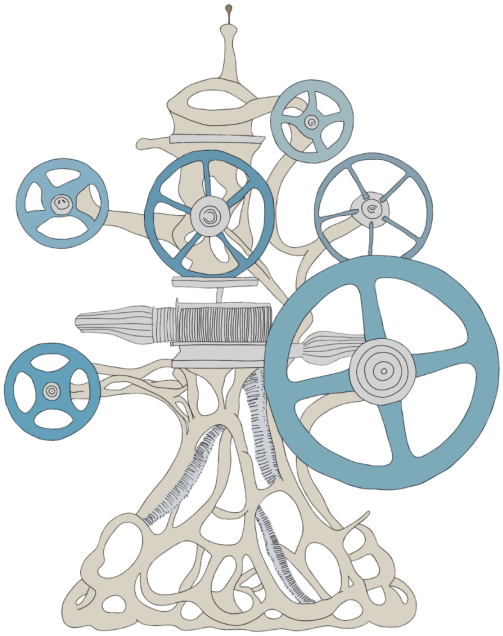
Regular Expressions Machinery

The Illustrated Guide



Staffan Nöteberg

To inquire about booking the author for podcasts, training sessions, and speaking engagements at conferences, please contact him directly at staffan.noteberg@rekursiv.se

PDF ISBN: 978-91-989983-0-6

EPUB ISBN: 978-91-989983-1-3

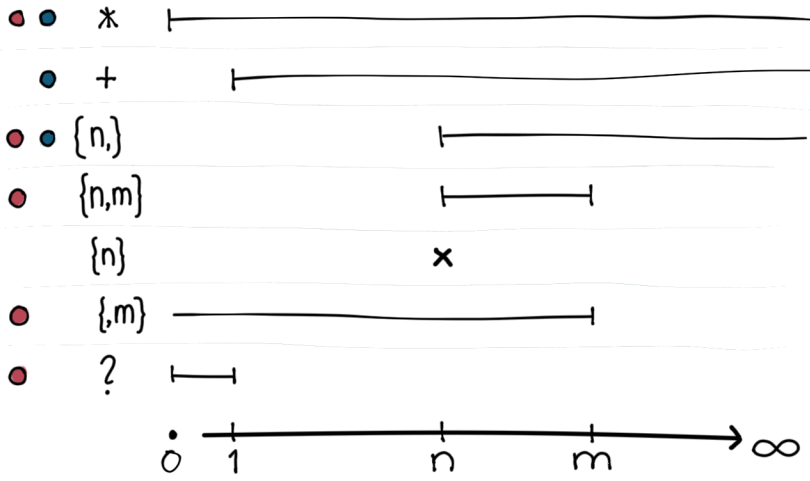
Book Version: V1.0—January, 2025

PART III: Syntactic Sugar, Abstractions, and Extensions

In Part II (Two Operations and One Function), we learned how powerful concatenation, alternation, and the kleene star work together. However, you're probably aware that in modern regex dialects, many other operators—for example, quantifiers, groups, and lookarounds—exist, most of which are abstractions. Without necessarily adding new regex functionality, abstractions help us write regexes without thinking about some of the complexity. Others are just syntactic sugar, making us write easier to read, yet synonymous, regexes. Finally, some extensions cannot be implemented with a finite automaton. We refer to them as path-dependent operations. While consuming the input string, we must know how we arrived in the current state; thus, we need a memory. Under the hood, this memory is implemented as a stack.

NOTE: *Where nothing else is said, examples are written in Ruby. You may install Ruby on your computer and then start IRB (Interactive Ruby Shell) in a terminal to try out the code. You may also run IRB online without any installation.*

Quantifiers



Sometimes, we want to repeat an expression to make it match more than one instance in the same input string. If the number of repetitions is known upfront, that is, it's a fixed number, we may simply repeat the whole expression. For example, the expression `/LaLaLa/` is a repetition of the expression `/La/` three times. Any kind of fixed or variable number of repetitions is possible with the two original operations (concatenation and alternation) and the function (kleene star). However, this might be difficult to read. For example, `/(La|LaLa|LaLaLa|LaLaLaLa)/` expresses *between one and four instances* of `La` in an annoyingly verbose way. This is why quantifier functions exist. These functions don't add any new functionality to regular expressions, but instead support us with crispness.

The two most popular repetition requirements are to match either *at least one* instance, or *at most one* instance.

At least one means one, two, three, or any other positive integer. What will happen if we replace the first instance of `/a*/` with `e` in the input string `caalery`?

```
'caalery'.sub /a*/,'e' #=> "ecaalery"
```

The answer is `ecaalery` because `caalery` starts with (that is, before the `c`) zero instances of `a` and, `/a*/` says that we must match zero to many instances. As I told you before, most regex automata return the leftmost match. What can be more leftmost than before the initial character of a string? However, our intention was to replace repetitions of `a` with an `e`. The handy *positive closure function*, written as a plus sign, `+`, comes to the rescue. The expression `/a+/` should be read as: match at least one `a`.

```
'caalery'.sub /a+/,'e' #=> "celery"
```

Thus, replacing `/a+/` with `e` in `caalery` gives us `celery`—a delicious vegetable.

