌜[0-9]*⌟ against 'a•1234•num', would 'a•1234•num' be part of a saved state?**

The answer is "no." I posed this question because the mistake is commonly made. Remember, a component that has star applied can *always* match. If that's the entire regex, it can always match anywhere. This certainly includes the attempt when the transmission applies the engine the first time, at the start of the string. In this case, the regex matches at 'a•1234•num' and that's the end of it—it never even gets as far the digits.

In case you missed this, there's still a chance for partial credit. Had there been something in the regex after the ⌜[0-9]*⌟ that kept an overall match from happening before the engine got to:

| at 'a•1234…' | matching ⌜[0-9]*…⌟ |
|---|---|

then indeed, the attempt of the '1' also creates the state:

| at 'a•1234…' | matching ⌜[0-9]*…⌟ |
|---|---|

more to say after that. It's very easy to use, but pretty boring to talk about. An NFA, however, is interesting because of the creative outlet its regex-directed nature provides. Besides lazy quantifiers, there are a variety of extra features an NFA can support, including lookaround, conditionals, backreferences, and atomic grouping. And on top of these, an NFA affords the regex author direct control over how a match is carried out, which can be a benefit when used properly, but it does create some efficiency-related pitfalls (discussed in Chapter 6.)

Despite these differences, the match results are often similar. For the next few pages, I'll talk of both engine types, but describe effects in terms of the regex-directed NFA. By the end of this chapter, you'll have a firm grasp of just when the results might differ, as well as exactly why.

## Problems of Greediness

As we saw with the last example, ⌜.*⌟ always marches to the end of the line.† This is because ⌜.*⌟ just thinks of itself and grabs what it can, only later giving up something if it is required to achieve an overall match.

† With a tool or mode where a dot can match a newline, ⌜.*⌟ applied to strings that contain multiline data matches through all the logical lines to the end of the whole string.

Sometimes this can be a real pain. Consider a regex to match text wrapped in double quotes. At first, you might want to write ⌜`".*"`⌟, but knowing what we know about ⌜`.*`⌟, guess where it matches in:

```
The name "McDonald's" is said "makudonarudo" in Japanese
```

Actually, since we understand the mechanics of matching, we don't need to guess, because we *know*. Once the initial quote matches, ⌜`.*`⌟ is free to match, and immediately does so all the way to the end of the string. It backs off (or, perhaps more appropriately, *is backed off* by the regex engine) only as much as is needed until the final quote can match. In the end, it matches

```
The name "McDonald's" is said "makudonarudo" in Japanese
```

which is obviously not the double-quoted string that was intended. This is one reason why I caution against the overuse of ⌜`.*`⌟, as it can often lead to surprising results if you don't pay careful attention to greediness.

So, how can we have it match `"McDonald's"` only? The key is to realize that we don't want "anything" between the quotes, but rather "anything except a quote." If we use ⌜`[^"]*`⌟ rather than ⌜`.*`⌟, it won't overshoot the closing quote.

The regex engine's basic approach with ⌜`"[^"]*"`⌟ is exactly the same as before. Once the initial double quote matches, ⌜`[^"]*`⌟ gets a shot at matching as much as it can. In this case, that's up to the double quote after `McDonald's`, at which point it finally stops because ⌜`[^"]`⌟ can't match the quote. At that point, control moves to the closing ⌜`"`⌟. It happily matches, resulting in overall success:

```
The name "McDonald's" is said "makudonarudo" in Japanese
```

Actually, there could be one unexpected change, and that's because in most flavors, ⌜`[^"]`⌟ can match a newline, while dot doesn't. If you want to keep the regex from crossing lines, use ⌜`[^"\n]`⌟.

## *Multi-Character "Quotes"*

In the first chapter, I talked a bit about matching HTML tags, such as the sequence `<B>very</B>` that renders the "very" in bold if the browser can do so. Attempting to match a `<B>`···`</B>` sequence seems similar to matching a quoted string, except the "quotes" in this case are the multi-character sequences `<B>` and `</B>`. Like the quoted string example, multiple sets of "quotes" cause problems if we use ⌜`.*`⌟:

```
··<B>Billions</B> and <B>Zillions</B> of suns··
```

With ⌜`<B>.*</B>`⌟, the greedy ⌜`.*`⌟ causes the match in progress to zip to the end of the line, backtracking only far enough to allow the ⌜`</B>`⌟ to match, matching the last `</B>` on the line instead of the one corresponding to the opening ⌜`<B>`⌟ at the start of the match.

Unfortunately, since the closing delimiter is more than one character, we can't solve the problem with a negated class as we did with double-quoted strings. We can't expect something like ⌈<B>**[^</B>]***</B>⌋ to work. A character class represents only one character and not the full </B> sequence that we want. Don't let the apparent structure of ⌈[^</B>]⌋ fool you. It is just a class to match one character — any one except <, >, /, and B. It is the same as, say ⌈[^/<>B]⌋, and certainly doesn't work as an "anything not </B>" construct. (With lookahead, you can insist that ⌈</B>⌋ not match at a particular point; we'll see this in action in the next section.)

## *Using Lazy Quantifiers*

These problems arise because the standard quantifiers are greedy. Some NFAs support lazy quantifiers (☞ 140), with *? being the lazy version of *. With that in mind, let's apply ⌈<B>**.*?**</B>⌋ to:

···<B>Billions</B> and <B>Zillions</B> of suns···

After the initial ⌈<B>⌋ has matched, ⌈.*?⌋ immediately decides that since it doesn't require any matches, it lazily doesn't bother trying to perform any. So, it immediately passes control to the following ⌈<⌋:

| at '···<B>Billions···' | matching ⌈<B>.*?</B>⌋ |
|---|---|

The ⌈<⌋ doesn't match at that point, so control returns back to ⌈.*?⌋ where it still has its untried option to attempt a match (to attempt multiple matches, actually). It begrudgingly does so, with the dot matching the underlined B in ···<B>Billions···. Again, the *? has the option to match more, or to stop. It's lazy, so it first tries stopping. The subsequent ⌈<⌋ still fails, so ⌈.*?⌋ has to again exercise its untried match option. After eight cycles, ⌈.*?⌋ eventually matches Billions, at which point the subsequent ⌈<⌋ (and the whole ⌈</B>⌋ subexpression) is finally able to match:

···<B>Billions</B> and <B>Zillions</B> of suns···

So, as we've seen, the greediness of star and friends can be a real boon at times, while troublesome at others. Having non-greedy, lazy versions is wonderful, as they allow you to do things that are otherwise very difficult (or even impossible). Still, I've often seen inexperienced programmers use lazy quantifiers in inappropriate situations. In fact, what we've just done may not be appropriate. Consider applying ⌈<B>.*?</B>⌋ to:

···<B>Billions and <B>Zillions</B> of suns···

It matches as shown, and while I suppose it depends on the exact needs of the situation, I would think that in this case that match is not desired. However, there's nothing about ⌈.*?⌋ to stop it from marching right past the Zillion's <B> to its </B>.

