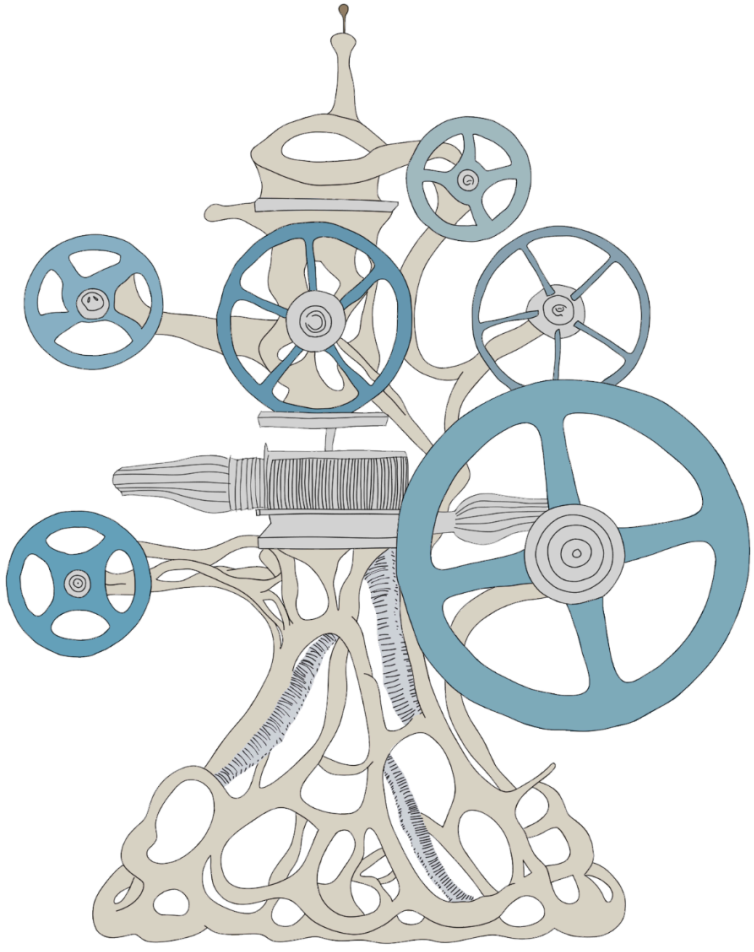
Regular Expressions Machinery

The Illustrated Guide



Staffan Nöteberg

Regular Expressions Machinery

The Illustrated Guide

Staffan Nöteberg

Rekursiv AB

Stockholm, Sweden

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

Contents

Introduction

> *"I define UNIX as 30 definitions of regular expressions living under one roof."*

Turing Award winner Donald Knuth said that, and it doesn't stop there. All mainstream programming languages host regular expressions as a domain-specific language (DSL) and there are numerous books about them. However, the book you're holding in your hand right now is very different.

Why You Might Find This Book Valuable

This book is written for those who want to deeply understand regular expressions. Maybe you've come across them in programming languages or tools but never really understood how they work under the hood. Even though the book is based on advanced mathematics, you don't need any prior knowledge.

Here are some examples of people who could benefit from the book:

- **Application developers:** If you regularly use regular expressions in languages like Java, C#, Python, or Ruby, you'll be able to write more efficient, accurate, and readable expressions after reading this book.
- **Data analysts:** If you work with text analysis or data mining, you can use regular expressions to identify patterns and extract information from large datasets.
- **Software developers focused on compilers and DSLs:** For those who develop compilers or domain-specific languages, this book provides insights into how regular expressions can be used to define and interpret languages.
- **System and network administrators:** If you administer systems or networks, you can use regular expressions to analyze log files, search for specific patterns, and automate tasks.

- **Computer science students:** If you're taking a course on automata theory or compiler design, this book will give you a concrete and illustrated introduction to regular expressions and the underlying automata theory.

By reading this book, you will:

- **Understand the theoretical basis of regular expressions:** You'll learn about finite automata, states, transitions, and how they relate to regular expressions.
- **Learn to construct and interpret complex regular expressions:** The book covers advanced concepts such as greediness, backtracking, quantifiers, and lookarounds.
- **Write more efficient and readable expressions:** You'll learn to avoid common pitfalls and write expressions that are easy to understand and maintain.
- **Use regular expressions in different contexts:** The book provides examples of how regular expressions can be used to solve problems in various domains.

Unlike many other books that focus solely on syntax, this book starts with a deep dive into the underlying automata theory. This provides you with a deeper understanding of how regular expressions work on a fundamental level. Clear illustrations and straightforward explanations make university-level mathematics accessible for everyone.

The goal of the book is to give you a deep understanding of regular expressions so that you can use them with confidence and efficiency in your work.

A Short Map of the Book

The book is divided into four parts, and my strong recommendation is that you read them in order.

Part I: The Automaton: We begin by looking at the theoretical foundation of regular expressions: finite automata. We will both delve into states and transitions, and also

learn what deterministic and non-deterministic finite automata are, and why it is important when you develop your own regular expressions.

Part II: Two Operations and One Function: Here we go through the two operations and the only function needed to create regular expressions. Spoiler alert! The operations are concatenation and alternation. The function is the Kleene star. Can that really be enough to describe all regular expressions? You'll know when you have finished this PART II.

Part III: Syntactic Sugar, Abstractions, and Extensions: The third part is more like a traditional regex book. We look at both the basic and the more advanced features of regular expressions, such as quantifiers, groups, and lookarounds. We also examine some of the most common pitfalls.

Part IV: Test-Driven Regex Development: In the final part, we explore the concept of test-driven development (TDD) and how you can leverage it to write better regexes. By writing tests before you write the actual regexes, you can ensure that your regexes are correct and easy to maintain. We will also look at how to use TDD to refactor legacy regexes.

Again, I strongly recommend reading the book in the presented order. After all, that's why you're here: Your goal is to understand the regex machinery first and then apply that understanding to the syntax in Part III.

Time to lift the hood and look inside the engine. The exciting story of the automaton begins on the next page.

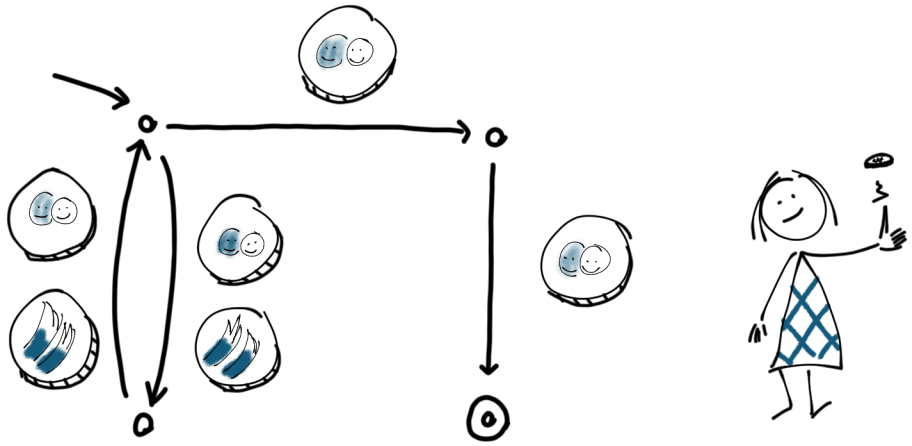
PART I: The Automaton

As you awake one morning from uneasy dreams, you find yourself transformed in your bed into a gigantic email spammer. To send unsolicited electronic mails, you write a regular expressions-based spider .... No, wait! We're going vastly too fast now. Regular expression-based spiders are still a few sections ahead.

To truly understand regular expressions, we must start under the hood. We must climb down into the automaton and watch the oily gears interact with each other and hear the plops and clicks.

The simplest kind of regular expression core is a finite automaton (FA), which is a part of a young science that took shape in the 20th century. Don't get horrified at the thought of states, symbols, and transition tables. We'll discover this in baby steps, and the return on investment will be generous.

Transition Diagram



Two-up is a traditional Australian casino game.

Take a look at the directed graph above, which contains four states, an alphabet, and four transitions.

- **States**: The small circles. We're jumping from state to state. The only thing that's certain is that at a particular time, exactly one state is the current one. The state with a small arrow pointing at it is the *start state*. All states surrounded by a ring are *accept states*.
- **Alphabet**: Only two symbols exist in this alphabet: Heads and Tails. Now, you might wonder why they're drawn multiple times. It's because the same symbol can be part of many transitions.
- **Transitions**: The long arrows between states are called *transitions*. They're armed with at least one symbol from the alphabet. Transitions are the rules for how we can travel. For example, we may go from the start state to the accept state in two transitions if we first get a tails, then another tails.

We call this type of directed graph a *transition diagram*, which is handy for us humans. It's easy to draw a transition diagram. It's also easy to read them and get a quick idea of what they mean. Computers are less enthusiastic about transition diagrams, but we'll find something for them as well.

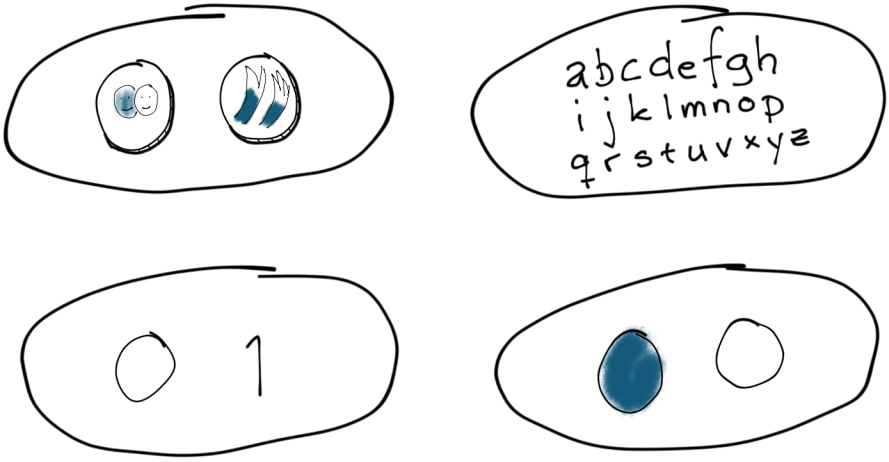
This graph describes how to win the final game of a *Two-up night*. Two-up is a traditional Australian casino game in which you win if you get heads. Twice. In a row. We all know that it's really difficult to win the final game at a casino. Every time we win, our human desire tells us to bet the money we just won—until our wallet is empty.

Every time you toss, you follow one of the transitions based on whether you had heads or tails, from your current state to the next state. If you end up in the accept state—the state surrounded by a ring—you've won the final game.

Two Takeaways

- A transition diagram is a graphical way to depict a finite automaton. It's made up of nodes (states) and edges (transitions).
- Symbols on the edges show which characters are needed to move between states. A ring marks an accepting state.

Alphabet



Four different alphabets: coin, English lowercase, binary digits, and two dots.

Think about *heads* and *tails*. We say that they're two *symbols*, but together they comprise the complete set of possible outcomes when you toss a coin once. We call a finite set of symbols an *alphabet*. So, heads and tails are the coin alphabet. What other alphabets can you think of?

Here are some examples of alphabets:

- A **coin** alphabet comprising the symbols `H` (heads), `T` (tails).
- The **English lowercase** alphabet comprises the symbols `a`, `b`, `c`, … `z`.
- The **binary digits** alphabet comprises the symbols `0` and `1`.
- The **ASCII** alphabet comprises 128 symbols (characters), including some that are not printable.
- A **Five-Emojis** alphabet comprising the symbols 😜, 🤔, 🙄, 🤭, 🙃.
- The **Unicode** alphabet comprises more than 100,000 symbols (characters) from hundreds of scripts.

Note that even Unicode, with more than 100,000 characters, is still finite.

A *string* or a *word* is a finite sequence of symbols from an alphabet. Here are some strings from the alphabets in the previous image:

- `HTTHHT` from the coin alphabet.
- `artichoke` and `grxsttt` from the English lowercase alphabet.
- `01101`, `10`, and `1` from the binary digits alphabet.
- `4$`, `a@a`, and `BIG` from the ASCII alphabet.
- 😜🤔🤔, 🙄🤔, and 🙄🤭🙃 from the Five-Emojis alphabet.
- `fänkål`, `apple`, and `teşekkürler` from the Unicode alphabet.

We may construct an infinite number of possible strings from any alphabet consisting of at least one symbol, even though the alphabet itself is finite. If you're not convinced about this, then imagine a string created with an arbitrary alphabet. For example, consider a string `101` created with the binary alphabet. We may concatenate another symbol—say, `0`—at the end of our string. Voila, we have another string: `1010`. This procedure can go on forever. Thus, the number of strings that can be constructed from a finite alphabet is infinite.

> While it's true that most alphabets can generate an infinite number of strings, there's a unique case involving the empty alphabet (a set with no symbols). This special alphabet can only produce one string, called the empty string, which contains no symbols. We conventionally represent this empty string with the symbol ε. Furthermore, in regular expressions, we don't even consider the empty alphabet to be an alphabet. All alphabets should be limited and non-empty.

In Unicode, and sometimes in this book, a symbol is called a *grapheme*, which is the smallest semantically distinguishing unit in a written language. How a grapheme actually is drawn—that is, what it looks like—is called a *glyph*.

Four different glyphs.

Note that 𝒶 and 𝕒 are two different glyphs that represent the same grapheme in the English lowercase alphabet. Having said this, I'll now tell you that in most cases, symbols will be called *characters* in this book. Thus, although the words are not synonyms, I'll use the words *symbol*, *grapheme,* and *character* interchangeably.

Not in automata theory, but in regular expressions, alphabets must be an *ordered* list of symbols. This means that every symbol has a unique *index* number. In the ASCII alphabet, `A` has an index of 65, `I` has an index of 73, and `K` has an index of 75.