The *optional quantifier function*—written as a question mark—matches at most one instance of an expression, that is, either zero or one instance. We want to match the word `chickpeas` in singular and plural forms, which is easy. The expression `/chickpeas?/` matches `chickpea`, as well as `chickpeas`.

```
'chickpea chicken chickpeas'.scan /chickpeas?/
   #=> ["chickpea", "chickpeas"]
```

The *optional quantifier function* binds to the `s`, that is, it makes the `s` optional. Why didn't this function bind to the whole `chickpeas` subpattern? You guessed it! It's because the optional quantifier function takes precedence over concatenation, the invisible glue between the literal characters in `chickpeas`.

There's also a *generic quantifier function*. With braces `{}`, we can express any kind of repetition. The normal form has a lowest and a highest acceptable number of matches, and these two numbers are separated by a comma. The expression `/(La){1, 4}/` matches at least one and at most four `La`. Let's see what we get when we replace matchings with `oh` in the input `LaLaLaLaLaLa`.

```
'LaLaLaLaLaLa'.sub /(La){1,4}/, 'oh'
   #=> "ohLaLa"
```

The default value for the first argument is zero. Thus, the expression `/(La){, 4}/` matches at most four and at least zero `La` uses.

```
'ohLaLaLaLaLaLa'.sub /(La){,4}/, 'oh'
   #=> "ohohLaLaLaLaLaLa"
```

What was that? Another oh was added, but no `La` was removed because most regex automata prefer to return the match that's leftmost in the input string. We explicitly said "at least zero instances," didn't we? Thus, we matched zero instances right at the start of the string and replaced that empty string with `oh`. Perhaps it's more clear in this example.

```
'OhLaLaLaLaLaLa'.sub /(La){,4}/, 'oh'
   #=> "ohOhLaLaLaLaLaLa"
```

No `La` actually was needed to achieve zero matches. I also said before that quantifiers are greedy. Why not match four instances of `La` rather than none? Is preferring leftmost and being greedy a contradiction? Not at all. However, our leftmost strategy is ahead of our greedy strategy (or reluctant when this is our flavor). First, we found a match farest left, then we decided to be greedy right there.

If we keep the first generic quantifier argument, then we can omit the second, in which case, there's no upper limit. The expression `/(La){1,}/` matches at least one `La` and is equivalent to `/La+/`.

```
'LaLaLaLaLaLa'.sub /(La){1,}/, 'oh'
   #=> "oh"
```

The final version of the *generic quantifier function* has no comma and contains only one number. The expression `/(La){2}/` matches exactly two instances of `La`—nothing more and nothing less.

```
'LaLaLaLaLaLa'.sub /(La){2}/, 'oh'
   #=> "ohLaLaLaLa"
```

Quantifiers are unary, left-associative, and take precedence over alternation and concatenation. Consider why the following are true.

- Concatenation: `/ab{1, 2}/` equals `/a(b{1, 2})/`
- Concatenation: `/a{1, 2}b/` equals `/(a{1, 2})b/`
- Alternation: `/a|b{1, 2}/` equals `/a|(b{1, 2})/`
- Alternation: `/a{1, 2}|b/` equals `/(a{1, 2})|b/`

The horizontal axis in the previous image states how many instances a quantifier matches. It starts with zero, which matches the empty string, and ends in infinity, that is, there's no limit to how many times an instance can be repeated and still match our expression.

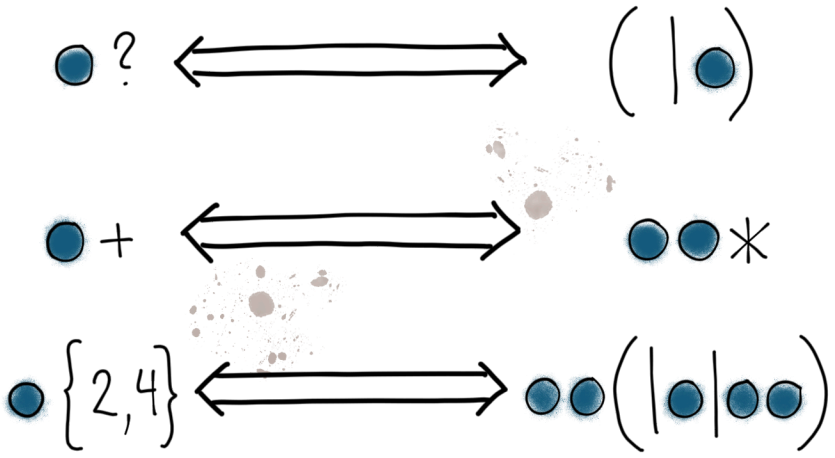
The first column (red dots) indicates that *kleene star function* `*`, *optional quantifier function* `?`, and *generic quantifier function* with first argument zero `{0, m}` or omitted `{, m}` may match zero instances.