This is an excellent example of why a lazy quantifier is often not a good replacement for a negated class. In the ⌜`".*"`⌟ example, using ⌜`[^"]`⌟ as a replacement for the dot specifically disallows it from marching past a delimiter—a quality we wish our current regex had.

However, if *negative lookahead* (☞ 132) is supported, you can use it to create something comparable to a negated class. Alone, ⌜`(?!<B>)`⌟ is a test that is successful if `<B>` is not at the current location in the string. Those are the locations that we want the dot of ⌜`<B>.*?</B>`⌟ to match, so changing that dot to ⌜`(?!<B>).`⌟ creates a regex that matches where we want it, but doesn't match where we don't. Assembled all together, the whole thing can become quite confusing, so I'll show it here in a free-spacing mode (☞ 110) with comments:

```
<B>              # Match the opening <B>
(                # Now, only as many of the following as needed . . .
  (?! <B>  )     #      If not <B> . . .
  .              #                  . . . any character is okay
)*?              #
</B>             #  . . . until the closing delimiter can match
```

With one adjustment to the lookahead, we can put the quantifier back to a normal greedy one, which may be less confusing to some:

```
<B>                # Match the opening <B>
(                  # Now, only as many of the following as needed . . .
  (?! </?B>  )     #      If not <B>, and not </B> . . .
  .                #                . . . any character is okay
)*                 #
</B>               #  . . . until the closing delimiter can match.
```

Now, the lookahead prohibits the main body to match beyond `</B>` as well as `<B>`, which eliminates the problem we tried to solve with laziness, so the laziness can be removed. This expression can still be improved; we'll see it again during the discussion on efficiency in Chapter 6 (☞ 270).

## *Greediness and Laziness Always Favor a Match*

Recall the price display example from Chapter 2 (☞ 51). We'll examine this example in detail at a number of points during this chapter, so I'll recap the basic issue: due to floating-point representation problems, values that should have been "1.625" or "3.00" were sometimes coming out like "1.62500000002828" and "3.00000000028822". To fix this, I used

```
$price =~ s/(\.\d\d[1-9]?)\d*/$1/;
```

to lop off all but the first two or three decimal digits from the value stored in the variable `$price`. The ⌜`\.\d\d`⌟ matches the first two decimal digits regardless, while the ⌜`[1-9]?`⌟ matches the third digit only if it is non-zero.

I then noted:

> Anything matched so far is what we want to *keep*, so we wrap it in parentheses to capture to `$1`. We can then use `$1` in the replacement string. If this is the only thing that matches, we replace exactly what was matched with itself—not very useful. However, we go on to match other items outside the `$1` parentheses. They don't find their way to the replacement string, so the effect is that they're removed. In this case, the "to be removed" text is any extra digits, the ⌜`\d*`⌟ at the end of the regex.

So far so good, but let's consider what happens when the contents of the variable `$price` is already well formed. When it is `27.625`, the ⌜`(\.\d\d[1-9]?)`⌟ part matches the entire decimal part. Since the trailing ⌜`\d*`⌟ doesn't match anything, the substitution replaces the '`.625`' with '`.625`' — an effective no-op.

This is the desired result, but wouldn't it be just a bit more efficient to do the replacement only when it would have some real effect (that is, do the replacement only when ⌜`\d*`⌟ actually matches something)? Well, we know how to write "at least one digit"! Simply replace ⌜`\d*`⌟ with ⌜`\d+`⌟:

```
$price =~ s/(\.\d\d[1-9]?)\d+/$1/
```

With crazy numbers like "1.62500000002828", it still works as before, but with something such as "9.43", the trailing ⌜`\d+`⌟ isn't able to match, so rightly, no substitution occurs. So, this is a great modification, yes? *No!* What happens with a three-digit decimal value like `27.625`? We want this value to be left alone, but that's not what happens. Stop for a moment to work through the match of `27.625` yourself, with particular attention to how the '`5`' interacts with the regex.

In hindsight, the problem is really fairly simple. Picking up in the action once ⌜`(\.\d\d[1-9]?)\d+`⌟ has matched `27.625`, we find that ⌜`\d+`⌟ can't match. That's no problem for the overall match, though, since as far as the regex is concerned, the match of '`5`' by ⌜`[1-9]`⌟ was *optional* and there is still a saved state to try. This state allows ⌜`[1-9]?`⌟ to match nothing, leaving the `5` to fulfill the must-match-one requirement of ⌜`\d+`⌟. Thus, we get the match, but not the right match: `.625` is replaced by `.62`, and the value becomes incorrect.

What if ⌜`[1-9]?`⌟ were lazy instead? We'd get the same match, but without the intervening "match the `5` but then give it back" steps, since the lazy ⌜`[1-9]??`⌟ first skips the match attempt. So, laziness is not a solution to this problem.

## The Essence of Greediness, Laziness, and Backtracking

The lesson of the preceding section is that it makes no difference whether there are greedy or lazy components to a regex; an overall match takes precedence over an overall non-match. This includes taking from what had been greedy (or giving to what had been lazy) if that's what is required to achieve a match, because when

a "local failure" is hit, the engine keeps going back to the saved states (retracing steps to the piles of bread crumbs), trying the untested paths. Whether greedily or lazily, *every possible path is tested before the engine admits failure.*

The order that the paths are tested is different between greedy and lazy quantifiers (after all, that's the whole point of having the two!), but in the end, if no match is to be found, it's known only after testing every possible path.