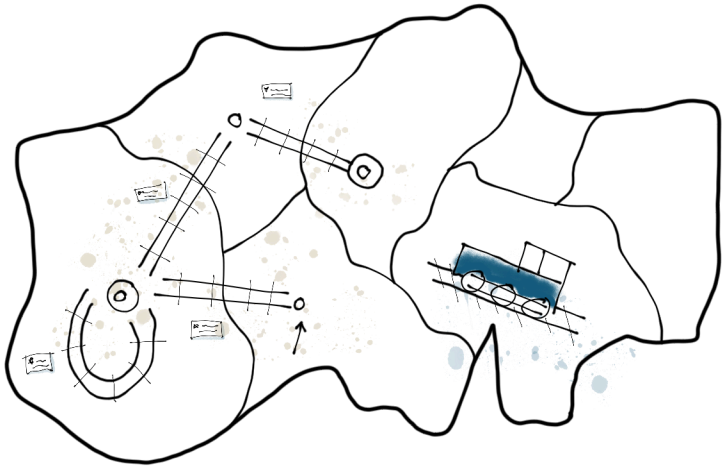
The numbers aren't important in any way other than that they designate and order the symbols. No two symbols have the same index. From indices, we can make ranges, for example, the ASCII range `a-f`, that is, the set `a`, `b`, `c`, `d`, `e`, and `f`. In Unicode, these indices are termed *code points*.

A language is a subset—proper or improper—of all strings that we can construct from an alphabet. For example, all binary strings with a trailing zero are a language: `0`, `00`, `10`, `000`, `010`, `100`, `110`, etc. We'll see later that a regular expression defines a language. As a matter of fact, a directed graph defines a language as well.

Two Takeaways

- An alphabet is a finite collection of symbols used to create strings.
- A string is a sequence of symbols from an alphabet.

States and Transitions



You can be in only one country at a time. Suppose you're traveling by train on a continent comprising eight countries. You always start your trips from your home country. With the right ticket, you can travel from the country in which you're currently located to a neighboring country. You may also take a trip within a country, that is, the journey starts and ends in the same country.

Buy a ticket booklet, and you can make it a tour, that is, you can take several trips, but you're forced to use the tickets in the order in which they're placed in the ticket booklet. If the next ticket in the booklet isn't valid for any trip departing from the country you currently are in, you've failed. You've also failed when you use the entire ticket booklet and end up in a country for which you don't have a permanent residence permit. However, if you start in your home country, take a tour so that you use the entire ticket booklet, and end up in a country in which you do have permanent residence status—then you've succeeded! And don't forget, as I noted, you can be in only one country at a time.

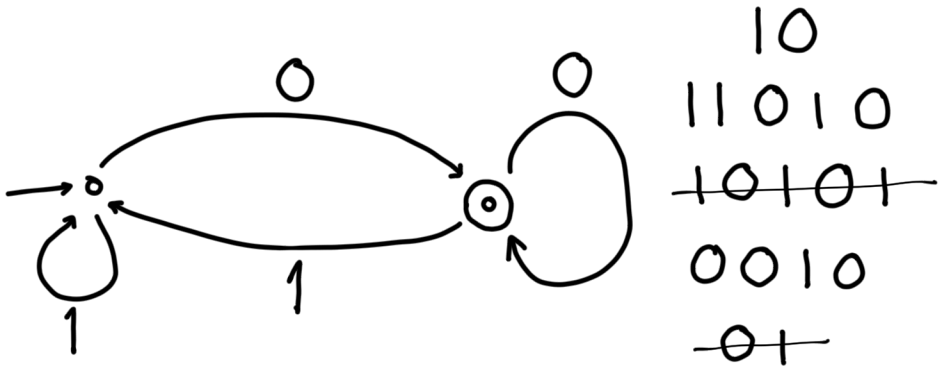
Now imagine an automaton that describes this traveling system:

- Tickets are *symbols.*
- A trip with departure and arrival in the same country is a *loop.*
- The countries are the *states* of the automaton.
- The ticket booklet is a *string.*
- Your home country is the *start state.*
- The countries where you have permanent residence are *accept states.*
- A successful tour is a *string* (of *symbols* from your *alphabet*) that the automaton *accepts*.
- A failed tour is a *string* that the automaton doesn't *accept*.
- The trips are termed *transitions*, and a tour is termed a *walk.*

We can visualize these types of automata as transition diagrams.

- The states are drawn as rings
- A small arrow points at the special start state
- All the accept states—there might be many—have double rings.
- The directed arcs between the rings demonstrate the possible transitions that we may do.
- Each transition is decorated with one or more allowed input symbols.

The following automaton accepts all even binary numbers, that is, the strings of zeroes and ones ending in one or more zeroes.



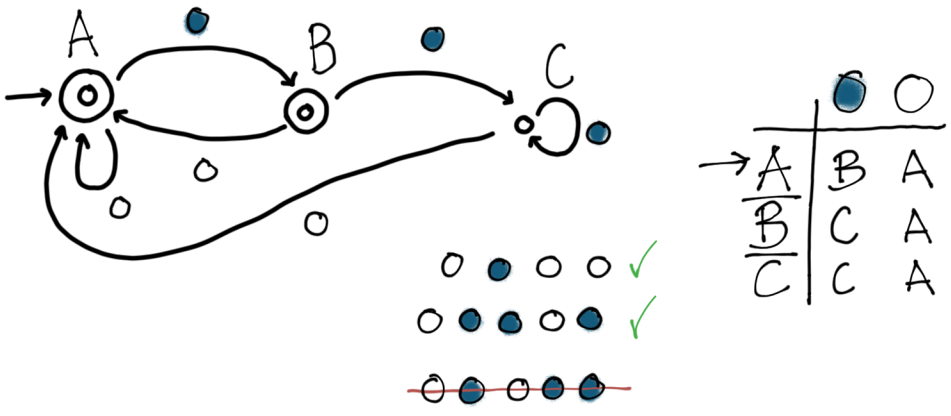
Two Takeaways

- A finite automaton has states, each with a set of possible transitions to other states based on an input symbol.
- A string is accepted if, starting from the designated start state, the automaton can follow transitions based on the input string's symbols and end in an accept state.

Transitions Table

The directed graphs—that is, transition diagrams—that we've seen so far are easy for humans to read, as we quickly develop an intuitive idea of how this system works. The problem is that computers cannot really compete with us when it comes to reading this kind of graphical information because it sometimes looks ambiguous to them. Furthermore, if the computer can't understand the system we want to describe, then it can't help us answer questions about the system's nature and condition. Thus, the $65,536 question is: Can we find a notation that the computer understands? Yes, we can do that.

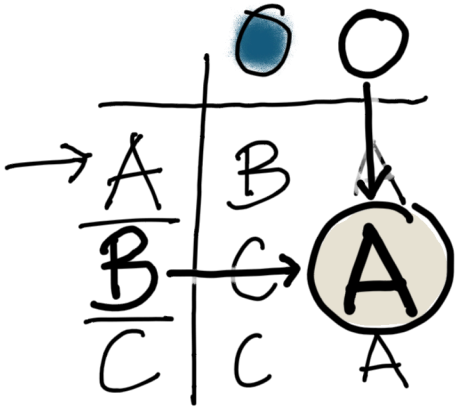
The transition diagram to the left in the following image describes all strings that don't end with two green dots.



This alphabet seems to comprise two symbols: white and green dots. We have three states—A, B, and C. The small arrow that points at A indicates that A is the start state. Both A and B have double rings; thus, they're accept states. If we end up in one of these when we've fed the automaton a whole string, then the string doesn't end with two green dots.

But now we want a notation for computers, such as a table. As a matter of fact, computers love tables. They use tables as directives on how they should behave. To the right in the previous image was a transition table describing exactly the same automaton as the graph to the left. In the table's left column, we have the three states—A, B, and C. A small arrow to the left of A indicates that A is the start state. A and B are underlined, that is, they're valid accept states. The only two symbols in this alphabet, the green and white dots, are in the top row.

At the intersection of the states to the left and the icons at the top, we can see what state we'll transition to next. Assume that we're in state B, and we read a white dot from our input string. We can see in the table that we'd then end up in state A.



The intersection of the state B and the white dot symbol says: "Go to state A."

Humans love graphs and computers love tables. So, can we find a format suitable for both? What's the ultimate interface between humans and computers, a language through which both can understand and express themselves? It's text, of course. As we shall see, Steven Kleene created a text-based language—an algebra—through which we can describe the same automata that we describe with transition tables, as well as with transition diagrams. This algebra is termed *regular expressions*.

Two Takeaways

- A transitions table offers a more computer-friendly representation of a finite automaton.
- The transitions table uses states as rows and input symbols as columns. The cell at the intersection of a state and an input symbol shows the next state the automaton will transition to.

Nondeterministic Finite Automaton

So, we've seen that directed graphs—so-called transition graphs—are easier for us humans to understand, compared with transition tables. We also outshine computers when it comes to interpreting these graphs. However, the opposite is true for something called *non-deterministic finite automata* (NFA). What's a NFA? Let's compare it with its friend *the deterministic finite automata* (DFA)

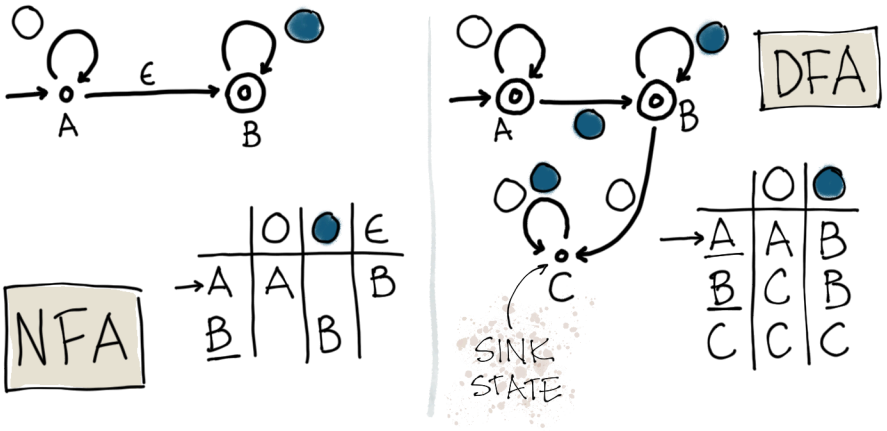
DFAs have exactly one transition—that is, one rule—for each unique pair of a state and an upcoming input symbol. In other words, there's never more than one allowed transition per symbol for each state.

NFAs have a less-rigid definition. Instead of providing a unique alternative for each state and symbol, we allow for multiple possible alternatives. For example, being in state `A` and reading a green dot from the input string gives us two options to choose from: to transition to state `B` or to state `C`. This sounds ambiguous, doesn't it? Here's the catch: We'll let the computer try out all options and see if any of them works—hence, the *nondeterministic* descriptor. Here are some consequences and details about NFAs:

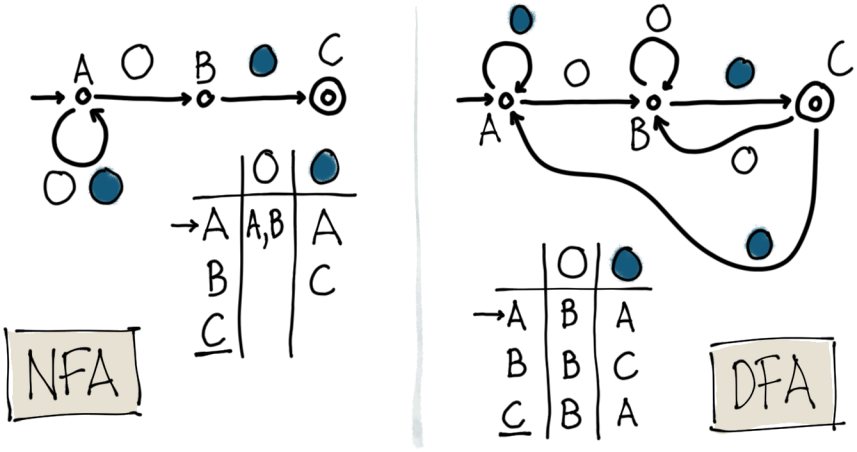
1. If we're in a specific state and have a specific upcoming input symbol, then we specify a set of possible transitions.
2. We have spontaneous transitions—denoted as ε—shifting from one state to another, without consuming any input symbol!
3. The sets of allowed transitions in (1) can be empty. That is, when we're in a specific state and have a specific symbol, we sometimes lack any possible transition away from there.

Computers want clear deterministic rules, for example, an NFA that accepts a string if *at least one possible walk* starts in the *start state*, consumes the string symbols in order, and ends up in one of the automaton's *accept states*.

Enough theory for now. In the following image there are two examples. In the first, we built an automaton—depicted as an NFA (left) and a DFA (right)—that accepts all strings in which no white dot can ever be preceded by a green dot. Notice how much easier the NFA to the left is to read compared with the DFA to the right.



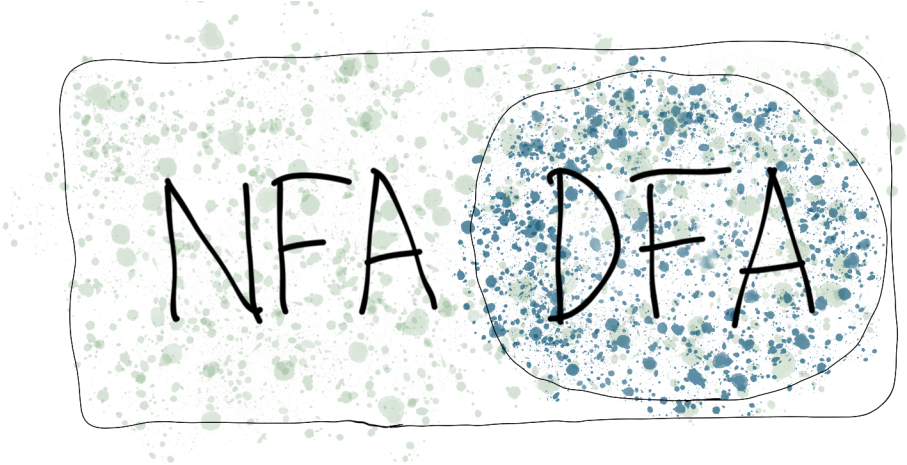
In the second example, an automaton accepts any string that ends with a sequence of first one white, then one green dot. Here, as well, the NFA is much easier to read (and write) than our equivalent DFA.