The second column (green dots) indicates that *kleene star function* `*`, *positive closure function* `+`, and *generic quantifier function* with second argument omitted `{n,}` have no upper limit when it comes to the number of possible matches.

Two Takeaways

- Quantifiers in regexes help specify how many times a preceding element should be matched, making expressions more compact and readable.
- Common quantifiers include the positive closure `+`, which matches one or more repetitions, and the optional quantifier `?`, which matches zero or one repetition.

Quantifier Equations



The notation of regular expressions, as Stephen Kleene defined it, has only two operations—*concatenation* and *alternation*—and a function: *kleene star*. This is enough to define shorthand functions, like some quantifiers. The shorthand provides us with a syntax that makes our expressions more readable and maintainable, though they don't add any new functionalities. Everything they can do also can be done with the original two operations (concatenation and alternation) and the function (kleene star). However, the latter would be more verbose.

To convince you that a quantifier really is just a shorthand, I'll give you what I call *regex equations*. An *equation* is a mathematical statement that says two expressions are equal.

Let's start with the *optional quantifier function*, written as a question mark. Like all quantifiers, it binds the expression to the left. In this image, you can see that saying that the green dot is optional is the equivalent of using alternation to say either we match nothing or else we match one green dot.

The *positive closure function* is written as a plus sign. Positive closure means we must match at least one instance of the expression to the left. In the second equation in the previous image, I claim that a green dot's positive closure is equivalent to one mandatory green point concatenated with the closure we get by applying the *kleene star*, written as an asterisk, to another green dot. One green dot is mandatory, then follows zero-to-many green dots. This wasn't a very provocative proposition, was it?

In regex, we use braces `{}` as another shorthand quantifier. They hold a pair of arguments that define the minimum and maximum number of repetitions of the expression to the left. The third equation in the previous image states that it's equivalent to (a) repeat the green dot at least two times and at most four times, and (b) match two green dots concatenated with the alternation zero, one, or two green dots.

> Do you ever say *curly brackets* when referring to "`{`" and "`}`"? Programmers in the US often call them *braces,* and in the UK, they sometimes are called *squiggly brackets*. India has the more poetic name *flower brackets*, and in Sweden, we say *gull wings*.

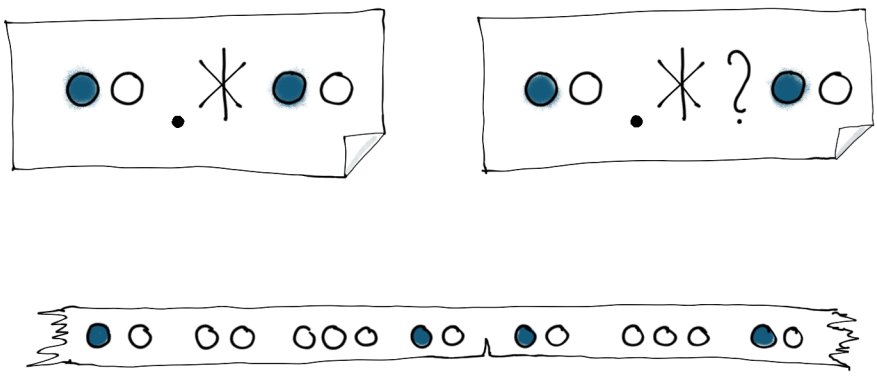
Here are some more equations. Inspect them to ensure that you agree with me that they're true.

- `a{3}` equals `a{3, 3}`
- `a*` equals `a{0, }`
- `a+` equals `a{1, }`
- `a?` equals `a{0, 1}`
- `(a*)*` equals `a*`
- `(a+)+` equals `a+`
- `(a?)?` equals `a?`
- `(a*)+` equals `(a+)*` equals `a*`
- `(a*)?` equals `(a?)*` equals `a*`
- `(a+)?` equals `(a?)+` equals `a*`
- `a*` equals `(a+|)`
- `a?` equals `(a|)?` equals `(a|)`
- `(a|)*` equals `a*`

Two Takeaways

- Quantifiers can be expressed using the basic operations (concatenation and alternation) and the function (kleene star) of regexes, but they offer a more concise syntax.
- For instance, `a?` (match zero or one instance of the expression to the left) is equivalent to `a|`, and `a+` is equivalent to `aa*`.

Reluctant Quantifiers



Greedy (left) and reluctant (right) kleene star.