If, on the other hand, there exists just *one* plausible match, both a regex with a greedy quantifier and one with a lazy quantifier find that match, although the series of paths they take to get there may be wildly different. In these cases, selecting greedy or lazy doesn't influence what is matched, but merely how long or short a path the engine takes to get there (which is an efficiency issue, the subject of Chapter 6).

Finally, if there is more than one plausible match, understanding greediness, laziness, and backtracking allows you to know which is selected. The ⌜`".*"`⌟ example has three plausible matches:

```
The name "McDonald's" is said "makudonarudo" in Japanese
```

We know that ⌜`".*"`⌟, with the greedy star, selects the longest one, and that ⌜`".*?"`⌟, with the lazy star, selects the shortest.

## *Possessive Quantifiers and Atomic Grouping*

The '`.625`' example on the facing page shows important insights about NFA matching as we know it, and how with that particular example our naïve intents were thwarted. Some flavors do provide tools to help us here, but before looking at them, it's absolutely essential to fully understand the preceding section, "The Essence of Greediness, Laziness, and Backtracking." Be sure to review it if you have any doubts.

So, continuing with the '`.625`' example and recalling what we really want to happen, we know that if the matching can successfully get to the marked position in ⌜`(\.\d\d[1-9]?)\d+`⌟, we never want it to go back. That is, we want ⌜`[1-9]`⌟ to match if possible, but if it does, we don't want that match to be given up. Saying it more forcefully, we would rather have the entire match attempt fail, if need be, before giving up something matched by the ⌜`[1-9]`⌟. (As you'll recall, the problem before when this regex was applied to '`.625`' was that it indeed *didn't* fail, but instead went back to try the remaining skip-me alternative.)

Well, what if we could somehow eliminate that skip-me alternative (eliminate the state that ⌜`?`⌟ saves before it makes the attempt to match ⌜`[1-9]`⌟)? If there was no state to go back to, a match of ⌜`[1-9]`⌟ wouldn't be given up. That's what we want! Ah, but if there was no skip-me state to go back to, what would happen if we

applied the regex to '.5000'? The ⌈[1-9]⌋ couldn't match, and in this case, we *do* want it to go back and skip the ⌈[1-9]⌋ so that the subsequent ⌈\d+⌋ can match digits to be removed.

It sounds like we have two conflicting desires, but thinking about it, what we really want is to eliminate the skip-me alternative only if the match-me alternative succeeds. That is, if ⌈[1-9]⌋ is indeed able to match, we'd like to get rid of the skip-me saved state so that it is never given up. This *is* possible, with regex flavors that support ⌈(?>···)⌋ atomic grouping (☞ 137), or possessive quantifiers like ⌈[1-9]?+⌋ (☞ 140). We'll look at atomic grouping first.

### *Atomic grouping with ⌈(?>···)⌋*

In essence, matching within ⌈(?>···)⌋ carries on normally, but if and when matching is able to exit the construct (that is, get past its closing parenthesis), all states that had been saved while within it are thrown away. In practice, this means that once the atomic grouping has been exited, whatever text was matched within it is now one unchangeable unit, to be kept or given back only as a whole. All saved states representing untried options within the parentheses are eliminated, so backtracking can never undo any of the decisions made within (at least not once they're "locked in" when the construct is exited).

So, let's consider ⌈(\.\d\d(?>[1-9]?))\d+⌋. Quantifiers work normally within atomic grouping, so if ⌈[1-9]⌋ is not able to match, the regex returns to the skip-me saved state the ⌈?⌋ had left. That allows matching to leave the atomic grouping and continue on to the ⌈\d+⌋. In this case, there are no saved states to flush when control leaves the atomic grouping (that is, there are no saved states remaining that had been created within it).

However, when ⌈[1-9]⌋ *is* able to match, matching can exit the atomic grouping, but this time, the skip-me state is still there. Since it had been created within the atomic grouping we're now exiting, it is thrown away. This would happen when matching against both '.625', and, say, '.625000'. In the latter case, having eliminated the state turns out not to matter, since the ⌈\d+⌋ has the '.625000' to match, after which that regex is done. With '.625' alone, the inability of ⌈\d+⌋ to match has the regex engine wanting to backtrack, but it can't since that skip-me alternative was thrown away. The lack of any state to backtrack to results in the overall match attempt failing, and '.625' is left undisturbed as we wish.

#### *The essence of atomic grouping*

The section "The Essence of Greediness, Laziness, and Backtracking," starting on page 168, makes the important point that neither greediness nor laziness influence *which* paths can be checked, but merely the *order* in which they are checked. If no match is found, whether by a greedy or a lazy ordering, in the end, every possible path will have been checked.

Atomic grouping, on the other hand, is fundamentally different because it actually *eliminates possible paths*. Eliminating states can have a number of different consequences, depending on the situation:

- **No Effect** If a match is reached before one of the eliminated states would have been called upon, there is no effect on the match. We saw this a moment ago with the '`.625000`' example. A match was found before the eliminated state would have come into play.
- **Prohibit Match** The elimination of states can mean that a match that would have otherwise been possible now becomes impossible. We saw this with the '`.625`' example.
- **Different Match** In some cases, it's possible to get a *different* match due to the elimination of states.
- **Faster Failure** It's possible for the elimination of states to do nothing more than allow the regex engine, when no match is to be found, report that fact more quickly. This is discussed right after the quiz.

Here's a little quiz: what does the construct ⌜`(?>.*?)`⌟ do? What kind of things do you expect it can match? ❖ Turn the page to check your answer.

***Some states may remain.*** When the engine exits atomic grouping during a match, only states that had been created *while inside the atomic grouping* are eliminated. States that might have been there before still remain after, so the entire text matched by the atomic subexpression may be unmatched, as a whole, if backtracking later reverts to one of those previous states.

***Faster failures with atomic grouping.*** Consider ⌜`^\w+:`⌟ applied to '`Subject`'. We can see, just by looking at it, that it will fail because the text doesn't have a colon in it, but the regex engine won't reach that conclusion until it actually goes through the motions of checking.