	0	1
→ A	A,B	A
B		C
C		

	0	1
→ A	B	A
B	B	C
C	B	A

You've probably already figured out that all the DFAs are also NFAs, but that not all NFAs are DFAs. However, what you may not have figured out is that if we take an arbitrary NFA, we can use a general algorithm to transform it into a DFA that accepts exactly the same language, that is, the same set of strings.



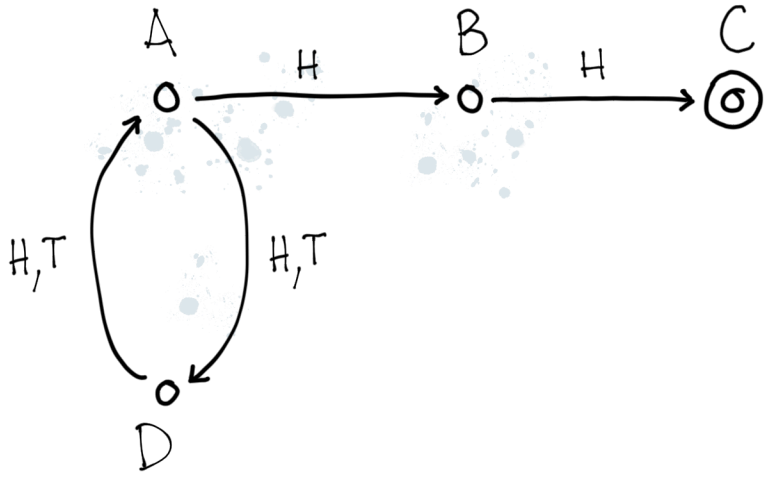
Venn diagram: All DFAs are NFAs. Some NFAs are DFAs.

Thus, DFAs and NFAs solve the same problems. A DFA is always faster, or at least as fast as an NFA. Why? An intuitive explanation is provided in the next section.

Two Takeaways

- A nondeterministic finite automaton (NFA) allows for multiple possible transitions for a given state and input symbol, while a deterministic finite automaton (DFA) has only one.
- Any NFA can be transformed into an equivalent DFA using a general algorithm. All DFAs are NFAs—that is, DFAs are a subset of NFAs.

Greedy and Reluctant



Automaton for Two-up. Symbol H represents heads. Symbol T represents tails.

The NFA's description left us with some question marks. If alternative transitions exist, which ones will the computer try first? How does it know that it has tried all possible walks?

The computer saves a marker each time more than one optional transition occurs from the current state. One marker is saved in the input string and another in the automaton, indicating in what automaton state it was and how much of the input string it had consumed. Then it continues, and the computer may backtrack to these markers later if it happens to reach a dead end, that is, part of the consumed string may be re-consumed in a new walk.

The previous image describes the automaton for Two-up. We flip a coin twice in each game. If the final two games give us heads and heads, then we've succeeded. Suppose we flip the following sequence: heads, tails, heads, and heads (`HTHH`). We then end up in the accept state. The following table shows how the computer works this.

Step	Read	Next	Buffer	Transition
1		H	THH	A → B
2	H	T	HH	*Fail. Backtrack 1 transition.*
3		H	THH	A → D
4	H	T	HH	D → A
5	HT	H	H	A → B
6	HTH	H		B → C
7	HTHH			*Success.* `C` *is an accept state.*

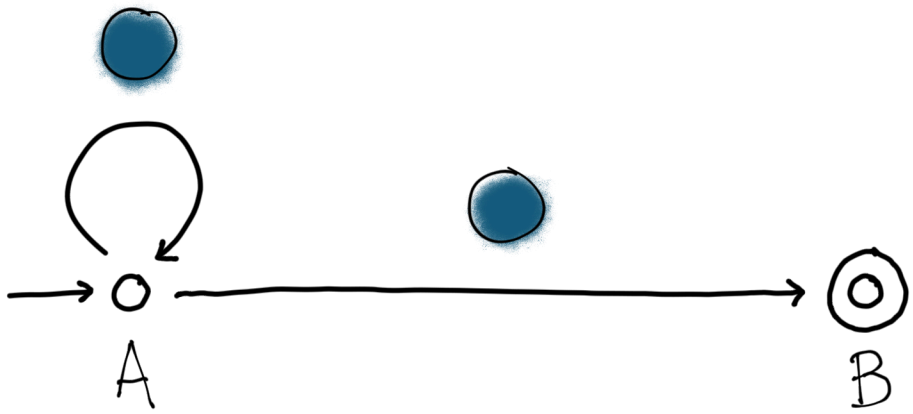
Another example is `HHHT`, in which we don't reach the accept state.

Step	Read	Next	Buffer	Transition
1		H	HHT	A → B
2	H	H	HT	B → C
3	HH	H	T	*Fail. Backtrack 2 transitions.*
4		H	HHT	A → D
5	H	H	HT	D → A
6	HH	H	T	A → B
7	HHH	T		*Fail. Backtrack 1 transition.*
8	HH	H	T	A → D
9	HHH	T		D → A
10	HHHT			*Fail.* `A` *isn't an accept state.*

The nondeterministic finite automaton in this example contains decision points in state A in which there are two options: Either let the current state consume one more symbol of the input string—that is, loop—or else transition to another state. Thus, we have two mutually exclusive strategies that the computer may use, which I term *greedy* and *reluctant*. Some people say *lazy* rather than reluctant.

- With the **greedy** strategy, we try—as long as possible—to be in a self-transitioning loop of the current state. In this strategy, each state is greedy and tries to consume as much as possible from the input string.
- With the **reluctant** strategy, we try—as quickly as possible—to transition from the current state to another state, ultimately to arrive early in an accept state. Each state is reluctant to consume more of the input string and, if possible, tries to pass the initiative to the next state instead.

Below is an NFA that accepts all non-empty sequences of green dots 🟢.



Suppose that our automaton is fed with a string comprising two green dots (🟢🟢). With the greedy strategy, it would be.

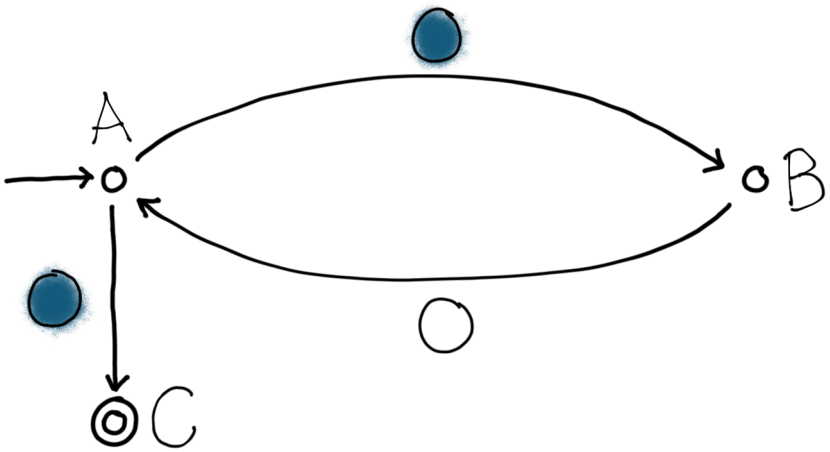
Step	Read	Next	Buffer	Transition
1		🟢	🟢	A → A

2	●	●		A → A
3	●●			*Fail. Backtrack 1 step.*
4	●	●		A → B
5	●●			*Success.* B *is an accept state.*

However, if the automaton consumes the same string with the reluctant strategy it would travel differently.

Step	Read	Next	Buffer	Transition
1		●	●	A → B
2	●	●		*Fail. Backtrack 1 step.*
3		●	●	A → A
4	●	●		A → B
5	●●			*Success.* B *is an accept state.*

The loop may be longer, such as in this automaton, which accepts all strings with alternating green and white dots, in which the first and last dots are green.



Imagine that the input is a white dot surrounded by two green dots (●○●). With the greedy strategy, the loop tries to consume as much as possible.

Step	Read	Next	Buffer	Transition
1		●	○●	A → B
2	●	○	●	B → A
3	●○	●		A → B
4	●○●			*Fail. Backtrack 1 step.*
5	●○	●		A → C
6	●○●			*Success.* C *is an accept state.*

However, the reluctant algorithm would instead give us a different journey.

Step	Read	Next	Buffer	Transition
1		●	○●	A → C
2	●	○	●	*Fail. Backtrack 1 step.*
3		●	○●	A → B
4	●	○	●	B → A
5	●○	●		A → C
6	●○●			*Success.* C *is an accept state.*

The conclusion is that backtracking is a waste of time for the computer. If our algorithm finds the best walk in the first shot, then our NFA will be as fast as a DFA. We'll see later in this book that we may influence exactly that.

The next section explains how an NFA can be translated into a DFA. You may skip this if you're not utterly fascinated with the most advanced parts of regular expressions, but I think you are.

Two Takeaways

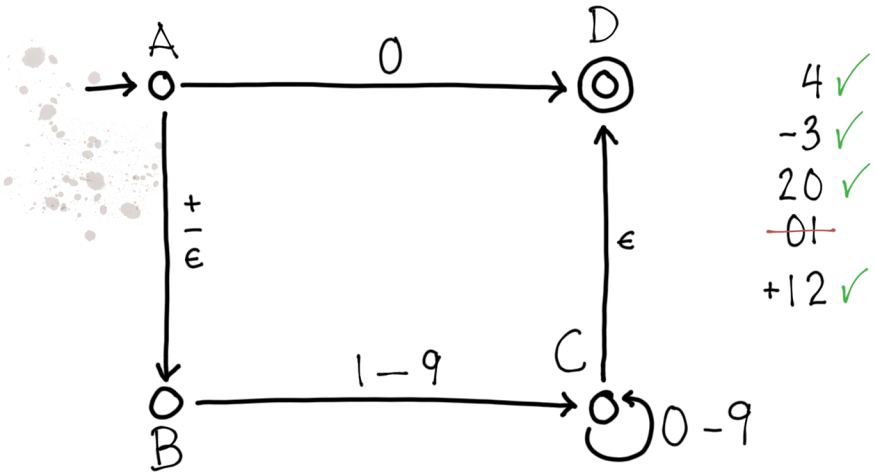
- When faced with multiple possible transitions, the automaton can use either a greedy strategy (consume as much of the input as possible) or a reluctant strategy (consume as little as possible).
- Backtracking can occur when the automaton reaches a state where it cannot proceed with the current strategy. It involves going back to a previous state and trying a different transition.

NFA to DFA

DFAs are faster than NFAs, so why bother with NFAs anyway? As we'll see later in this book, NFAs can be extended with features that are either impossible or at least very difficult to implement in DFAs. When we don't use these features, DFAs are superior. The good news is that any NFA can be transformed into a DFA that accepts exactly the same set of strings. Here's an example of how to do this.

Following, we have an NFA that accepts every integer, that is, digit sequences. The integers may be preceded by a plus or minus sign. We have two constraints:

1. Only the integer zero can start with the digit `0`.
2. The integer zero can't be preceded by a plus or minus sign.



Nondeterministic finite automaton.

Remember that the symbol ε is used to indicate that we can make a spontaneous transition, that is, a transition without consuming any symbol from the input. To translate an NFA into a DFA, we need something called ε-closure. The ε-closure for a

state is the set of states that can be reached with ε-transitions. Thus, the ε-closure of `C` in the previous image is `{C, D}`.

Considering that `A` is our NFA's start state, the ε-closure of `A`—which is `{A, B}`—becomes the start state of our new DFA that we're creating. From the DFA state `{A, B}`, we can get to DFA state `{D}` if we read the symbol `0`, to DFA state `{B}` if we read a plus or minus sign, and to DFA state `{C, D}` (the ε-closure of `C`) with symbols in the range of `1–9`. Thus, the first attempt in our DFA transition table is the following.

STATE	+ -	0	1–9
{A, B}	{B}	{D}	{C, D}
{B}	?	?	?
{D}	?	?	?
{C, D}	?	?	?

The question marks will soon be replaced by ε-closures that we can reach from `{B}` and `{D}`.

Let's continue with `{B}`, from which we can reach `{C, D}` if we read a digit in the `1–9` range. That's it. No other transitions from DFA state `{B}` exist.

STATE	+ -	0	1–9
{A, B}	{B}	{D}	{C, D}
{B}	¤	¤	{C, D}
{D}	?	?	?
{C, D}	?	?	?

The pillow symbol ¤ means the empty set `{ }`, which is for sure a set, even though it doesn't include any of our NFA-states. We use it for example at the intersection in the previous table between the DFA state `{B}` and the symbols `+-` to visualize that reading plus or minus while in `{B}` is a failure. If that happens then we'll simply not accept the input string—it's not an integer—and therefore transition to the *sink state*.

Most certainly, you already know that the pillow symbol ¤ is also known as *the generic currency sign*. As computers started handling financial information in the 1970s, a need emerged for a universal symbol to represent different currencies. Enter the ¤, proposed and included in early character sets like ASCII. The idea was to use it as a placeholder, later replaced with the specific currency symbol as needed. Though it never quite caught on as a universal stand-in, the ¤ remains a curious relic of the efforts to standardize and digitize global currencies. However, let's return to the DFA we were building before this finance trivia story got in our way.