Because of backtracking, we sometimes might match more than we hoped for. The task below is to catch all the div tags in an HTML document and put them in a vector. Our naïve solution provides the wrong answer.

```
'<div>a</div><span>c</span><div>b</div>'.scan /<div>.*<\/div>/
   #=> ["<div>a</div><span>c</span><div>b</div>"]
```

Do you remember from Part I what *backtracking* is? No problem if you don't. Let's explore a regex version:

Regex quantifiers are naturally greedy, that is, they consume as much as they can. The *period* symbol in the idiom `/.*/` matches anything except newline (more on this later in the book), and the *kleene star* `*` in that expression means that this anything is repeated as many times as possible. In its first attempt, `/.*/` matches `a</div><span>c</span><div>b</div>`. Unfortunately, this means that the last part of the regex `/<\/div>/` is starving, which makes `/.*/` very sad. The latter then backtracks—that is, un-consumes—the last part of the input string, character by

character, until the whole expression matches. Considering that the string ends with a `</div>`, the substring `a</div><span>c</span><div>b` will be consumed by `/.*/`. Problems may arise from greed.

Many stories tell tales of greedy people who claim more than they need. As Louis Blanc wrote in 1840 in *The Organization of Work*: *"From each according to his abilities, to each according to his needs."* We tend to forget that the kleene star `*` and the period symbol—that is, the idiom `/.*/`—in the previous example *need* to consume more than the substring `<div>a</div>`.

> Alexander Pushkin describes in *The Tale of the Fisherman and the Fish* how a magic fish promises to fulfill whatever the fisherman wishes. The fisherman's wife eventually starts asking the fish for bigger and better things—and she gets them—until she eventually wants to become Ruler of the Sea. The magic fish then takes back everything he gave the fisherman's wife.

Quantifiers in regex are naturally greedy. They attempt to consume as much of the input string as possible. The good news is that regex provides an alternative: the reluctant (sometimes called lazy) quantifier modifier. You guessed it: The reluctant modifier makes the quantifiers attempt to consume as little as possible of the input string. The verb *attempt* is important here. After the attempt to consume as much (greedy) or little (reluctant) as possible, there might be subexpressions further to the right that can't match the remaining input string, that is, the entire expression can't be matched. If this happens, the automaton will backtrack, and our quantifier will make a new attempt. This time, it will—contrary to its inherent ideology—consume one less (greedy) or more (reluctant) character compared with its first uncompromising attempt. The method is repeated until we either can match everything or confirm that there's no possible way to create a match.

Does it matter whether we use the greedy or reluctant approach? Well, the strings that can be matched with greedy are also matched with reluctant, and vice versa. However, when multiple matches are possible, greedy tracking sometimes will choose a different match than reluctant tracking. These two approaches also differ in performance. In some cases, reluctant is faster, while in other cases, greedy is. It's a

question of how many backtrackings we must make. For finite automata, Mr. Performance shudders at the mere mention of his foe: Mr. Backtracking.

No special symbol for reluctant quantifiers exists. Instead, we have a modifier symbol—written as a question mark—that may be added to the right of any quantifier. While `*` says "repeat as many times as possible," `*?` says "repeat as few times as possible." What a great duo! Similarly, we may modify any other quantifier.

- **Positive closure** function with reluctant modifier: at least one, as few as possible: `+?`
- **Optional quantifier** function with reluctant modifier: zero or one, preferably zero: `??`
- **Generic quantifier** function with reluctant modifier: between three and five and as few as possible: `{3, 5}?`

Note that the question mark that modifies quantifiers isn't the same question mark that's used as the *optional quantifier function*. They may even be used in conjunction with each other, as `??` indicates above. It's context dependent whether a question mark represents the *reluctant modifier* or the *optional quantifier function*. Of course, in some contexts, the question mark also can be a literal, that is, we want to match a question mark in the input string.

Here are some quantifier examples with and without the reluctant modifier. The first one is an improved solution to the div tag problem.

```
'<div>a</div><span>c</span><div>b</div>' \
   .scan /<div>.*?<\/div>/
   #=> ["<div>a</div>", "<div>b</div>"]
'aa'.match /a?/ #=> #<MatchData "a">
'aa'.match /a??/ #=> #<MatchData "">
'aaaaa'.match /a{2,4}/
   #=> #<MatchData "aaaa">
   # at least 2, at most 4, as much as possible
'aaaaa'.match /a{2,4}?/
   #=> #<MatchData "aa">
   # at least 2, at most 4, as little as possible
```

```
'aaaaa'.match /a{2,}/ #=> #<MatchData "aaaaa">
'aaaaa'.match /a{2,}?/ #=> #<MatchData "aa">
'aaaaa'.match /a{,4}/ #=> #<MatchData "aaaa">
'aaaaa'.match /a{,4}?/ #=> #<MatchData "">
```

Two Takeaways

- While quantifiers are typically greedy (matching as many repetitions as possible), they can be made reluctant by adding a question mark `?` to match as few repetitions as possible.
- Reluctant quantifiers can affect performance and may lead to different matches when multiple matches are possible.