So, by the time ⌜`:`⌟ is first checked, the ⌜`\w+`⌟ will have marched to the end of the string. This results in a lot of states—one "skip me" state for each match of ⌜`\w`⌟ by the plus (except the first, since plus requires one match). When then checked at the end of the string, ⌜`:`⌟ fails, so the regex engine backtracks to the most recently saved state:

| at '`Subject`' | matching ⌜`^\w+:`⌟ |
|---|---|

at which point the ⌜`:`⌟ fails again, this time trying to match '`t`'. This backtrack-test-fail cycle happens all the way back to the oldest state:

| at '`Subject`' | matching ⌜`^\w+:`⌟ |
|---|---|

After the attempt from the final state fails, overall failure can finally be announced.

## Quiz Answer

❖ *Answer to the question on page 171.*

**What does ⌜(?>.*?)⌟ match?**

It can never match, anything. At best, it's a fairly complex way to accomplish nothing! ⌜*?⌟ is the lazy ⌜*⌟, and governs a dot, so the first path it attempts is the skip-the-dot path, saving the try-the-dot state for later, if required. But the moment that state has been saved, it's thrown away because matching exits the atomic grouping, so the skip-the-dot path is the only one ever taken. If something is always skipped, it's as if it's not there at all.

All that backtracking is a lot of work that after just a glance we know to be unnecessary. If the colon can't match after the last letter, it certainly can't match one of the letters the ⌜+⌟ is forced to give up!

So, knowing that none of the states left by ⌜\w+⌟, once it's finished, could possibly lead to a match, we can save the regex engine the trouble of checking them: ⌜^(?>\w+):⌟ By adding the atomic grouping, we use our global knowledge of the regex to enhance the local working of ⌜\w+⌟ by having its saved states (which we know to be useless) thrown away. If there *is* a match, the atomic grouping won't have mattered, but if there's not to be a match, having thrown away the useless states lets the regex come to that conclusion more quickly. (An advanced implementation may be able to apply this optimization for you automatically ☞ 251.)

As we'll see in the Chapter 6 (☞ 269), this technique shows a very valuable use of atomic grouping, and I suspect it will become the most common use as well.

## *Possessive Quantifiers,* ?+, *+, ++, and* {m,n}+

Possessive quantifiers are much like greedy quantifiers, but they never give up a partial amount of what they've been able to match. Once a plus, for example, finishes its run, it has created quite a few saved states, as we saw with the ⌜^\w+⌟ example. A *possessive* plus simply throws those states away (or, more likely, doesn't bother creating them in the first place).

As you might guess, possessive quantifiers are closely related to atomic grouping. Something possessive like ⌜\w++⌟ appears to match in the same way as ⌜(?>\w+)⌟; one is just a notational convenience for the other.† With possessive quantifiers, ⌜^(?>\w+):⌟ can be rewritten as ⌜^\w++:⌟, and ⌜(\.\d\d(?>[1-9]?))\d+⌟ can be rewritten as ⌜(\.\d\d[1-9]?+)\d+⌟.

† A smart implementation may be able to make the possessive version a bit more efficient than its atomic-grouping counterpart (☞ 250).

Be sure to understand the difference between ⌜`(?>M)+`⌟ and ⌜`(?>M+)`⌟. The first one throws away unused states created by ⌜`M`⌟, which is not very useful since ⌜`M`⌟ doesn't create any states. The second one throws away unused states created by ⌜`M+`⌟, which certainly can be useful.

When phrased as a comparison between ⌜`(?>M)+`⌟ and ⌜`(?>M+)`⌟, it's perhaps clear that the second one is the one comparable to ⌜`M++`⌟, but when converting something more complex like ⌜`(\\"|[^"])*+`⌟ from possessive quantifiers to atomic grouping, it's tempting to just add '`?>`' to the parentheses that are already there: ⌜`(?>\\"|[^"])*+`⌟. The new expression might happen to achieve your goal, but be clear that is *not* comparable to the original possessive-quantifier version; it's as if changing ⌜`M++`⌟ to ⌜`(?>M)+`⌟. Rather, to be comparable, remove the possessive plus, and then wrap what remains in atomic grouping: ⌜`(?>(\\"|[^"])*)`⌟.

## *The Backtracking of Lookaround*

It might not be apparent at first, but lookaround (introduced in Chapter 2 ☞ 59) is closely related to atomic grouping and possessive quantifiers. There are four types of lookaround: positive and negative flavors of lookahead and lookbehind. They simply test whether their subexpression can and can't match starting at the current location (lookahead), or ending at the current location (lookbehind).

Looking a bit deeper, how does lookaround work in our NFA world of saved states and backtracking? As a subexpression within one of the lookaround constructs is being tested, it's as if it's in its own little world. It saves states as needed, and backtracks as necessary. If the entire subexpression is able to match successfully, what happens? With *positive* lookaround, the construct, as a whole, is considered a success, and with *negative* lookaround, it's considered a failure. In either case, since the only concern is whether there's a match (and we just found out that, yes, there's a match), the "little world" of the match attempt, including any saved states that might have been left over from that attempt, is thrown away.

What about when the subexpression within the lookaround can't match? Since it's being applied in its "own little world," only states created within the current lookaround construct are available. That is, if the regex finds that it needs to backtrack further, beyond where the lookaround construct started, it's found that the current subexpression can not match. For positive lookahead, this means failure, while for negative lookahead, it means success. In either case, there are no saved states left over (had there been, the subexpression match would not have finished), so there's no "little world" left to throw away.

So, we've seen that in all cases, once the lookaround construct has finished, there are no saved states left over from its application. Any states that might have been left over, such as in the case of successful positive lookahead, are thrown away.

Well, where else have we seen states being thrown away? With atomic grouping and possessive quantifiers, of course.

### *Mimicking atomic grouping with positive lookahead*

It's perhaps mostly academic for flavors that support atomic grouping, but can be quite useful for those that don't: *if* you have positive lookahead, and *if* it supports capturing parentheses within the lookahead (most flavors do, but Tcl's lookahead, for example, does not), you can mimic atomic grouping and possessive quantifiers. ⌈`(?>`*regex*`)`⌋ can be mimicked with ⌈`(?=(`*regex*`))\1`⌋. For example, compare ⌈`^(?>\w+):`⌋ with ⌈`^(?=(\w+))\1:`⌋.