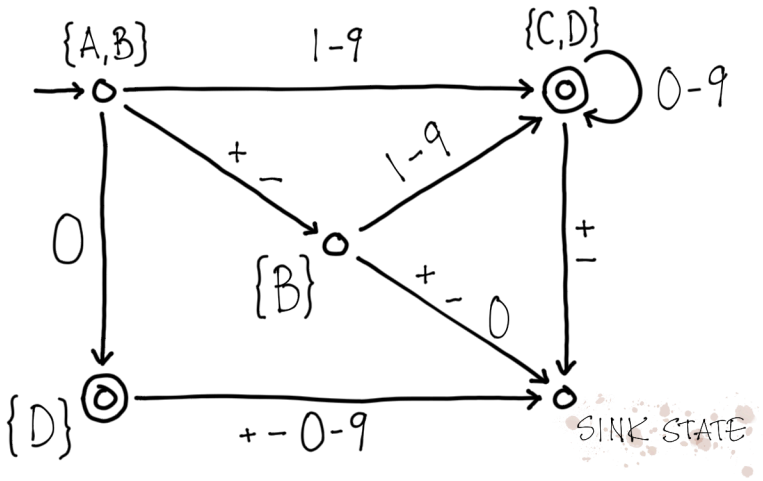
Perhaps you're now wondering which of these new states are accept states. The answer is that any DFA state containing an accept state from the NFA is an accept state. So far, `{D}` and `{C, D}` are accept states because they contain `D` which was the only accept state in our NFA.

We continue to fill the transition table the same way.

STATE	+ -	0	1-9
{A, B}	{B}	{D}	{C, D}
{B}	¤	¤	{C, D}
{D}	¤	¤	¤
{C, D}	¤	{C, D}	{C, D}

We now have four states, plus the *sink state*. The latter is where we end up when we know that this isn't an integer.

Following is a transition diagram for our DFA. Note that from each and every state, exactly one transition exists for each and every symbol in our alphabet `{+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}`, that is, this finite automaton is deterministic.



Deterministic finite automaton.

To summarize:

- Transitioning in NFAs may include alternatives, but in DFAs the path is deterministic.
- All DFAs are NFAs, but all NFAs are not DFAs.
- We have an algorithm to transform any given NFA to a DFA.

Two Takeaways

- While NFAs offer flexibility, DFAs are generally faster because they avoid backtracking.
- The transformation from NFA to DFA involves creating new states in the DFA that represent sets of states from the NFA.

Pushdown Automata

A problem with finite automata is that they don't have any memory. Once they're in a state, they have no idea how they got there. Wise people invented the stack. When we add a stack to a finite automaton, it becomes very powerful, but also quite complicated. Thus, it's not a finite automaton anymore, of course—it's a pushdown automaton.

We have seen that the finite automaton reads an input string serially. Every new symbol read from the input string initiates a transition in the automaton from the current state to a new state (the new state may be the same as the current state, that is looping). If we're in an accept state when we've read the whole input string, then we have a match. Bingo! However, in a pushdown automaton, we also add a stack, that is, a data structure operating last in, first out (LIFO).

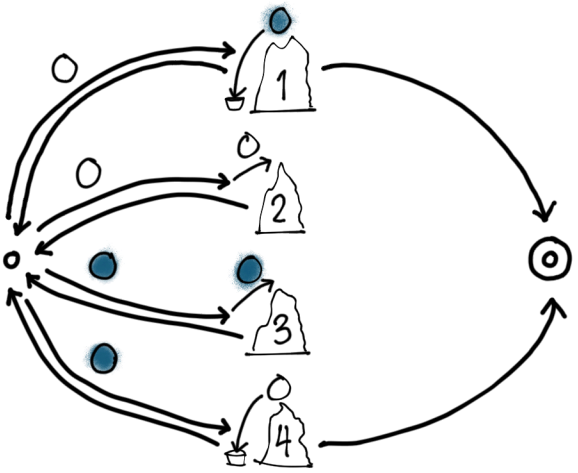
Supported by the stack, our pushdown automaton iterates in two steps. For each iteration, our automaton may read a symbol from the input string, pop a symbol from the top of the stack, or both:

- **First Step Option 1:** Read a symbol from the input string.
- **First Step Option 2:** Pop a symbol from the top of the stack.
- **First Step Option 3:** Read a symbol AND pop a symbol.

After the first step, our pushdown automaton needs to make a decision based on the current state, the symbol read from the input string, and/or the symbol popped from the stack. The possible decisions are to initiate a state transition, push a symbol onto the top of the stack, or both:

- **Second Step Option 1:** Initiate a state transition.
- **Second Step Option 2:** Push a symbol onto the top of the stack.
- **Second Step Option 3:** Initiate a state transition AND push a symbol.

If we end up in an accept state when the whole input string is consumed (read) and the stack is empty at that time, then we have a match. Pushdown Bingo!



Pushdown automaton.

This image depicts a pushdown automaton that matches any input with the same number of green and white dots—something impossible to describe with a finite automaton. This automaton has two states: a start state and accept state. The four icons in the middle indicate that a dot is popped or pushed from the stack. Suppose that the input is green-white-white-green ●○○●:

1. Read the green dot and make a transition from the start state, through the third stack icon from the top—that is, a green dot is pushed to the stack—and back to the start state.
2. Read the white dot and make a transition from the start state, through the first stack icon—that is, a green dot is popped from the stack—and back to the start state. Now the stack is empty.
3. Read the white dot and make a transition from the start state, through the second stack icon from the top—that is, a white dot is pushed to the stack—and back to the start state.
4. Read the green dot and make a transition from the start state, through the fourth stack icon—that is, a white dot is popped from the stack—and to the

accept state. Now the stack is empty again, and all the input is read. It's a match.

You might suspect by now that the pushdown automaton gives us a tool for all kinds of recursive implementations. Now that you understand how, I'll go back to finite automata in all explanations wherever possible. It's very important to truly understand how, for example, backtracking and greediness impact performance and also what input will be matched. You don't need a complicated pushdown automaton to learn this. We'll stick to finite automata in this book and simultaneously remember that many operators in modern regular expressions automata rely on a stack, which is why modern regular expressions cores are implemented as pushdown automata.

By the way, did you notice the corner case bug in the pushdown automaton that we just explored? Doesn't an empty input string have the same number of green and white dots?

Two Takeaways

- A pushdown automaton is an extension of a finite automaton with an added stack, a LIFO data structure.
- The stack allows the pushdown automaton to remember previous states and symbols, enabling it to recognize languages that finite automata cannot.

PART II: Two Operations and One Function

In Part I of this book (The Automaton), we observed that the same problem could be expressed in a human-friendly way (a graph) and a computer-friendly way (a table). We, the humans, are creative. We formulate new problems that we want to solve. On the other hand, computers solve problems at a high speed, and they never make mistakes—at least not if we implement the correct algorithms. How do we find a notation that's easy for people to express themselves with and also easy for computers to interpret?

Directed graphs, that is, transition diagrams, give us a good overview, as long as they don't contain too many details. We can focus on a piece of the graph, and we may also zoom out to see the big picture as an abstract description. When we walk around in the graph, we're in an area. Our brains thrive like a fish in water with this way of presenting and digesting information.

The computer excels at having full control over every detail of a large amount of data, as it doesn't need any abstraction to understand the big picture. It just wants deterministic instructions on how to behave in its current state. If we provide all possible transitions in tabular form, the computer will remain happy, but for us humans, it's difficult to get an overview of a transition table. We will wonder what problem this table is solving.

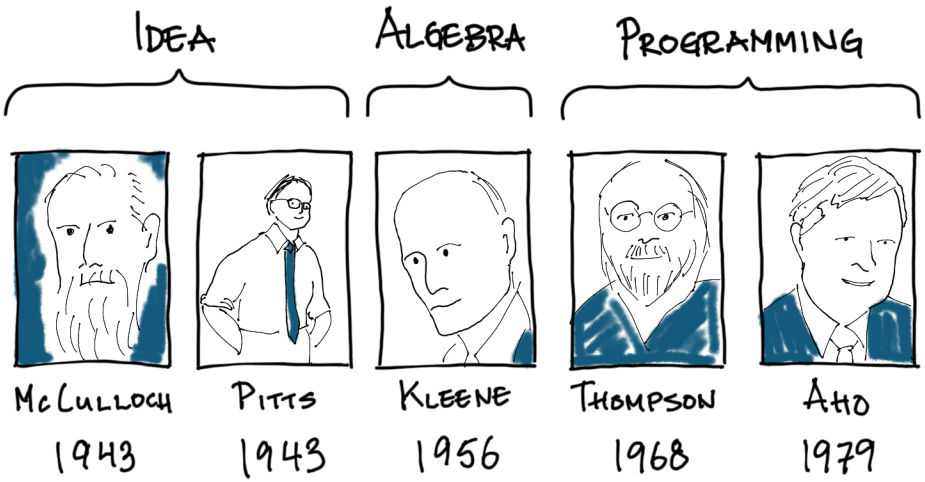
When it comes to details, the ultimate interface between humans and computers has long proven to be text. Visual programming languages are launched recurrently in our industry, but the market never takes off. We still must teach the computer all the details about how it should behave. Too many details make the graphical programs as difficult to interpret for us humans as they are for computers. Programming

languages such as C, Java, C#, Python, and Ruby are completely text-based and still are uncrowned queens of expression for us programmers.

> Where nothing else is said, examples are executed 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.

Even a text-based programming language needs an effective and simple notation. In this part of the book, we'll see that regular expressions (one regex, many regexes) is a language with only two operations and one function. Sounds powerful, doesn't it? Two operations and one function are sufficient to describe all the transition diagrams and transition tables that can be constructed. It's enough to describe every possible DFA and NFA in text form.

History of Regular Expressions



The regular expressions pioneers.

Regular expressions is a programming language with which we can specify a set of strings. Supported by only two operations and one function, we can be very concise. A non-concise alternative would be to list all the strings included in the set. Where does this regular expressions language come from, why is it called *regular*, and how does it differ from *regex*?

The story begins with a neuroscientist and a logician who together tried to understand how the human brain could produce complex patterns using simple cells that are bound together. In 1943, *Warren McCulloch* and *Walter Pitts* published "A logical calculus of the ideas immanent in nervous activity" in the *Bulletin of Mathematical Biophysics* 5:115–133. Although it was not the objective, it turned out that this paper greatly influenced computer science in our time. In 1956, mathematician *Stephen Kleene* took McCulloch and Pitts' theories one step further. In the paper "Representation of events in nerve nets and finite automata" in the Annals of Mathematics Studies, Number 34, Kleene presented a simple algebra, and

somewhere along the line, the terms *regular sets* and *regular expressions* were born. As mentioned, Kleene's algebra had only *two operations* and *one function*.

In 1968, Unix pioneer *Ken Thompson* published the article "Regular Expression Search Algorithm" in *Communications of the ACM* (CACM), Volume 11. With code and prose, he described a regular expression compiler that creates IBM 7094 object code. Thompson's efforts did not end there. He also implemented Kleene's notation in the editor QED. The value was that users could do advanced pattern matching in text files. The same feature appeared later in the editor ed.

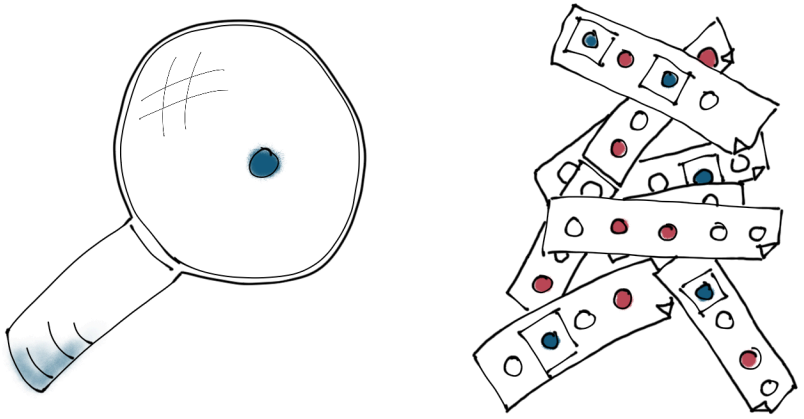
To search for a regular expression in ed, the user wrote `g/<regular expression>/p`. The letter `g` meant global search, and p meant print the result. The command—`g/re/p`—resulted in the stand-alone program grep, released in the fourth edition of Unix in 1973. However, grep didn't have a complete implementation of regular expressions, and it was not until 1979, in the seventh edition of Unix, when we were blessed with Alfred Aho's egrep (extended grep). Now the circle was complete. The egrep program translated any regular expressions into a corresponding DFA.

Larry Wall's Perl programming language from the late 1980s helped regular expressions become mainstream. Perl integrated regular expressions seamlessly, even with regular expression literals. Perl also added new features to regular expressions. The language was extended with abstractions, syntactic sugar, and also some brand new features that may not even be possible to implement in finite automata. This raises the question of whether modern regular expressions can be called regular expressions. Perhaps we can override that discussion if we use the term regex instead of regular expressions when we refer to the not-so-regular regular expressions?

Two Takeaways

- Regular expressions stemmed from the work of Warren McCulloch and Walter Pitts in 1943, who studied how the human brain produces complex patterns with interconnected cells.
- The term "regular expressions" originated from Stephen Cole Kleene's work in 1956, who built upon McCulloch and Pitts' theories to develop a simplified algebra for describing languages.

Match One Character



An alphabet is a finite, nonempty set of symbols, that is, at least one symbol and a limited number. Here are some examples of alphabets:

- The binary alphabet `{0, 1}` contains only two symbols: `1` and `0`.
- The American Standard Code for Information Interchange (ASCII) contains 128 symbols, only 95 of which are printable. The others, such as *Backspace* (ASCII 8), are also symbols in our definition.
- The set of 95 printable symbols in ASCII is also an alphabet.
- Unicode contains over 100,000 symbols that include Arabic, Chinese, Latin, and many other types. However, even if it's a very large number, it's still a finite number of symbols. Therefore, Unicode is an alphabet.
- The set of the Cyrillic symbols in Unicode is an alphabet.
- We can construct an alphabet comprising a green dot 🟢 and a white dot ⚪, that is, an alphabet comprising two symbols {🟢, ⚪}, just like the binary alphabet `{0, 1}`.

Traditionally, a specific order of the symbols doesn't make an alphabet unique. Thus, `{a, b}` is the same alphabet as `{b, a}`. However, as we'll see later in this book, order is significant for defining ranges in modern regex.