Get a Grip on the Regex Machinery

To effectively use regular expressions, you need to understand how the machinery works under the hood. It's about taking control of the search process, controlling how the pattern is matched, and thus getting both faster and more accurate results.

In this illustrated guide, you gain precisely that understanding.

You can even get started without any prior knowledge of regular expressions. Before you know it, advanced tools like reluctant, lookbehind and nondeterministic finite automata will be at your fingertips as you write efficient and elegant regexes with ease.

This book presents complex concepts in a simple and visual way, with clear examples and practical applications. Whether you are a programmer, data analyst, or just want to get better at text processing, this book will take your knowledge to the next level.

Staffan Nöteberg is a bestselling author passionate about helping people become more efficient and focused. He is the author of these popular books:

- Pomodoro Technique Illustrated: The Easy Way to Do More in Less Time
- Monotasking: How to Focus Your Mind and Be More Productive
- Guiding Star OKRs: A New Approach to Setting and Achieving Goals

With a background in software development and an interest in productivity, Staffan combines his expertise with an ability to explain complex topics in an easy-to-understand way.

PDF ISBN: 978-91-989983-0-6

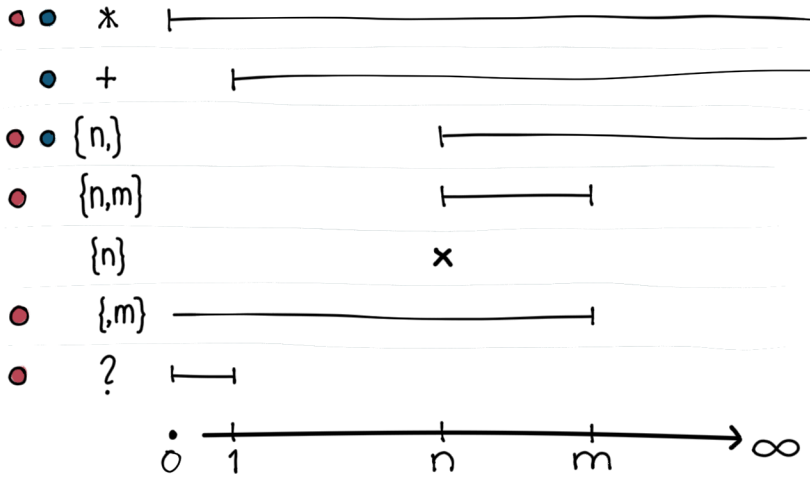
EPUB ISBN: 978-91-989983-1-3

PART III: Syntactic Sugar, Abstractions, and Extensions

In Part II (Two Operations and One Function), we learned how powerful concatenation, alternation, and the kleene star work together. However, you're probably aware that in modern regex dialects, many other operators—for example, quantifiers, groups, and lookarounds—exist, most of which are abstractions. Without necessarily adding new regex functionality, abstractions help us write regexes without thinking about some of the complexity. Others are just syntactic sugar, making us write easier to read, yet synonymous, regexes. Finally, some extensions cannot be implemented with a finite automaton. We refer to them as path-dependent operations. While consuming the input string, we must know how we arrived in the current state; thus, we need a memory. Under the hood, this memory is implemented as a stack.

NOTE: *Where nothing else is said, examples are written in Ruby. You may install Ruby on your computer and then start IRB (Interactive Ruby Shell) in a terminal to try out the code. You may also run IRB online without any installation.*

Quantifiers



Sometimes, we want to repeat an expression to make it match more than one instance in the same input string. If the number of repetitions is known upfront, that is, it's a fixed number, we may simply repeat the whole expression. For example, the expression `/LaLaLa/` is a repetition of the expression `/La/` three times. Any kind of fixed or variable number of repetitions is possible with the two original operations (concatenation and alternation) and the function (kleene star). However, this might be difficult to read. For example, `/(La|LaLa|LaLaLa|LaLaLaLa)/` expresses *between one and four instances* of `La` in an annoyingly verbose way. This is why quantifier functions exist. These functions don't add any new functionality to regular expressions, but instead support us with crispness.

The two most popular repetition requirements are to match either *at least one* instance, or *at most one* instance.

At least one means one, two, three, or any other positive integer. What will happen if we replace the first instance of `/a*/` with `e` in the input string `caalery`?

```
'caalery'.sub /a*/,'e' #=> "ecaalery"
```

The answer is `ecaalery` because `caalery` starts with (that is, before the `c`) zero instances of `a` and, `/a*/` says that we must match zero to many instances. As I told you before, most regex automata return the leftmost match. What can be more leftmost than before the initial character of a string? However, our intention was to replace repetitions of `a` with an `e`. The handy *positive closure function*, written as a plus sign, `+`, comes to the rescue. The expression `/a+/` should be read as: match at least one `a`.

```
'caalery'.sub /a+/,'e' #=> "celery"
```

Thus, replacing `/a+/` with `e` in `caalery` gives us `celery`—a delicious vegetable.