The lookahead version has ⌈`\w+`⌋ greedily match as much as it can, capturing an entire word. Because it's within lookahead, the intermediate states are thrown away when it's finished (just as if, incidentally, it had been within atomic grouping). Unlike atomic grouping, the matched word is not included as part of the match (that's the whole point of lookahead), but the word does remain captured. That's a key point because it means that when ⌈`\1`⌋ is applied, it's actually being applied to the very text that filled it, and it's certain to succeed. This extra step of applying ⌈`\1`⌋ is simply to move the regex past the matched word.

This technique is a bit less efficient than real atomic grouping because of the extra time required to rematch the text via ⌈`\1`⌋. But, since states are thrown away, it fails more quickly than a raw ⌈`\w+:`⌋ when the ⌈`:`⌋ can't match.

## *Is Alternation Greedy?*

How alternation works is an important point because it can work in fundamentally different ways with different regex engines. When alternation is reached, any number of the alternatives might be able to match at that point, but which will? Put another way, if more than one can match, which will? If it's always the one that matches the most text, one might say that alternation is greedy. If it's always the shortest amount of text, one might say it's lazy? Which (if either) is it?

Let's look at the Traditional NFA engine used in Perl, Java packages, .NET languages, and many others (☞ 145). When faced with alternation, each alternative is checked in the left-to-right order given in the expression. With the example regex of ⌈`^(Subject|Date):•`⌋, when the ⌈`Subject|Date`⌋ alternation is reached, the first alternative, ⌈`Subject`⌋, is attempted. If it matches, the rest of the regex (the subsequent ⌈`:•`⌋) is given a chance. If it turns out that it can't match, and if other alternatives remain (in this case, ⌈`Date`⌋), the regex engine backtracks to try them. *This is just another case of the regex engine backtracking to a point where untried options are still available.* This continues until an overall match is achieved, or until all options (in this case, all alternatives) are exhausted.

So, with that common Traditional NFA engine, what text is actually matched by ⌈`tour|to|tournament`⌋ when applied to the string '`three•tournaments•won`'? All the alternatives are attempted (and fail) during attempts starting at each character position until the transmission starts the attempt at '`three•tournaments•won`'. This time, the first alternative, ⌈`tour`⌋, matches. Since the alternation is the last thing in the regex, the moment the ⌈`tour`⌋ matches, the whole regex is done. The other alternatives are not even tried again.

So, we see that alternation is neither greedy nor lazy, but *ordered*, at least for a Traditional NFA. This is more powerful than greedy alternation because it allows more control over just how a match is attempted — it allows the regex author to express "try this, then that, and finally try that, until you get a match."

Not all flavors have ordered alternation. DFAs and POSIX NFAs do have greedy alternation, always matching with the alternative that matches the most text (⌈`tournament`⌋ in this case). But, if you're using Perl, a .NET language, virtually any Java regex package, or any other system with a Traditional NFA engine (list ☞ 145), your alternation is *ordered*.

## *Taking Advantage of Ordered Alternation*

Let's revisit the ⌈`(\.\d\d[1-9]?)\d*`⌋ example from page 167. If we realize that ⌈`\.\d\d[1-9]?`⌋, in effect, says "allow either ⌈`\.\d\d`⌋ or ⌈`\.\d\d[1-9]`⌋", we can rewrite the entire expression as ⌈`(\.\d\d|\.\d\d[1-9])\d*`⌋. (There is no compelling reason to make this change — it's merely a handy example.) Is this *really* the same as the original? If alternation is truly greedy, then it is, but the two are quite different with ordered alternation.

Let's consider it as ordered for the moment. The first alternative is selected and tested, and if it matches, control passes to the ⌈`\d*`⌋ that follows the alternation. If there are digits remaining, the ⌈`\d*`⌋ matches them, including any initial non-zero digit that was the root of the original example's problem (if you'll recall the original problem, that's a digit we want to match only within the parentheses, not by the ⌈`\d*`⌋ after the parentheses). Also, realize that if the first alternative can't match, the second alternative will certainly not be able to, as it begins with a copy of the entire first alternative. If the first alternative doesn't match, though, the regex engine nevertheless expends the effort for the futile attempt of the second.

Interestingly, if we swap the alternatives and use ⌈`(\.\d\d[1-9]|\.\d\d)\d*`⌋, we do effectively get a replica of the original greedy ⌈`(\.\d\d[1-9]?)\d*`⌋. The alternation has meaning in this case because if the first alternative fails due to the trailing ⌈`[1-9]`⌋, the second alternative still stands a chance. It's still ordered alternation, but now we've selected the order to result in a greedy-type match.

When first distributing the ⌈[1-9]?⌋ to two alternatives, in placing the shorter one first, we fashioned a non-greedy ⌈?⌋ of sorts. It ends up being meaningless in this particular example because there is nothing that could ever allow the second alternative to match if the first fails. I see this kind of faux-alternation often, and it is invariably a mistake. In one book I've read, ⌈a*((ab)*|b*)⌋ is used as an example in explaining something about regex parentheses. It's a pointless example because the first alternative, ⌈(ab)*⌋, can never fail, so any other alternatives are utterly meaningless. You could add

⌈a*((ab)*|b*|.*|partridge in a pear tree|[a-z])⌋

and it wouldn't change the meaning a bit. The moral is that with ordered alternation, when more than one alternative can potentially match the same text, care must be taken when selecting the order of the alternatives.

### *Ordered alternation pitfalls*

Ordered alternation can be put to your advantage by allowing you to craft just the match you want, but it can also lead to unexpected pitfalls for the unaware. Consider matching a January date of the form '`Jan 31`'. We need something more sophisticated than, say, ⌈Jan [0123][0-9]⌋, as that allows "dates" such as '`Jan 00`', '`Jan 39`', and disallows, '`Jan 7`'.