> Did you, by any chance, know that the word *alphabet* comes from *alpha* and *beta*, the first two letters of the 3,000-year-old Phoenician alphabet, and that alpha meant ox and beta meant house?

A string is a finite ordered sequence of symbols chosen from an alphabet. Here are some examples of strings:

- `0110` and `1001` are two strings from the binary alphabet.
- The strings `kadikoy` and `uskudar` are constructed with symbols from the ASCII alphabet.
- `футбол` is a string from the Cyrillic alphabet.
- اللوز is a string from the Arabic alphabet.
- From any alphabet, it's possible to create an empty string, that is, a string comprising zero symbols. By convention, we denote that string with the character ε.

The same symbol can occur multiple times in a string, but the order is significant, for example, `gof` isn't the same string as `fog`.

A language is a countable set of strings from a fixed alphabet. Recall that the alphabets are restricted to be finite. A language, however, may well include an infinite number of strings. Here are some examples of languages:

- All binary numbers ending in `0`, for example, `10`, `100`, `110`, etc.
- All palindromes—strings that are the same forward and backward, like `otto` and `anna`—constructed from ASCII symbols.
- All strings with an even number of symbols from Unicode.
- A finite set `{dog, cat, bird}`.
- An empty set, that is, a language with no strings. Such an alphabet is by convention denoted as ∅.
- The language `{ε}`, which comprises only one symbol: the empty string ε. (Note that `{ε}` is very different from ∅, for example in string cardinality.)

Our focus in this book is on regular expressions. Evidently. A regular expression is an algebraic description of an entire language. Not all languages can be described using regular expressions. For example, no regular expression—without non-regular extensions—can describe the language comprising all palindromes created with symbols from ASCII. However, I'll try to convince you that regular expressions are ridiculously easy to learn and amazingly powerful to use.

Regular expressions starts with two basic rules:

(1) Empty String Language. The empty string ε is a regular expression that describes a language comprising only one string, which happens to be the empty string.

(2) Single Symbol Language. If a symbol, `a`, is a member of an alphabet, then a is a regular expression. It describes a language that has the string "`a`" as its only member.

That was easy, wasn't it? In addition to these two basic rules, we have two operations and one function that we'll cover with another four rules in the next section. Out of pure laziness, we have conventions for precedence between these operations and the function. More on this later.

First, you should try the two basic rules in the Interactive Ruby Shell (IRB). Either you install Ruby on your computer and start IRB in a shell, or you may use an online version of IRB.

With IRB, we write regular expressions between two dashes. For example, you may write the regular expression `a` as `/a/`. IRB can help us figure out whether a string is in a language described by a particular regular expression:

```
'a'.match /a/ #=> #<MatchData "a">
   # string a matched by regex /a/
''.match /a/ #=> nil
   # empty string not matched by /a/
'b'.match /a/ #=> nil
   # string b not matched by /a/
'a'.match // #=> #<MatchData "">
```

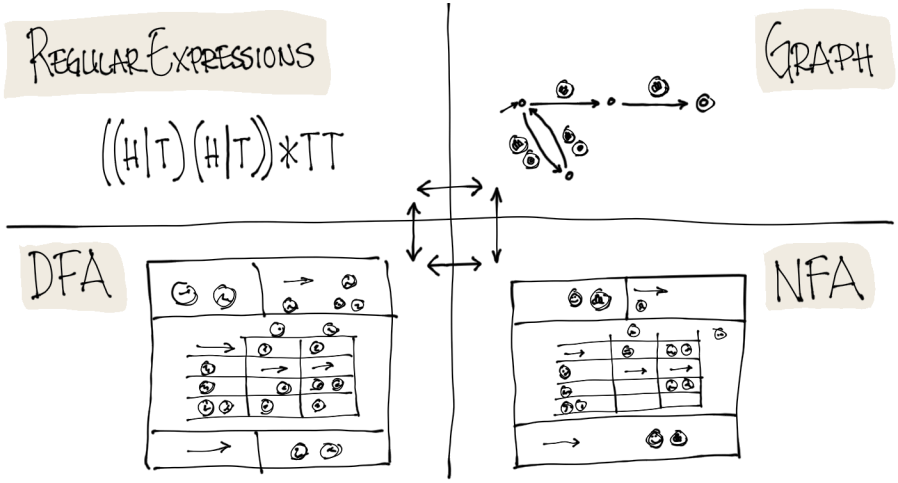
```
# string a not matched by empty regex ε
```

Only when IRB returns the entire string is it matched by the regular expression. The number sign `#` (hash tag) and everything to the right of it is ignored by the computer. We can put messages to humans there, for example what data we expect to match.

Two Takeaways

- A string is a finite sequence of symbols from an alphabet, and a language is a set of strings.
- A regular expression is a way to describe a language algebraically

Four More Rules



Now that we have two basic rules, we'd like to add four more. We then can build more advanced regular expressions recursively from very basic regular expressions. The first three rules (№3–5) describe regular expressions' only two necessary binary operations and one and only necessary function. The fourth rule (№6) deals with parentheses:

(3) Alternation. If `p` and `q` are regular expressions, then `p|q` also will be a regular expression. The expression `p|q` matches the union of the strings matched by `p` and `q`. Think of it as either `p` or `q`.

(4) Concatenation. If `p` and `q` are two regular expressions, then `pq` also will be a regular expression. Note that the symbol for concatenation is invisible. Some literature uses × for concatenation, for example, `p×q`. The expression `pq` denotes a language comprising all strings with a prefix matched by `p`, followed by a suffix matched by `q`—and nothing between the prefix and suffix.

(5) Closure. If p is a regular expression, then p* also will be a regular expression. This is the closure of concatenation with the expression itself. The expression p* matches all strings that may be divided into zero or more substrings, each of which is matched by p.

(6) Parentheses. If p is a regular expression, then (p) will be a regular expression as well, that is, we can enclose an expression in parentheses without altering its meaning.

In addition to these rules, we'll add some convenience rules for operator precedence shortly. They're not necessary, but allow us to write shorter and more readable regular expressions. Quite soon, you'll also see real regular expression examples based on these two operations and one function. It might be difficult to imagine that these six rules are sufficient for writing every possible regular expression in the world.

Do you remember George Bernard Shaw's quote *"The golden rule is that there are no golden rules"*? What about Mark Twain's *"It is a good idea to obey all the rules when you're young just so you'll have the strength to break them when you're old"*? This is exactly how we should think. For now, these rules are all we need, but modern regex automata contain powerful functions, for example, a back reference and lookarounds. To implement these functions, we need more than these six rules, but until we're there, it'll be very useful to think of regular expressions as a system comprising only these six rules.

Thus, regular expression is a mathematical theory, and modern regex automata are based on a super set of this theory. With the help of the theory, we can prove the following:

- For each regular expression, we may construct at least one DFA and at least one NFA, so that all three (regular expression, DFA, and NFA) solve the same problem.
- For every finite automaton—deterministic (DFA) as well as non-deterministic (NFA)—we may write a regular expression, so that both (automaton and regular expression) solve the same problem.

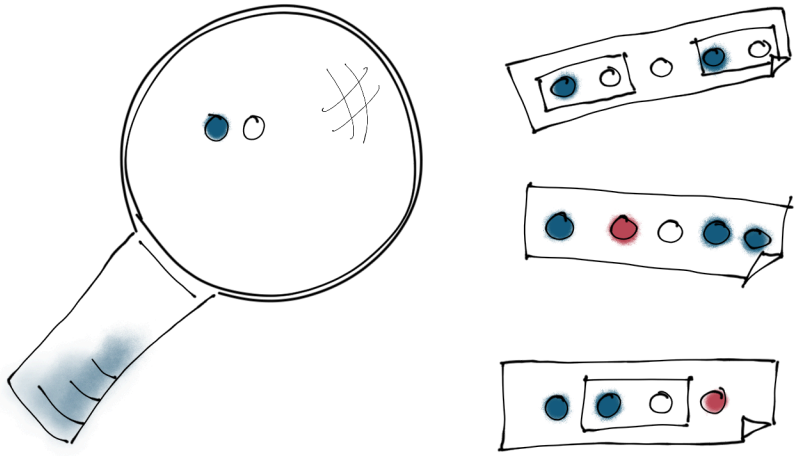
Solving a problem here means determining whether a string is part of a language, that is, a specific set of strings. The proofs mentioned here aren't reproduced in this book, but they're easily found in every textbook on automata theory.

In summary, regardless of which format we start with—NFA, DFA, regular expression, or state diagram— there is always an equivalent representation in all the other formats. The beauty of this equivalence is that I can explain several key features of regular expressions for you, with the help of graphs of finite automata. Furthermore, in almost every mainstream programming language, there is a compiler that translates our handwritten regular expressions into computer-friendly finite automata, or possibly more advanced pushdown automata.

Two Takeaways

- In addition to the two basic rules for the empty string and single symbol languages, regular expressions have two operations (alternation `p|q` and concatenation `p×q`) and one function (closure `p*`).
- Parentheses can be used in regular expressions to group expressions and alter the order of operations.

Operation №1: Concatenation



Using Rule №2 (Single Symbol Language) and №4 (Concatenation), any possible sequence of symbols from our alphabet may be written as a regular expression. Rule №2 said that if the symbol `a` is in the alphabet, then "`a`" is a regular expression. Rule №4 said that if `p` and `q` are two regular expressions, then the concatenation `pq` is a regular expression as well. The concatenation symbol itself is invisible. Just write the two regular expressions right after each other.

```
'moda'.match /m/ #=> #<MatchData "m">
'moda'.match /o/ #=> #<MatchData "o">
'moda'.match /mo/ #=> #<MatchData "mo"> m×o is mo
'moda'.match /da/ #=> #<MatchData "da"> d×a is da
'moda'.match /moda/ #=> #<MatchData "moda"> mo×da is moda
'moda'.match /mado/ #=> nil -- mado is not moda
```

We usually use some handy terms for substrings:

- A **prefix** is the substring we leave if we remove zero or more symbols from the end of a string. The strings "m", "mo", "mod", and "moda" are all prefixes of the string "moda". Even the empty string ε is a prefix of "moda".
- A **suffix** is the substring that remains if we remove zero or more characters from the beginning of the string. The strings "moda", "oda", "da", "a" and ε are all possible suffixes of the string "moda".
- A **substring** is what we have left if we remove a prefix and a suffix from a string. Note that even ε is a valid prefix and suffix. Symbols in the substrings must be consecutive in the original string. The strings "od" and "moda", but not "mda", are substrings of "moda".

For any regular expression p, it's true that εp = pε = p; thus, we say that the empty string ε is the *identity operand* under concatenation.

Concatenation isn't commutative because pq isn't equal to qp, but it's associative because for any regular expressions, p and q, it's true that p(qr) = (pq)r.

If we view concatenation as a multiplication product, then regular expressions also support exponentiation, that is, x^n. We write the exponent enclosed in brackets {n} to the right of the regular expression.

```
'aaa'.match /aaa/ #=> #<MatchData "aaa">
'aaa'.match /a{3}/ #=> #<MatchData "aaa">
   # yes, the string includes 3 concatenated a
'aaa'.match /a{4}/ #=> nil
   # no, the string doesn't include 4 a
```

This is obviously just syntactic sugar. We can unfold all regular expressions that we may write using the exponential operation. More shortcuts can be used for finite repeated concatenations.

```
'aa'.match /a?/ #=> #<MatchData "a">
   # optional match, written as question mark
'b'.match /a?/ #=> #<MatchData "">
```

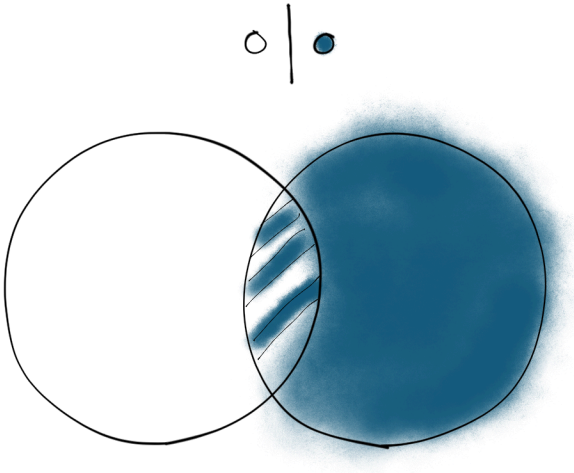
```
   # zero repeats of a matches empty string
'aa'.match /a{,2}/ #=> #<MatchData "aa"> at least two a
'aa'.match /a{1,2}/ #=> #<MatchData "aa">
   # at least one a and at most two a
'a'.match /a{1,2}/ #=> #<MatchData "a">
```

We'll soon see that concatenation of two regular expressions isn't the same as concatenation of two literal strings. This is obvious when we recall that a regular expression corresponds to a set of strings, not a single literal string. For example, if `p` = `{a, b}` and `q` = `{c, d}`, then `pq` = `{ac, ad, bc, bd}`

Two Takeaways

- Concatenation in regular expressions involves combining two regular expressions, where the resulting expression matches strings that are formed by concatenating strings matched by the individual expressions.
- The empty string is the identity element for concatenation, and concatenation is associative but not commutative.

Operation №2: Alternation



Venn diagram showing exclusive disjunction (XOR).

From Rule №2 (Single Symbol Language) and №3 (Alternation), we may define paradigms—a number of possible patterns, that is, we add two or more languages by applying the *union* set operation to them. The union of the sets `{a, b}` and `{c, d}` is `{a, b, c, d}`; thus, it's all the elements that are either in one or more of the sets that we unite. In Boolean logic, we call this the *inclusive or*. In regular expressions, it's called alternation and is written with a vertical bar `|`.

```
'a'.match /a|b/ #=> #<MatchData "a"> a is either a or b
'ab'.match /a|b/ #=> #<MatchData "a"> leftmost chosen
'ba'.match /a|b/ #=> #<MatchData "b"> leftmost chosen
'c'.match /a|b/ #=> nil -- here we found neither a nor b
```

Note that when more than one alternative is correct, most regex dialects select the leftmost alternative, but exceptions to this rule exist. A regex automaton based on DFA or POSIX NFA selects the longest alternative. However, most other regex automata select the leftmost.