The *optional quantifier function*—written as a question mark—matches at most one instance of an expression, that is, either zero or one instance. We want to match the word `chickpeas` in singular and plural forms, which is easy. The expression `/chickpeas?/` matches `chickpea`, as well as `chickpeas`.

```
'chickpea chicken chickpeas'.scan /chickpeas?/
   #=> ["chickpea", "chickpeas"]
```

The *optional quantifier function* binds to the `s`, that is, it makes the `s` optional. Why didn't this function bind to the whole `chickpeas` subpattern? You guessed it! It's because the optional quantifier function takes precedence over concatenation, the invisible glue between the literal characters in `chickpeas`.

There's also a *generic quantifier function*. With braces `{}`, we can express any kind of repetition. The normal form has a lowest and a highest acceptable number of matches, and these two numbers are separated by a comma. The expression `/(La){1, 4}/` matches at least one and at most four `La`. Let's see what we get when we replace matchings with `oh` in the input `LaLaLaLaLaLa`.

```
'LaLaLaLaLaLa'.sub /(La){1,4}/, 'oh'
   #=> "ohLaLa"
```

The default value for the first argument is zero. Thus, the expression `/(La){, 4}/` matches at most four and at least zero `La` uses.

```
'ohLaLaLaLaLaLa'.sub /(La){,4}/, 'oh'
   #=> "ohohLaLaLaLaLaLa"
```

What was that? Another oh was added, but no `La` was removed because most regex automata prefer to return the match that's leftmost in the input string. We explicitly said "at least zero instances," didn't we? Thus, we matched zero instances right at the start of the string and replaced that empty string with `oh`. Perhaps it's more clear in this example.

```
'OhLaLaLaLaLaLa'.sub /(La){,4}/, 'oh'
   #=> "ohOhLaLaLaLaLaLa"
```

No `La` actually was needed to achieve zero matches. I also said before that quantifiers are greedy. Why not match four instances of `La` rather than none? Is preferring leftmost and being greedy a contradiction? Not at all. However, our leftmost strategy is ahead of our greedy strategy (or reluctant when this is our flavor). First, we found a match farest left, then we decided to be greedy right there.

If we keep the first generic quantifier argument, then we can omit the second, in which case, there's no upper limit. The expression `/(La){1,}/` matches at least one `La` and is equivalent to `/La+/`.

```
'LaLaLaLaLaLa'.sub /(La){1,}/, 'oh'
   #=> "oh"
```

The final version of the *generic quantifier function* has no comma and contains only one number. The expression `/(La){2}/` matches exactly two instances of `La`—nothing more and nothing less.

```
'LaLaLaLaLaLa'.sub /(La){2}/, 'oh'
   #=> "ohLaLaLaLa"
```

Quantifiers are unary, left-associative, and take precedence over alternation and concatenation. Consider why the following are true.

- Concatenation: `/ab{1, 2}/` equals `/a(b{1, 2})/`
- Concatenation: `/a{1, 2}b/` equals `/(a{1, 2})b/`
- Alternation: `/a|b{1, 2}/` equals `/a|(b{1, 2})/`
- Alternation: `/a{1, 2}|b/` equals `/(a{1, 2})|b/`

The horizontal axis in the previous image states how many instances a quantifier matches. It starts with zero, which matches the empty string, and ends in infinity, that is, there's no limit to how many times an instance can be repeated and still match our expression.

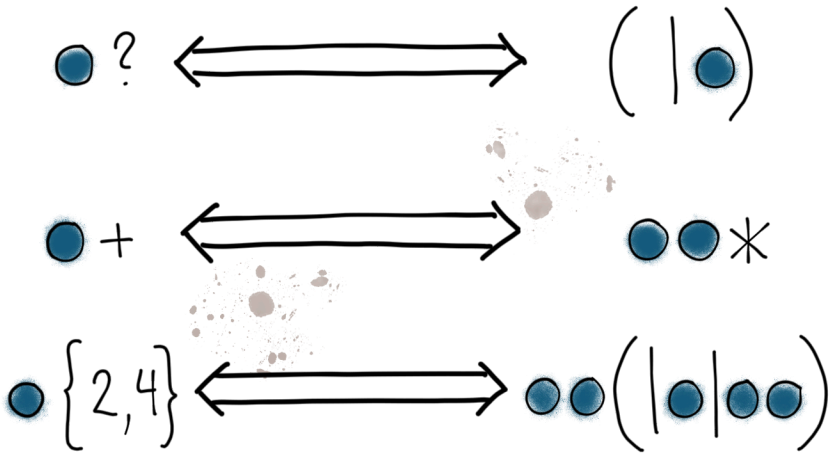
The first column (red dots) indicates that *kleene star function* `*`, *optional quantifier function* `?`, and *generic quantifier function* with first argument zero `{0, m}` or omitted `{, m}` may match zero instances.