One way to match the date part is to attack it in sections. To match from the first through the ninth, using ⌈0?[1-9]⌋ allows a leading zero. Adding ⌈[12][0-9]⌋ allows for the tenth through the 29th, and ⌈3[01]⌋ rounds it out. Putting it all together, we get ⌈Jan (0?[1-9]|[12][0-9]|3[01])⌋.

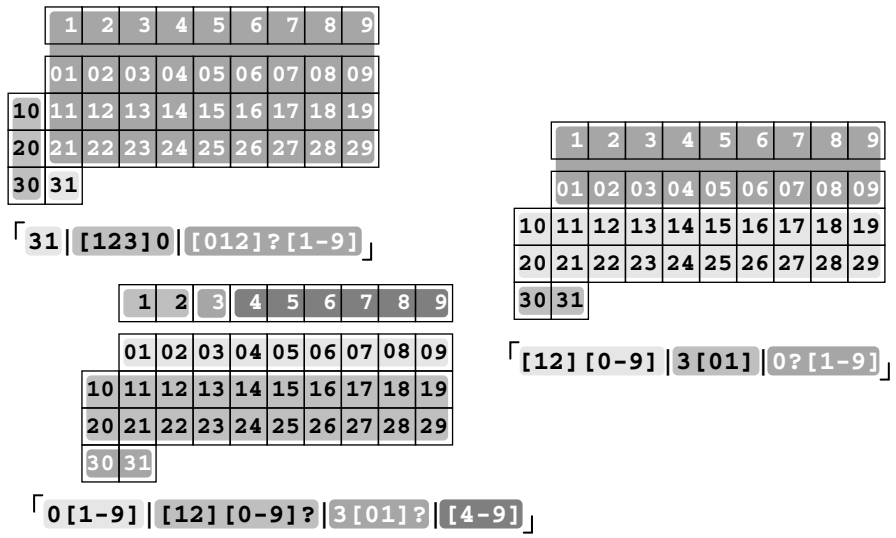
Where do you think this matches in '`Jan 31 is Dad's birthday`'? We want it to match '`Jan 31`', of course, but ordered alternation actually matches only '`Jan 3`'. Surprised? During the match of the first alternative, ⌈0?[1-9]⌋, the leading ⌈0?⌋ fails, but the alternative matches because the subsequent ⌈[1-9]⌋ has no trouble matching the `3`. Since that's the end of the expression, the match is complete.

When the order of the alternatives is adjusted so that the alternative that can potentially match a shorter amount of text is placed last, the problem goes away. This works: ⌈Jan ([12][0-9]|3[01]|0?[1-9])⌋.

Another approach is ⌈Jan (31|[123]0|[012]?[1-9])⌋. Like the first solution, this requires careful arrangement of the alternatives to avoid the problem. Yet, a third approach is ⌈Jan (0[1-9]|[12][0-9]?|3[01]?|[4-9])⌋, which works properly regardless of the ordering. Comparing and contrasting these three expressions can prove quite interesting (an exercise I'll leave for your free time, although the sidebar on the facing page should be helpful).

### *A Few Ways to Slice and Dice a Date*

A few approaches to the date-matching problem posed on page 176. The calendar associated with each regex shows what can be matched by each alternative color-coded within the regex.




# *NFA, DFA, and POSIX*

## *"The Longest-Leftmost"*

Let me repeat what I've said before: when the transmission starts a DFA engine from some particular point in the string, and there exists a match or matches to be found at that position, the DFA finds the longest possible match, period. Since it's the longest from among all possible matches that start equally furthest to the left, it's the "longest-leftmost" match.

### *Really, the longest*

Issues of which match is longest aren't confined to alternation. Consider how an NFA matches the (horribly contrived) 「`one(self)?(selfsufficient)?`」 against the string `oneselfsufficient`. An NFA first matches 「`one`」 and then the greedy 「`(self)?`」, leaving 「`(selfsufficient)?`」 left to try against `sufficient`. It doesn't match, but that's okay since it is optional. So, the Traditional NFA returns `oneselfsufficient` and discards the untried states. (A POSIX NFA is another story that we'll get to shortly.)

On the other hand, a DFA finds the longer oneselfsufficient. An NFA would also find that match if the initial (self)? were to somehow go unmatched, as that would leave (selfsufficient)? then able to match. A Traditional NFA doesn't do that, but the DFA finds it nevertheless, since it's the longest possible match available to the current attempt. It can do this because it keeps track of all matches simultaneously, and knows at all times about all possible matches.

I chose this silly example because it's easy to talk about, but I want you to realize that this issue is important in real life. For example, consider trying to match *continuation lines*. It's not uncommon for a data specification to allow one logical line to extend across multiple real lines if the real lines end with a backslash before the newline. As an example, consider the following:

```
SRC=array.c builtin.c eval.c field.c gawkmisc.c io.c main.c \
        missing.c msg.c node.c re.c version.c
```

You might normally want to use ^\w+=.* to match this kind of "var = value" assignment line, but this regex doesn't consider the continuation lines. (I'm assuming for the example that the tool's dot won't match a newline.) To match continuation lines, you might consider appending (\\\n.*)* to the regex, yielding ^\w+=.*(\\\n.*)*. Ostensibly, this says that any number of additional logical lines are allowed so long as they each follow an escaped newline. This seems reasonable, but it will never work with a traditional NFA. By the time the original .* has reached the newline, *it has already passed the backslash*, and nothing in what was added forces it to backtrack (☞ 152). Yet, a DFA finds the longer multiline match if available, simply because it is, indeed, the longest.

If you have lazy quantifiers available, you might consider using them, such as with ^\w+=.*?(\\\n.*?)*. This allows the escaped newline part to be tested each time before the first dot actually matches anything, so the thought is that the \\ then gets to match the backslash before the newline. Again, this won't work. A lazy quantifier actually ends up matching something optional only when forced to do so, but in this case, everything after the = is optional, so there's nothing to force the lazy quantifiers to match anything. Our lazy example matches only 'SRC=', so it's certainly not the answer.

There are other approaches to solving this problem; we'll continue with this example in the next chapter (☞ 186).

## POSIX and the Longest-Leftmost Rule

The POSIX standard requires that if you have multiple possible matches that start at the same position, the one matching the most text must be the one returned.