Can you write a regular expression that matches all binary strings of length one? The binary alphabet is `{0, 1}`. Considering that few binary strings are of length one, you may even list them: `{0, 1}`. The regular expression with alternation then becomes `/0|1/`.

```
'0'.match /0|1/ #=> #<MatchData "0">
'1'.match /0|1/ #=> #<MatchData "1">
'2'.match /0|1/ #=> nil
'10'.match /0|1/ #=> #<MatchData "1">
```

Note that in the last example, IRB matched the substring `1`.

Four binary strings of length two `{00, 01, 10, 11}` can be captured with `/00|10|01|11/`.

```
'10'.match /00|10|01|11/ #=> #<MatchData "10">
'01'.match /00|10|01|11/ #=> #<MatchData "01">
'12'.match /00|10|01|11/ #=> nil
'11'.match /00|10|01|11/ #=> #<MatchData "11">
'1210'.match /00|10|01|11/ #=> #<MatchData "10">
```

Did you notice that we used concatenation in all these regular expressions? (Can you see the invisible concatenation symbol between the two binary digits in the regular expression? If not, maybe you should make an appointment with an optometrist—or maybe not. Not even an optometrist can help you see invisible symbols, which is what the concatenation symbol is.) Each of the binary strings of length two comprises two concatenated binary strings of length one. Considering that concatenation takes precedence over alternation, we didn't need any parentheses.

Alternation is commutative, that is, for two regular expressions, `p` and `q`, it holds that `p|q` = `q|p`. It's also associative: `p|(q|r)` = `(p|q)|r`.

An interesting and very useful fact is that concatenation distributes over alternation, that is, for all regular expressions `p`, `q`, and `r`, it's true that `p(q|r)` = `pq|pr` and `(p|q)r` = `pq|pr`. A consequence of this is that `/(0|1)(0|1)/` = `/(0|1)0|(0|1)1/` = `/00|10|01|11/`. So, another way to match any binary strings of length two is the following.

```
'10'.match /(0|1)(0|1)/ #=> #<MatchData "10">
'01'.match /(0|1)(0|1)/ #=> #<MatchData "01">
'12'.match /(0|1)(0|1)/ #=> nil
'11'.match /(0|1)(0|1)/ #=> #<MatchData "11">
'1210'.match /(0|1)(0|1)/ #=> #<MatchData "10">
```

Of course, the brackets are necessary because concatenation takes precedence over alternation. We also may add the empty string ε as one of our alternatives.

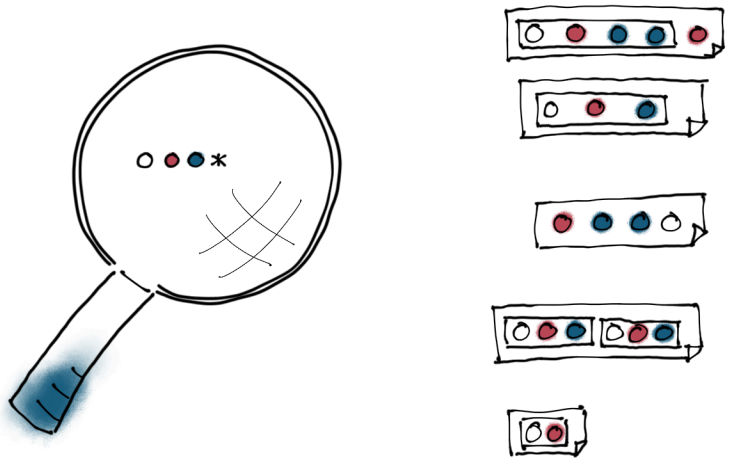
```
'moda'.match /moda|/ #=> #<MatchData "moda">
   # either moda or nothing is moda
'moda'.match /mado|/ #=> #<MatchData "">
   # either mado (not moda) or nothing is nothing
```

Given all this, here's an opportunity to convince you that all regular expressions have an infinite number of synonyms. A revealing example: `(a) = (a|) = (a||)` etc.

Two Takeaways

- Alternation in regular expressions, denoted by the vertical bar `|`, allows matching either the expression before or after the bar.
- Alternation is commutative and associative, and concatenation distributes over alternation.

The Function: Kleene Star



All finite languages can be described using regular expressions. We simply list the strings as an `string1|string2|string3|…` alternation. We may also use regular expressions to describe some languages that have an infinite number of strings. To achieve that, we use a function we call the *kleene star*, named after the aforementioned American mathematician Stephen Cole Kleene.

If `p` is a regular expression, then `p*` is the smallest superset of `p` that contains ε (the empty string) and is closed under the concatenation operation. Huh? Read that definition again, then try this definition: It's the set of finite length strings that can be created by concatenating strings that match the expression `p`. If `p` can match any string other than ε, then `p*` will match an infinite number of possible strings.

The real name of the typographic symbol (glyph) that denotes the kleene star is an *asterisk*, a Greek (not geek) word that means, appropriately enough, *little star*. Normally, this glyph has five or six arms, but its original purpose was to describe birth dates in a family tree, and it had seven arms.

This is a very popular symbol. In Unicode, loads of asterisk variants are adorned with interesting names, for example, Heavy Teardrop-Spoked Pinwheel Asterisk and Balloon-Spoked Asterisk.

Many fields have assigned their own meanings to the asterisk. In typography, it means a footnote. In musical notation, it may mean that the piano's sustain pedal should be lifted. On early cell phones, we used the asterisk to navigate menus in touch-tone systems. Thus, no worldwide consensus interpretation of the asterisk `*` exists, but in this book, it always means the function *kleene star*.

Do you want to see a simple example? Of course, you do! The concatenation closure of one single symbol—for example, `a`—is this:

- `/a*/ = { ε, a, aa, aaa, … }`.

Want to see a more academic example? The concatenation closure of the set comprising solely the empty string ε is—♫ ta–da♫—well, the set comprising solely the empty string, that is, `ε* = ε`.

Want to see a more complicated example?

- `/(1|0)*/ = { ε, 0, 1, 01, 10, 001, 010, 011, … }`.

Yay, it'll match every possible binary string, that is, a language comprising infinitely many strings.

Can you write a regular expression that matches all binary strings that contain at least one zero? Or all binary strings with an even number of ones?, I urge you to do some trial-and-error in IRB.

The *kleene star* function is pronounced KLAYnee star/*'kleɪni: star*/. (Don't think that the last thing in that sentence was a regular expression, even though it's flanked by the oblique slanting line punctuation mark we usually call a *slash*. It happens to be a representation of speech sounds in written form. But wait, it's actually a regular expression with symbols from the IPA alphabet, and it'll match nothing but the exact string: *'kleɪni: star*).

The kleene star function is unary, that is, it takes only one operand. The operand is the regular expression to the left, which allows us to say that it's a postfix operator. The operand comes first, and the operator (the asterisk) comes last. It takes precedence over concatenation and alternation, and it's associative. The latter means that if two operators of the same precedence are competing, then the operator closest to the operand wins. Considering that `p**` = `p*`, we say that the kleene star is idempotent. I want to emphasize again that `p*` = `(p|)*`, that is, the empty string ε is always present in a closure. We'll see later that a very common—and disastrous—source of software bugs is to forget that important fact.

Here are some possible answers to the questions I gave you:

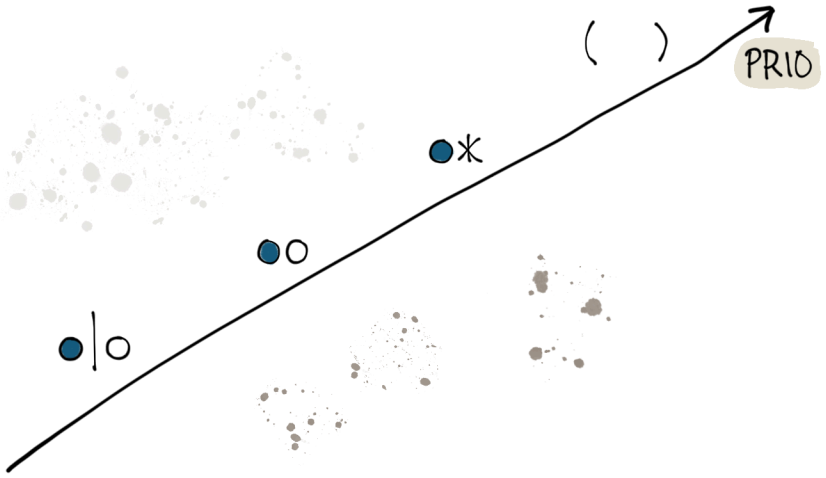
```
'110'.match /(0|1)*0(0|1)*/ #=> #<MatchData "110">
   # all strings with at least one zero
'1111'.match /(0|1)*0(0|1)*/ #=> nil
'1001'.match /((10*1)|0*)*/ #=> #<MatchData "1001">
   # all strings with an even number of ones
'11001'.match /((10*1)|0*)*/ #=> #<MatchData "1100">
''.match /((10*1)|0*)*/ #=> #<MatchData "">
   # even empty string has even number of ones
'1001'.match /((10*1)|0)|/ #=> #<MatchData "1001">
   # again 'an even number of ones'
'11001'.match /((10*1)|0)|/ #=> #<MatchData "11">
''.match /((10*1)|0)|/ #=> #<MatchData "">
'1'.match /((10*1)|0)|/ #=> #<MatchData "">
'010'.match /((10*1)|0)|/ #=> #<MatchData "0">
'01'.match /((10*1)|0)|/ #=> #<MatchData "0">
```

The positive closure operation + and the *at least n* operation `{n,}` are abstractions for expressing infinite concatenation. We will examine them in more detail later in this book.

Two Takeaways

- The kleene star function, denoted by an asterisk `*`, allows matching zero or more repetitions of the preceding expression.
- The kleene star function is a unary postfix operator and has the highest precedence among the regular expression operators discussed so far.

Precedence



You might be tempted to read the following regular expression as “(third or fifth) row.”

```
'fifth row'.match /third|fifth row/
   #=> #<MatchData "fifth row">
'third row'.match /third|fifth row/
   #=> #<MatchData "third">
```

Unfortunately, as you can see, it’s more like either “third” (only) or “fifth row” because of something called *order of operations* or *operator precedence*. The invisible operator for concatenation takes precedence over the alternation operator `|`.

To oil these wheels, we now add parentheses to our three operators. In a regular expression, the subexpression enclosed in parentheses gets the highest priority.

```
'fifth row'.match /(third|fifth) row/
   #=> #<MatchData "fifth row">
'third row'.match /(third|fifth) row/
```

```
   #=> #<MatchData "third row">
```

Note that the parentheses are metacharacters, not literals. They won't match anything in the subject string.

You've probably guessed by now that we also may nest parentheses.

```
'third row'.match /(third|(four|fif)th) row/
   #=> #<MatchData "third row">
'fourth row'.match /(third|(four|fif)th) row/
   #=> #<MatchData "fourth row">
'fifth row'.match /(third|(four|fif)th) row/
   #=> #<MatchData "fifth row">
```

We must understand three things to predict in what order and with what operands the regular expression automata will execute the operators:

- **Operator precedence** is an ordered list of all operators. It tells us which operator must be executed before another operator in a regular expression. Several operators can have the same priority. In mathematics, the terms inside parentheses have the highest priority. Multiplication and division have a lower priority, and addition and subtraction have the lowest. This is why 6+6/(2+1) = 8.
- **Operator position** indicates where the operands are located in relation to the operator. The position can be *prefix*, *infix*, or *postfix*. If the operator is prefix, then the operand resides to the right of the operator, as the mathematical unary minus sign in, for example, -3. An infix operator has an operand on each side, as in addition, for example, 1+2. Finally, a postfix operator stands to the right of its operand, as the exclamation point that represents the faculty operator, for example, 5!.
- **Operator associativity** tells us how to group two operators on the same precedence level. An infix operator can be right-associative, left-associative, or non-associative. In mathematics, the infix operations addition and subtraction have the same precedence. Considering that both are left-associative, the following equation holds: 1–2+3 = (1–2)+3 = 2. Prefix or postfix operators are either associative or non-associative. If they're

associative, we start with the operator closest to the operand. A non-associative operator can't compete with operators of the same precedence.

Here's a table of the regular expressions operators we've studied so far:

Priority	Operator	Symbol	Position	Associativity
1	Kleene star	*	Postfix	Associative
2	Concatenation	*none*	Infix	Left
3	Alternation	|	Infix	Left

If you find that difficult to remember, then try to memorize the mnemonic *SCA*, which stands for Star-Concat-Alter, that is, the order of precedence in regular expressions.

Two Takeaways

- Operator precedence in regular expressions determines the order in which operations are performed, with kleene star having the highest precedence, followed by concatenation, and then alternation.
- Parentheses can be used to override the natural operator precedence.

Some Examples With *, |, and ×

Currently, we have three operators and a small framework. After all this theory, you might wonder whether it's possible for us to solve any problems. Of course we can.