The second column (green dots) indicates that *kleene star function* `*`, *positive closure function* `+`, and *generic quantifier function* with second argument omitted `{n,}` have no upper limit when it comes to the number of possible matches.

Two Takeaways

- Quantifiers in regexes help specify how many times a preceding element should be matched, making expressions more compact and readable.
- Common quantifiers include the positive closure `+`, which matches one or more repetitions, and the optional quantifier `?`, which matches zero or one repetition.

Quantifier Equations



The notation of regular expressions, as Stephen Kleene defined it, has only two operations—*concatenation* and *alternation*—and a function: *kleene star*. This is enough to define shorthand functions, like some quantifiers. The shorthand provides us with a syntax that makes our expressions more readable and maintainable, though they don't add any new functionalities. Everything they can do also can be done with the original two operations (concatenation and alternation) and the function (kleene star). However, the latter would be more verbose.

To convince you that a quantifier really is just a shorthand, I'll give you what I call *regex equations*. An *equation* is a mathematical statement that says two expressions are equal.

Let's start with the *optional quantifier function*, written as a question mark. Like all quantifiers, it binds the expression to the left. In this image, you can see that saying that the green dot is optional is the equivalent of using alternation to say either we match nothing or else we match one green dot.

The *positive closure function* is written as a plus sign. Positive closure means we must match at least one instance of the expression to the left. In the second equation in the previous image, I claim that a green dot's positive closure is equivalent to one mandatory green point concatenated with the closure we get by applying the *kleene star*, written as an asterisk, to another green dot. One green dot is mandatory, then follows zero-to-many green dots. This wasn't a very provocative proposition, was it?

In regex, we use braces `{}` as another shorthand quantifier. They hold a pair of arguments that define the minimum and maximum number of repetitions of the expression to the left. The third equation in the previous image states that it's equivalent to (a) repeat the green dot at least two times and at most four times, and (b) match two green dots concatenated with the alternation zero, one, or two green dots.

> Do you ever say *curly brackets* when referring to "`{`" and "`}`"? Programmers in the US often call them *braces,* and in the UK, they sometimes are called *squiggly brackets*. India has the more poetic name *flower brackets*, and in Sweden, we say *gull wings*.

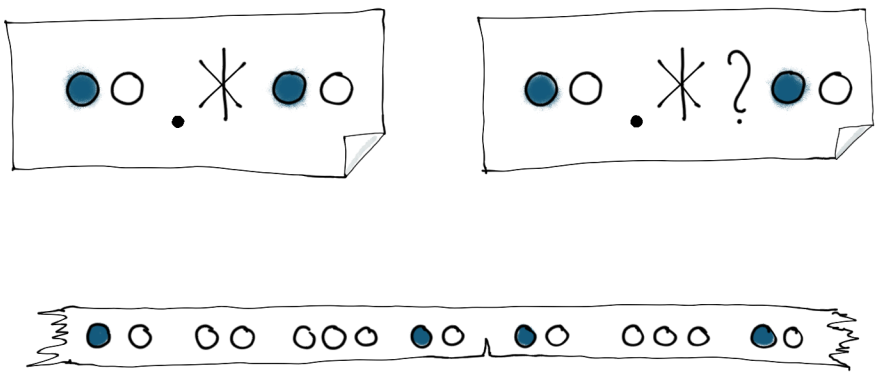
Here are some more equations. Inspect them to ensure that you agree with me that they're true.

- `a{3}` equals `a{3, 3}`
- `a*` equals `a{0, }`
- `a+` equals `a{1, }`
- `a?` equals `a{0, 1}`
- `(a*)*` equals `a*`
- `(a+)+` equals `a+`
- `(a?)?` equals `a?`
- `(a*)+` equals `(a+)*` equals `a*`
- `(a*)?` equals `(a?)*` equals `a*`
- `(a+)?` equals `(a?)+` equals `a*`
- `a*` equals `(a+|)`
- `a?` equals `(a|)?` equals `(a|)`
- `(a|)*` equals `a*`

Two Takeaways

- Quantifiers can be expressed using the basic operations (concatenation and alternation) and the function (kleene star) of regexes, but they offer a more concise syntax.
- For instance, `a?` (match zero or one instance of the expression to the left) is equivalent to `a|`, and `a+` is equivalent to `aa*`.

Reluctant Quantifiers



Greedy (left) and reluctant (right) kleene star.