The POSIX standard document uses the phrase "longest of the leftmost." It doesn't say you have to use a DFA, so if you want to use an NFA when creating a POSIX

tool, what's a programmer to do? If you want to implement a POSIX NFA, you'd have to find the full `oneselfsufficient` and all the continuation lines, despite these results being "unnatural" to your NFA.

A Traditional NFA engine stops with the first match it finds, but what if it were to continue to try options (states) that might remain? Each time it reached the end of the regex, it would have another plausible match. By the time *all* options are exhausted, it could simply report the longest of the plausible matches it had found. Thus, a POSIX NFA.

An NFA applied to the first example would, in matching `(self)?`, have saved an option noting that it could pick up matching `one(self)?(selfsufficient)?` at `oneselfsufficient`. Even after finding the `oneselfsufficient` that a Traditional NFA stops at, a POSIX NFA continues to exhaustively check the remaining options, eventually realizing that yes, there is a way to match the longer (and in fact, longest) `oneselfsufficient`.

In Chapter 7, we'll see a method to trick Perl into mimicking POSIX semantics, having it report the longest match (☞ 335).

## *Speed and Efficiency*

If efficiency is an issue with a Traditional NFA (and with backtracking, believe me, it can be), it is doubly so with a POSIX NFA since there can be so much more backtracking. A POSIX NFA engine really does have to try every possible permutation of the regex, every time. Examples in Chapter 6 show that poorly written regexes can suffer extremely severe performance penalties.

### *DFA efficiency*

The text-directed DFA is a really fantastic way around all the inefficiency of backtracking. It gets its matching speed by keeping track of all possible ongoing matches at once. How does it achieve this magic?

The DFA engine spends extra time and memory when it first sees the regular expression, before any match attempts are made, to analyze the regular expression more thoroughly (and in a different way) from an NFA. Once it starts actually attempting a match, it has an internal map describing "If I read such-and-such a character now, it will be part of this-and-that possible match." As each character of the string is checked, the engine simply follows the map.

Building that map can sometimes take a fair amount of time and memory, but once it is done for any particular regular expression, the results can be applied to an unlimited amount of text. It's sort of like charging the batteries of your electric car. First, your car sits in the garage for a while, plugged into the wall, but when you actually use it, you get consistent, clean power.

### NFA: Theory Versus Reality

The true mathematical and computational meaning of "NFA" is different from what is commonly called an "NFA regex engine." In theory, NFA and DFA engines should match exactly the same text and have exactly the same features. In practice, the desire for richer, more expressive regular expressions has caused their semantics to diverge. An example is the support for backreferences.

The design of a DFA engine precludes backreferences, but it's a relatively small task to add backreference support to a true (mathematically speaking) NFA engine. In doing so, you create a more powerful tool, but you also make it decidedly *nonregular* (mathematically speaking). What does this mean? At most, that you should probably stop calling it an NFA, and start using the phrase "nonregular expressions," since that describes (mathematically speaking) the new situation. No one has actually done this, so the *name* "NFA" has lingered, even though the implementation is no longer (mathematically speaking) an NFA.

What does all this mean to you, as a user? Absolutely nothing. As a user, you don't care if it's regular, nonregular, unregular, irregular, or incontinent. So long as you know what you can expect from it (something this chapter shows you), you know all you need to care about.

For those wishing to learn more about the theory of regular expressions, the classic computer-science text is chapter 3 of Aho, Sethi, and Ullman's *Compilers—Principles, Techniques, and Tools* (Addison-Wesley, 1986), commonly called "The Dragon Book" due to the cover design. More specifically, this is the "red dragon." The "green dragon" is its predecessor, Aho and Ullman's *Principles of Compiler Design*.

The work done when a regex is first seen (the once-per-regex overhead) is called *compiling the regex*. The map-building is what a DFA does. An NFA also builds an internal representation of the regex, but an NFA's representation is like a mini program that the engine then executes.

## Summary: NFA and DFA in Comparison

Both DFA and NFA engines have their good and bad points.

### *DFA versus NFA: Differences in the pre-use compile*

Before applying a regex to a search, both types of engines compile the regex to an internal form suited to their respective match algorithms. An NFA compile is generally faster, and requires less memory. There's no real difference between a Traditional and POSIX NFA compile.

### *DFA versus NFA: Differences in match speed*

For simple literal-match tests in "normal" situations, both types match at about the same rate. A DFA's match speed is generally unrelated to the particular regex, but an NFA's is directly related.

A Traditional NFA must try every possible permutation of the regex before it can conclude that there's no match. This is why I spend an entire chapter (Chapter 6) on techniques to write NFA expressions that match quickly. As we'll see, an NFA match can sometimes take forever. If it's a Traditional NFA, it can at least stop if and when it finds a match.

A POSIX NFA, on the other hand, must always try every possible permutation of the regex to ensure that it has found the longest possible match, so it generally takes the same (possibly very long) amount of time to complete a successful match as it does to confirm a failure. Writing efficient expressions is doubly important for a POSIX NFA.

In one sense, I speak a bit too strongly, since optimizations can often reduce the work needed to return an answer. We've already seen that an optimized engine doesn't try ⌈^⌋-anchored regexes beyond the start of the string (☞ 149), and we'll see many more optimizations in Chapter 6.

The need for optimizations is less pressing with a DFA since its matching is so fast to begin with, but for the most part, the extra work done during the DFA's pre-use compile affords better optimizations than most NFA engines take the trouble to do.

Modern DFA engines often try to reduce the time and memory used during the compile by postponing some work until a match is attempted. Often, much of the compile-time work goes unused because of the nature of the text actually checked. A fair amount of time and memory can sometimes be saved by postponing the work until it's actually needed during the match. (The technobabble term for this is *lazy evaluation*.) It does, however, create cases where there can be a relationship among the regex, the text being checked, and the match speed.

### *DFA versus NFA: Differences in what is matched*

A DFA (or anything POSIX) finds the longest leftmost match. A Traditional NFA might also, or it might find something else. Any individual engine always treats the same regex/text combination in the same way, so in that sense, it's not "random," but other NFA engines may decide to do slightly different things. Virtually all Traditional NFA engines I've seen work exactly the way I've described here, but it's not something absolutely guaranteed by any standard.