All binary strings with no more than one zero:

```
'01101'.match /1*(0|)1*/ #=> #<MatchData "011">
'0111'.match /1*(0|)1*/ #=> #<MatchData "0111">
'1101'.match /1*(0|)1*/ #=> #<MatchData "1101">
'11010'.match /1*(0|)1*/ #=> #<MatchData "1101">
```

All binary strings with at least one pair of consecutive zeroes:

```
'101001'.match /(1|0)*00(1|0)*/ #=> #<MatchData "101001">
'10101'.match /(1|0)*00(1|0)*/ #=> nil
'1010100'.match /(1|0)*00(1|0)*/ #=> #<MatchData "1010100">
```

All binary strings that have no pair of consecutive zeroes:

```
'1010100'.match /1*(011*)*(0|)/ #=> #<MatchData "101010">
'101001'.match /1*(011*)*(0|)/ #=> #<MatchData "1010">
'0010101'.match /1*(011*)*(0|)/ #=> #<MatchData "0">
'0110101'.match /1*(011*)*(0|)/ #=> #<MatchData "0110101">
```

All binary strings ending in 01:

```
'110101'.match /(0|1)*01/ #=> #<MatchData "110101">
'11010'.match /(0|1)*01/ #=> #<MatchData "1101">
'1'.match /(0|1)*01/ #=> nil
'01'.match /(0|1)*01/ #=> #<MatchData "01">
```

Remember that concatenation takes precedence over alternation when we match all binary strings *not* ending in 01.

```
'010'.match /(0|1)*(0|11)|1|0|/ #=> #<MatchData "010">
'011'.match /(0|1)*(0|11)|1|0|/ #=> #<MatchData "011">
''.match /(0|1)*(0|11)|1|0|/ #=> #<MatchData "">
'1'.match /(0|1)*(0|11)|1|0|/ #=> #<MatchData "1">
'01'.match /(0|1)*(0|11)|1|0|/ #=> #<MatchData "0">
'101'.match /(0|1)*(0|11)|1|0|/ #=> #<MatchData "10">
```

All binary strings that have every pair of consecutive zeroes before every pair of consecutive ones:

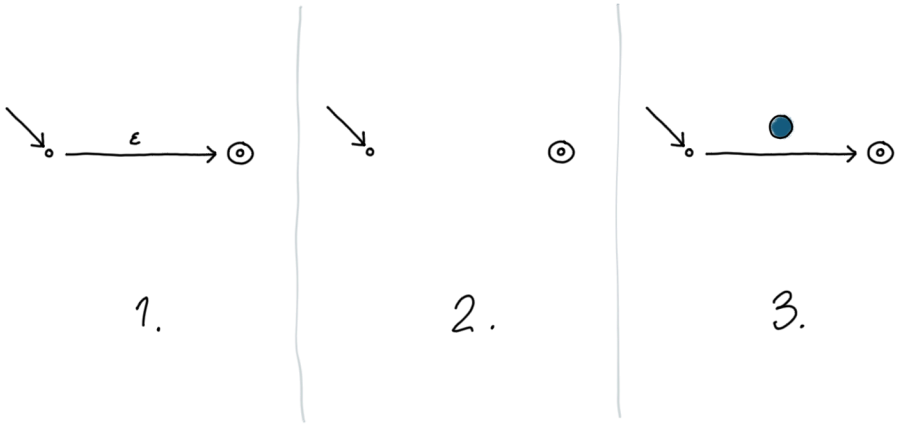
```
'0110101'.match /0*(100*)*1*(011*)*(0|)/
   #=> #<MatchData "0110101">
'00101100'.match /0*(100*)*1*(011*)*(0|)/
   #=> #<MatchData "0010110">
'11001011'.match /0*(100*)*1*(011*)*(0|)/
   #=> #<MatchData "110">
'1100'.match /0*(100*)*1*(011*)*(0|)/
   #=> #<MatchData "110">
'0011'.match /0*(100*)*1*(011*)*(0|)/
   #=> #<MatchData "0011">
```

See if you can find even better regular expressions that solve these problems. Remember that there's an infinite number of synonyms for each regular expression.

Two Takeaways

- Kleene star, alternation, and concatenation can be combined to create complex regular expressions.
- There can be multiple (or even infinite) regular expressions that match the same set of strings.

Regular Expressions Are Finite Automata



For each regular expression—and I mean the two operations, the function, and the six recursive rules style—a finite automaton accepts exactly the same strings as the regular expression. Considering that this isn't a university mathematics textbook, I'll show you some inductive reasoning on this and not a formal proof.

Thus, the hypothesis is that for an arbitrary regular expression `p`, we can create a finite automaton that has exactly one start state, no transitions toward the start state, no transitions leaving the accept state, and that accepts the exact same strings matched by `p`.

We'll need the following three automata:

1. **The empty string ε** is a regular expression corresponding to a finite automaton with a start state, a transition that accepts the empty string ε and leads from the start state to an accept state. We'll call this an *ε-transition*.

2. **The empty set ∅** is equivalent to a regular expression that can't match any single string—not even the empty string ε. It's the same as a two-state automaton, with no single transition. One state is the start state, and the other is the accept state. Unfortunately, they're not linked.
3. **A regular expression that only matches one specific symbol**, for example a green dot 🟢, corresponds to a finite automaton with two states: start and accept. There's a transition from start to accept, and this transition only accepts the specific symbol, for example, green dot 🟢.

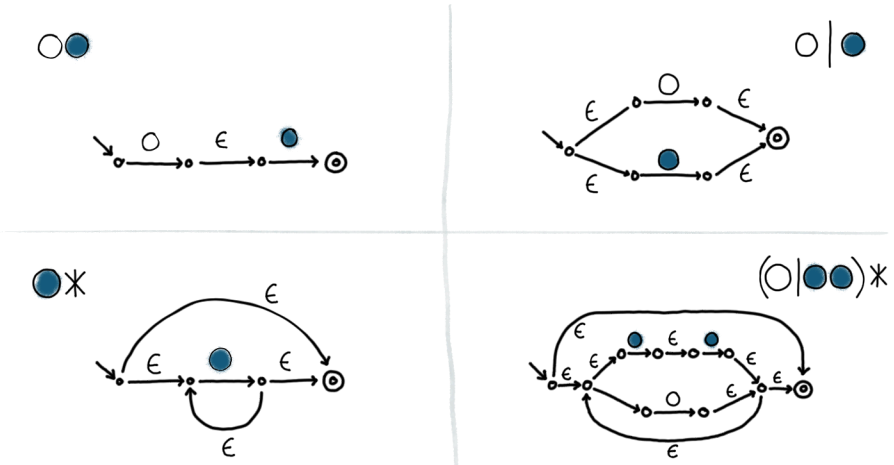
All these three finite automata have two states. One is the start state and the other is the accept state. The differences between these three automata are that the first one has an ε-transition from start to accept, the second has no transition, and the third has a 🟢-transition.

These three automata are our building block types, and now we'll try to use them as lego pieces to build the concatenation, alternation, and concatenation closure.

Imagine that we have two regular expressions, `p` and `q`, corresponding to two finite automata, `s` and `t`, respectively:

- **Concatenation** of `p` and `q` means that we first match a string with `p`, directly followed by a string matched with `q`. To create this finite automaton, we first add ε-transitions from every accept state in `s` to the start state of `t`, then we remove the accept status from all accept states in `s`. We also withdraw the start status of the start state in `t`, that is, `s` cannot accept a string, and `t` cannot start reading. However, `s` and `t` now are connected, and we may start in `s` and accept in `t`. In the top left graph of the following image, this is the method we use to transform the two regular expressions, `/◯/` and `/🟢/`, into `/◯🟢/`.
- **Alternation** of `p` and `q`, that is, `p|q`, is like a finite automaton with a new start state that has ε-transitions to all start states of `s` and `t`. The new finite automaton also has a new accept state reached with ε-transitions from all accept states of `s` and `t`. Thus, the former start and accept states of `s` and `t` aren't start and accept states in our new automaton. In this way, we transform the two regular expressions, `/◯/` and `/🟢/`, into `/◯|🟢/` in the top right graph of the following image.

- **Kleene star** is the concatenation closure. Assume that `p = q*`. Then `s` is the finite automaton that we get when we take `t` and add two states and four transitions as follows: One new state is the start state, and the other is an accept state. All accept states of `s` lose their accept status and instead get an ε-transition to the new accept state. We add two ε-transitions from the new initial state—one to the old start state and one to the new accept state. Furthermore, we insert one ε-transition from each of the old accept states to the old start state. Using this method helps us transform the regular expression `/●/` into `/●*/` in the lower left graph of the following image.



How to design finite automata for concatenation (top left), alternation (top right), kleene star (bottom left), and a combined pattern (bottom right).

Now look again at the first three graphs in this picture. Take your time, then take a deep breath and ponder whether you can transform any arbitrary regular expression into a finite automaton. Finally, assess the bottom right graph in the image, in which the regular expression `/(○|●●)*/` is depicted using the methods we just explored. Does it feel reasonable?

Two Takeaways

- For any regular expression, a finite automaton can be constructed that accepts the same strings as the regular expression.
- The construction of a finite automaton from a regular expression can be done inductively, by combining smaller automata corresponding to subexpressions.

Traits

Now that we've created a small programming language, that is, regex, and are comfortable with all the theory, we can examine what separates this language from many other programming languages. Three special, but not unique, regex characteristics are that it's declarative, it's a domain-specific language, and it (usually) has no whitespace, code comments, or separators.

Declarative programming languages express logic in a calculation without specifying its control flow. In regex, we declare what the outcome is, but not how the computer can get there. This can be compared with imperative languages such as C, Java, C#, and Ruby, in which we instruct the computer to first do this, then do that, and finally we inspect the results. Thus, in regex, we describe a pattern that the solution must meet, for example, *"the string begins with a one, which is followed by one or more zeroes."* By understanding how a regex automaton behaves, for example, how backtracking works under the hood, we may still write the regex that requires minimum energy and time from the computer.

Domain-specific languages (DSLs) are designed to solve a specific type of problem. SQL is a domain-specific language for querying relational databases, VHDL is a specialized language for describing hardware, and regex is a domain-specific language for describing sets of strings. The complement to DSL is general-purpose programming languages—for example, C, Java, C#, and Ruby—which are designed to solve problems found in most applications. Regex isn't suitable for solving every type of problem; it's even beyond what it, by all means, can do. However, when it comes to describing a set of strings, it's phenomenal.

Separators and space are present in most programming languages. We use separators, such as semicolons, to signal that a statement is complete. Most programming languages also allow the programmer to add extra white space in strategic places in the code. It's not unusual for the programmer to write code comments in the middle of the code. Neither separators, whitespace, nor code comments are really meaningful to the computer. They don't add, remove, or change

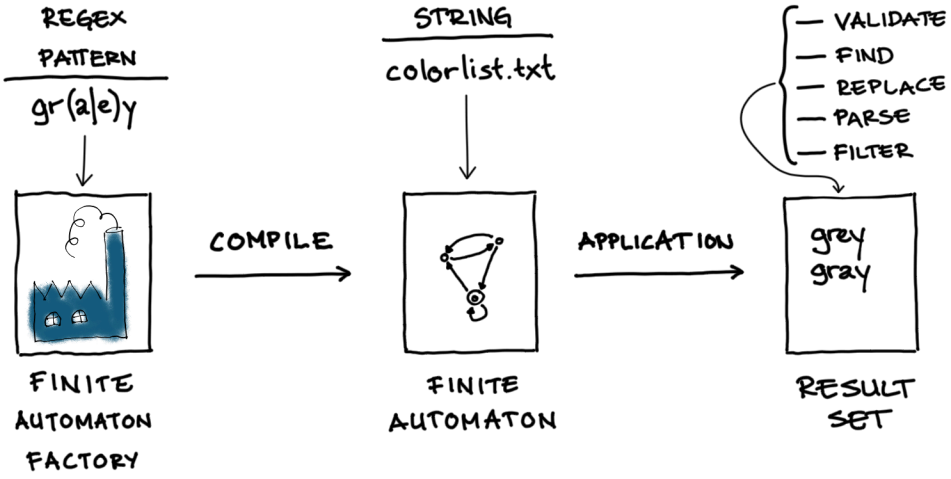
any computer calculations, but they may make the code more readable for us humans. In regex, no redundant characters exist. Everything has a meaning. The code comprises literals, metacharacters, and operators. In some regex dialects, you can add an extra flag to get permission to insert code comments and whitespace, but in the normal case, this isn't possible.

Finally, a few words about dialects. *"The nice thing about standards is that there are so many of them to choose from"* is a quote that has been attributed to many celebrities. Almost every regex dialect has its own syntax and semantics. They're mostly the same, but certain details differ. The biggest difference among the dialects is which operators are actually available, that is, it's not enough to find a regex on the web. It must be compatible with the regex dialect in which you intend to use it.

Two Takeaways

- Regular expressions are declarative, meaning they describe the pattern to be matched without specifying the control flow.
- Regular expressions are domain-specific languages designed specifically for describing sets of strings.

Architecture



The normal regex usage flow comprises three steps:

1. Compile a regex pattern.
2. Feed input into an application.
3. Iterate the matched results.

Sometimes the three steps are combined in a macro, and sometimes you need to go through all three steps, one by one.

(1) In the first step, our carefully written regex pattern is compiled into an automaton. If we store a reference to our compiled automaton in a variable, we may reuse the automaton as many times as we want. Here's how to compile in IRB:

```
fa = Regexp.compile('10*') #=> /10*/
```

(2) Our shiny automaton, referred to by the variable `fa`, now offers several services. It can validate, find, replace, parse, and filter text that we feed it. We select a service

and feed the automaton with input. In IRB, we can, for example, use the method `scan` to find all instances of our pattern:

```
rs = '1 100 00 10'.scan fa #=> ["1", "100", "10"]
```

(3) Now we can iterate over our result set:

```
rs.each { |x| puts x }
   # 1
   # 100
   # 10
   #=> ["1", "100", "10"]
```

Usually, we don't need to perform each step, **(1)**, **(2)**, and **(3)**, explicitly. Instead, we may compile them implicitly, like this:

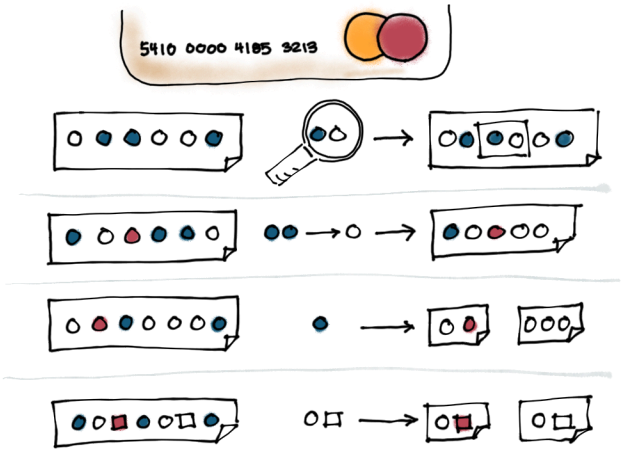
```
'1 100 00 10'.scan(/10*/) { |x| puts x }
   # 1
   # 100
   # 10
   #=> "1 100 00 10"
```

Note that our regex pattern was compiled on the fly in this last example.