Because of backtracking, we sometimes might match more than we hoped for. The task below is to catch all the div tags in an HTML document and put them in a vector. Our naïve solution provides the wrong answer.

```
'<div>a</div><span>c</span><div>b</div>'.scan /<div>.*<\/div>/
   #=> ["<div>a</div><span>c</span><div>b</div>"]
```

Do you remember from Part I what *backtracking* is? No problem if you don't. Let's explore a regex version:

Regex quantifiers are naturally greedy, that is, they consume as much as they can. The *period* symbol in the idiom `/.*/` matches anything except newline (more on this later in the book), and the *kleene star* `*` in that expression means that this anything is repeated as many times as possible. In its first attempt, `/.*/` matches `a</div><span>c</span><div>b</div>`. Unfortunately, this means that the last part of the regex `/<\/div>/` is starving, which makes `/.*/` very sad. The latter then backtracks—that is, un-consumes—the last part of the input string, character by

character, until the whole expression matches. Considering that the string ends with a `</div>`, the substring `a</div><span>c</span><div>b` will be consumed by `/.*/`. Problems may arise from greed.

Many stories tell tales of greedy people who claim more than they need. As Louis Blanc wrote in 1840 in *The Organization of Work*: *"From each according to his abilities, to each according to his needs."* We tend to forget that the kleene star `*` and the period symbol—that is, the idiom `/.*/`—in the previous example *need* to consume more than the substring `<div>a</div>`.

> Alexander Pushkin describes in *The Tale of the Fisherman and the Fish* how a magic fish promises to fulfill whatever the fisherman wishes. The fisherman's wife eventually starts asking the fish for bigger and better things—and she gets them—until she eventually wants to become Ruler of the Sea. The magic fish then takes back everything he gave the fisherman's wife.

Quantifiers in regex are naturally greedy. They attempt to consume as much of the input string as possible. The good news is that regex provides an alternative: the reluctant (sometimes called lazy) quantifier modifier. You guessed it: The reluctant modifier makes the quantifiers attempt to consume as little as possible of the input string. The verb *attempt* is important here. After the attempt to consume as much (greedy) or little (reluctant) as possible, there might be subexpressions further to the right that can't match the remaining input string, that is, the entire expression can't be matched. If this happens, the automaton will backtrack, and our quantifier will make a new attempt. This time, it will—contrary to its inherent ideology—consume one less (greedy) or more (reluctant) character compared with its first uncompromising attempt. The method is repeated until we either can match everything or confirm that there's no possible way to create a match.

Does it matter whether we use the greedy or reluctant approach? Well, the strings that can be matched with greedy are also matched with reluctant, and vice versa. However, when multiple matches are possible, greedy tracking sometimes will choose a different match than reluctant tracking. These two approaches also differ in performance. In some cases, reluctant is faster, while in other cases, greedy is. It's a

question of how many backtrackings we must make. For finite automata, Mr. Performance shudders at the mere mention of his foe: Mr. Backtracking.

No special symbol for reluctant quantifiers exists. Instead, we have a modifier symbol—written as a question mark—that may be added to the right of any quantifier. While `*` says "repeat as many times as possible," `*?` says "repeat as few times as possible." What a great duo! Similarly, we may modify any other quantifier.

- **Positive closure** function with reluctant modifier: at least one, as few as possible: `+?`
- **Optional quantifier** function with reluctant modifier: zero or one, preferably zero: `??`
- **Generic quantifier** function with reluctant modifier: between three and five and as few as possible: `{3, 5}?`

Note that the question mark that modifies quantifiers isn't the same question mark that's used as the *optional quantifier function*. They may even be used in conjunction with each other, as `??` indicates above. It's context dependent whether a question mark represents the *reluctant modifier* or the *optional quantifier function*. Of course, in some contexts, the question mark also can be a literal, that is, we want to match a question mark in the input string.

Here are some quantifier examples with and without the reluctant modifier. The first one is an improved solution to the div tag problem.

```
'<div>a</div><span>c</span><div>b</div>' \
   .scan /<div>.*?<\/div>/
   #=> ["<div>a</div>", "<div>b</div>"]
'aa'.match /a?/ #=> #<MatchData "a">
'aa'.match /a??/ #=> #<MatchData "">
'aaaaa'.match /a{2,4}/
   #=> #<MatchData "aaaa">
   # at least 2, at most 4, as much as possible
'aaaaa'.match /a{2,4}?/
   #=> #<MatchData "aa">
   # at least 2, at most 4, as little as possible
```

```
'aaaaa'.match /a{2,}/ #=> #<MatchData "aaaaa">
'aaaaa'.match /a{2,}?/ #=> #<MatchData "aa">
'aaaaa'.match /a{,4}/ #=> #<MatchData "aaaa">
'aaaaa'.match /a{,4}?/ #=> #<MatchData "">
```

Two Takeaways

- While quantifiers are typically greedy (matching as many repetitions as possible), they can be made reluctant by adding a question mark `?` to match as few repetitions as possible.
- Reluctant quantifiers can affect performance and may lead to different matches when multiple matches are possible.