### *DFA versus NFA: Differences in capabilities*

An NFA engine can support many things that a DFA cannot. Among them are:

- Capturing text matched by a parenthesized subexpression. Related features are backreferences and after-match information saying *where* in the matched text each parenthesized subexpression matched.
- Lookaround, and other complex zero-width assertions† (☞ 132).
- Non-greedy quantifiers and ordered alternation. A DFA could easily support a guaranteed shortest overall match (although for whatever reason, this option never seems to be made available to the user), but it cannot implement the local laziness and ordered alternation that we've talked about.
- Possessive quantifiers (☞ 140) and atomic grouping (☞ 137).

### *DFA versus NFA: Differences in ease of implementation*

Although they have limitations, simple versions of DFA and NFA engines are easy enough to understand and to implement. The desire for efficiency (both in time and memory) and enhanced features drives the implementation to greater and greater complexity.

With code length as a metric, consider that the NFA regex support in the Version 7 (January 1979) edition of *ed* was less than 350 lines of C code. (For that matter, the *entire* source for *grep* was a scant 478 lines.) Henry Spencer's 1986 freely available implementation of the Version 8 regex routines was almost 1,900 lines of C, and Tom Lord's 1992 POSIX NFA package `rx` (used in GNU sed, among other tools) is a stunning 9,700 lines.

For DFA implementations, the Version 7 *egrep* regex engine was a bit over 400 lines long, while Henry Spencer's 1992 full-featured POSIX DFA package is over 4,500 lines long.

To provide the best of both worlds, GNU *egrep* Version 2.4.2 utilizes two fully functional engines (about 8,900 lines of code), and Tcl's hybrid DFA/NFA engine (see the sidebar on the facing page) is about 9,500 lines of code.

Some implementations are simple, but that doesn't necessarily mean they are short on features. I once wanted to use regular expressions for some text processing in Pascal. I hadn't used Pascal since college, but it still didn't take long to write a simple NFA regex engine. It didn't have a lot of bells and whistles, and wasn't built for maximum speed, but the flavor was relatively full-featured and was quite useful.

† *lex* has *trailing context*, which is exactly the same thing as zero-width positive lookahead at the end of the regex, but it can't be generalized and put to use for embedded lookahead.

### *DFA Speed with NFA Capabilities: Regex Nirvana?*

I've said several times that a DFA can't provide capturing parentheses or backreferences. This is quite true, but it certainly doesn't preclude hybrid approaches that mix technologies in an attempt to reach regex nirvana. The sidebar on page 180 told how NFAs have diverged from the theoretical straight and narrow in search of more power, and it's only natural that the same happens with DFAs. A DFA's construction makes it more difficult, but that doesn't mean impossible.

GNU *grep* takes a simple but effective approach. It uses a DFA when possible, reverting to an NFA when backreferences are used. GNU awk does something similar—it uses GNU *grep*'s fast shortest-leftmost DFA engine for simple "does it match" checks, and reverts to a different engine for checks where the actual extent of the match must be known. Since that other engine is an NFA, GNU awk can conveniently offer capturing parentheses, and it does via its special `gensub` function.

Tcl's regex engine is a true hybrid, custom built by Henry Spencer (whom you may remember having played an important part in the early development and popularization of regular expressions ☞88). The Tcl engine sometimes appears to be an NFA—it has lookaround, capturing parentheses, backreferences, and lazy quantifiers. Yet, it has true POSIX longest-leftmost match (☞177), and doesn't suffer from some of the NFA problems that we'll see in Chapter 6. It really seems quite wonderful.

Currently, this engine is available only to Tcl, but Henry tells me that it's on his to-do list to break it out into a separate package that can be used by others.

# *Summary*

If you understood everything in this chapter the first time you read it, you probably didn't need to read it in the first place. It's heady stuff, to say the least. It took me quite a while to understand it, and then longer still to *understand* it. I hope this one concise presentation makes it easier for you. I've tried to keep the explanation simple without falling into the trap of oversimplification (an unfortunately all-too-common occurrence which hinders true understanding). This chapter has a lot in it, so I've included a lot of page references in the following summary, for when you'd like to quickly check back on something.

There are two underlying technologies commonly used to implement a regex match engine, "regex-directed NFA" (☞153) and "text-directed DFA" (☞155). The abbreviations are spelled out on page 156.

Combine the two technologies with the POSIX standard (☞ 178), and for practical purposes, there are three types of engines:

- Traditional NFA (gas-guzzling, power-on-demand)
- POSIX NFA (gas-guzzling, standard-compliant)
- DFA (POSIX or not) (electric, steady-as-she-goes)

To get the most out of a utility, you need to understand which type of engine it uses, and craft your regular expressions appropriately. The most common type is the Traditional NFA, followed by the DFA. Table 4-1 (☞ 145) lists a few common tools and their engine types, and the section "Testing the Engine Type" (☞ 146) shows how you can test the type yourself.

One overriding rule regardless of engine type: matches starting sooner take precedence over matches starting later. This is due to how the engine's "transmission" tests the regex at each point in the string (☞ 148).

For the match attempt starting at any given spot:

**DFA Text-Directed Engines**

Find the longest possible match, period. That's it. End of discussion (☞ 177). Consistent, very fast (☞ 179), and boring to talk about.

**NFA Regex-Directed Engines**

Must "work through" a match. The soul of NFA matching is *backtracking* (☞ 157, 162). The metacharacters control the match: the standard quantifiers (star and friends) are *greedy* (☞ 151), while others may be lazy or possessive (☞ 169). Alternation is ordered (☞ 174) in a traditional NFA, but greedy with a POSIX NFA.

**POSIX NFA** Must find the longest match, period. But, it's not boring, as you must worry about efficiency (the subject of Chapter 6).

**Traditional NFA** Is the most expressive type of regex engine, since you can use the regex-directed nature of the engine to craft exactly the match you want.

Understanding the concepts and practices covered in this chapter is the foundation for writing correct and efficient regular expressions, which just happens to be the subject of the next two chapters.