Two Takeaways

- The typical workflow for using regular expressions involves compiling the regex pattern into an automaton, feeding input to the automaton, and iterating over the matched results.
- Some regex dialects allow implicit compilation, where the regex pattern is compiled on the fly when it is used.

Functions



Verify (for example a card number), find, replace, filter, and parse are five regex applications.

In IRB, it's good enough to write a regex between two slashes to get the regex compiled implicitly into an automaton. That puts us in a position in which we may invoke many different functions on our automaton object.

(1) We may *verify* that the input matches a specific pattern:

```
/^ab*$/ === 'abbb' #=> true
/^ab*$/ === 'baaa' #=> false
```

(2) We may *find* a pattern in the input:

```
'ab b abb a ba'.scan /ab*/ #=> ["ab", "abb", "a", "a"]
```

(3) We may *replace* text in the input that matches a pattern:

```
'ab b abb a ba'.gsub(/ab*/, '¤') #=> "¤ b ¤ ¤ b¤"
```

(4) We may *filter* out a pattern from the input:

```
'ab b abb a ba'.gsub(/ab*/, '') #=> " b   b"
```

(5) And we also may *parse* a pattern from the input:

```
'ab b abb a ba'.split /ab*/ #=> ["", " b ", " ", " b"]
```

Two Takeaways

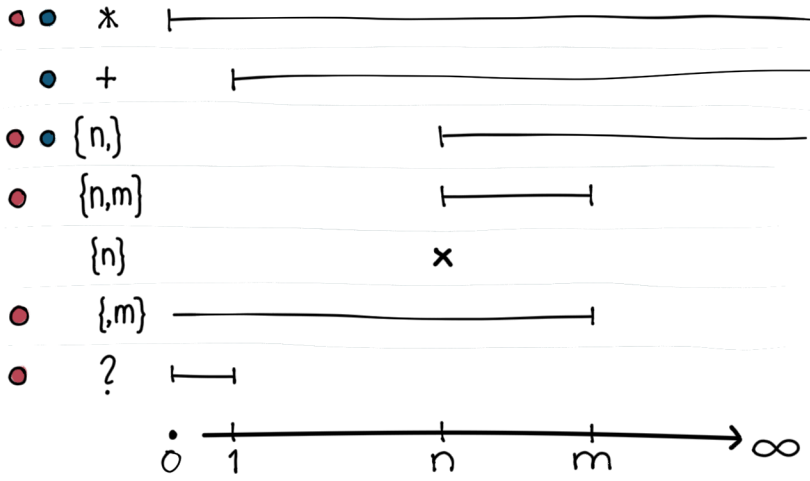
- Regexes can be used for a variety of functions, including verifying, finding, replacing, filtering, and parsing text.
- The specific functions available and their syntax may vary depending on the regex dialect

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.

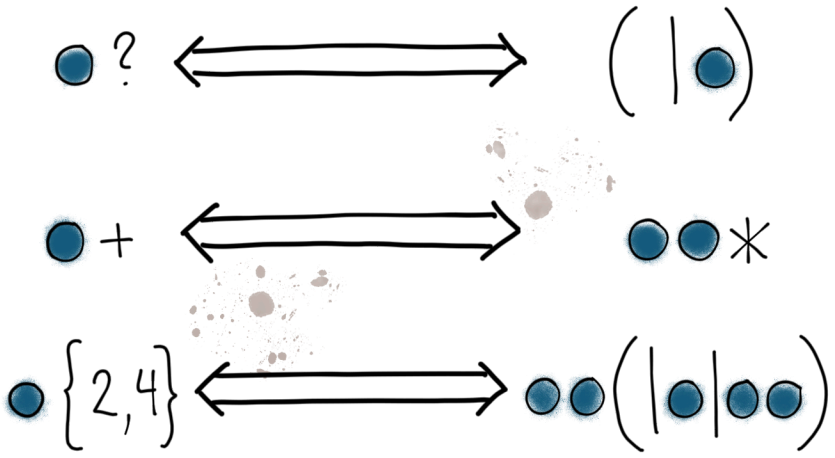
The first column (red dots) indicates that *kleene star function* `*`, *optional quantifier function* `?`, and *generic quantifier function* with first argument zero `{0, m}` or omitted `{, m}` may match zero instances.

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*.

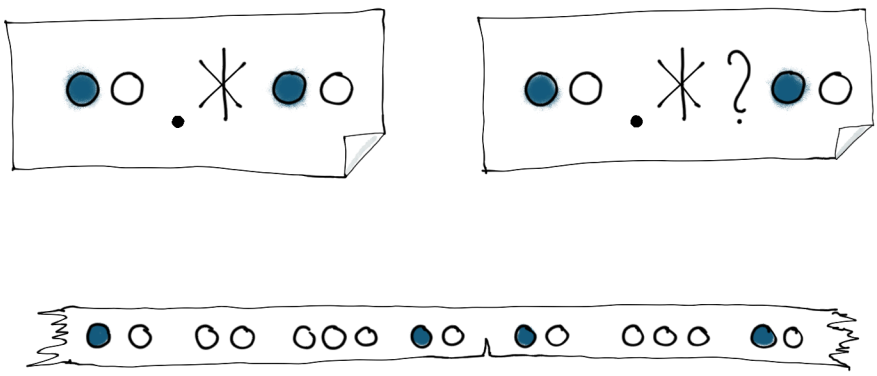
Here are some more equations. Inspect them to ensure that you agree with me that they're true.

- `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.

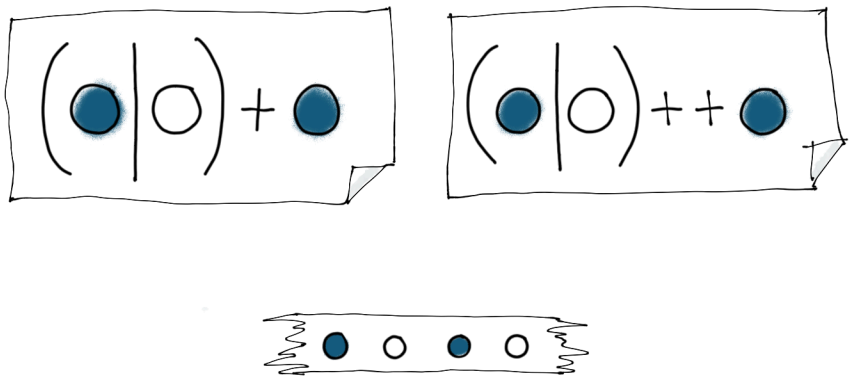
```
'<div>a</div><span>c</span><div>b</div>' \
   .scan /<div>.*?<\/div>/
   #=> ["<div>a</div>", "<div>b</div>"]
'aa'.match /a?/ #=> #<MatchData "a">
'aa'.match /a??/ #=> #<MatchData "">
'aaaaa'.match /a{2,4}/
   #=> #<MatchData "aaaa">
   # at least 2, at most 4, as much as possible
'aaaaa'.match /a{2,4}?/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

Possessive Quantifier



Greedy and reluctant (lazy) quantifiers find the same matches. Considering that you always ask the quantifier for just *one* match, *greedy* and *reluctant* choose the first one they find. The difference is the search method they use and, therefore, the order in which they find matches, that is, which match they find first.

Possessive quantifiers will only find a subset of the matches found by *reluctant* and *greedy*; therefore, *possessive* may ignore some perfectly correct matches. If things go badly, a possessive quantifier sometimes returns empty-handed despite the fact that possible matches existed.

So, what value do possessive quantifiers bring? They sometimes come up with a result in less time. The quantifiers we've seen so far use brute force. When they notice that they're on the wrong track, they backtrack, then try the next possible way. It's all these failures—the backtrackings—that waste our time! A possessive quantifier *never, ever* backtracks. It refuses to backtrack. Thus, it'll quickly give you an answer: yes or no.

If possessive quantifiers deliver faster due to the strategy of not backtracking, then they must choose the shortest search path (to maximize consumption of the input string) up front, mustn't they? Is it even possible to know the shortest path up front? The badly kept secret is that possessive quantifiers don't know. They just possess (that word again!) an unusually high dose of self-esteem. Possessive quantifiers use the *greedy* algorithm to consume as much as possible. They won't notice whether a subexpression further to the right in your regex fails simply because Mr. Possessive Operator speculatively consumed too many characters from the input string. You see, this is a hungry rascal who doesn't care about its fellow subexpressions.

Enough is enough—show me some code! The possessive quantifier syntax is a modifier written as a plus, `+`, to the right of the quantifier. All quantifiers can have the possessive modifier: possessive optional `?+`; possessive kleene star `*+`; possessive positive closure `++`; and possessive generic quantifier `{2, 5}+`.

Whenever there's a match in the following code examples, the match is replaced with the character pillow symbol ¤ (also known as *the generic currency symbol*, as you know). Note that the period symbol matches any character except line breaks.

Possessive optional:

```
'b'.sub /a?+b/, '¤' #=> "¤"
'b'.sub /a?b/, '¤' #=> "¤"
'b'.sub /.?+b/, '¤' #=> "b"
'b'.sub /.?b/, '¤' #=> "¤"
'ab'.sub /.?+b/, '¤' #=> "¤"
'ab'.sub /.?b/, '¤' #=> "¤"
```

Possessive kleene star:

```
'b'.sub /a*+b/, '¤' #=> "¤"
'b'.sub /a*b/, '¤' #=> "¤"
'b'.sub /.*+b/, '¤' #=> "b"
'b'.sub /.*b/, '¤' #=> "¤"
'ab'.sub /.*+b/, '¤' #=> "ab"
'ab'.sub /.*b/, '¤' #=> "¤"
```

Possessive positive closure:

```
'b'.sub /a++b/, '¤' #=> "b"
'b'.sub /a+b/, '¤' #=> "b"
'b'.sub /.++b/, '¤' #=> "b"
'b'.sub /.+b/, '¤' #=> "b"
'ab'.sub /.++b/, '¤' #=> "ab"
'ab'.sub /.+b/, '¤' #=> "¤"
```

As of this book's writing, Ruby/IRB doesn't support the possessive quantifier in default mode. Instead, it treats `{m, n}+` as two consecutive quantifiers `{m, n}` and `+`.

```
ruby> 'aab'.sub /.?+b/, '¤' #=> "a¤"
ruby> 'aab'.sub /.{0,1}+b/, '¤' #=> "¤"
   # Warning: nested repeat operators '?' and '+'
   # were replaced with '*' in regular expression: /.{0,1}+b/
```

In Scala, all four types of quantifiers—`?`, `*`, `+`, and `{m, n}`—support the possessive modifier.

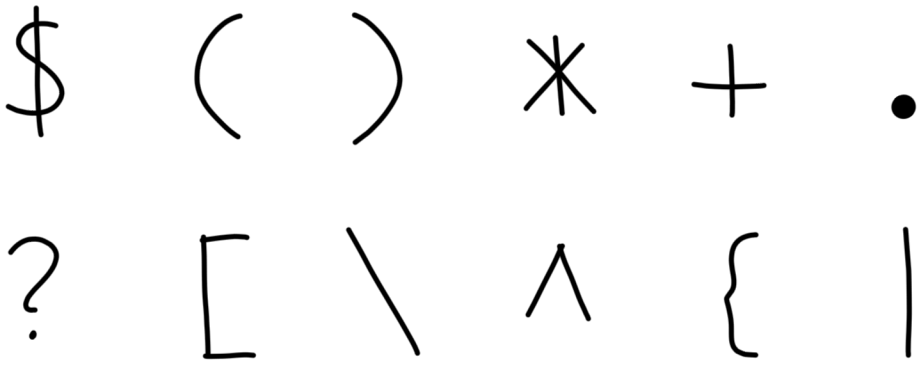
```
scala> ".{0,1}+b".r.replaceAllIn("aab", "¤")
res3: String = a¤
scala> ".?+b".r.replaceAllIn("aab", "¤")
res4: String = a¤
```

You may test Scala code online in Scastie.

Two Takeaways

- Possessive quantifiers, denoted by a plus sign + after the quantifier, refuse to backtrack, which can improve performance but may miss valid matches.
- Possessive quantifiers use the greedy algorithm and do not consider the needs of subsequent subexpressions, potentially leading to missed matches.

Literal vs. Metacharacters



The twelve literals which don't match literally.

Most characters in a regex match literally, that is, you simply search for a text the same way as when you're searching in your word processor.

```
'kadıköy karaköy köyun'.scan /köy/
   #=> ["köy", "köy", "köy"]
```

A dozen characters normally carry special meaning in a regex; thus, they don't match literally.

- **Caret `^` and dollar `$`** assert a position at the beginning or the end.
- **Left and right parentheses `(` and `)`** start and end a capturing group.
- **Asterisk `*`, plus `+`, questionmark `?`, and left brace `{`** repeat the preceding subexpression.
- **Period `.`** matches any character except line breaks.
- **Left square bracket `[`** starts a character class.
- **Backslash `\`** lets a metacharacter be matched literally.
- **Vertical line `|`** is an alternation.

These twelve not-literally-matching characters are called metacharacters. In specific contexts, these characters might have other meanings, yet they don't match literally. Note that the closing square bracket], hyphen -, and closing curly bracket } normally do match literally in regex.

```
'ho hoho ]hohoho'.scan /]ho.*/ #=> ["]hohoho"]
'ho hoho -hohoho'.scan /-ho.*/ #=> ["-hohoho"]
'ho hoho }hohoho'.scan /}ho.*/ #=> ["}hohoho"]
'ho hoho )hohoho'.scan /)ho.*/
   # unmatched close parenthesis: /)ho.*/ (SyntaxError)
```

If you still want to match one of the twelve metacharacters in the previous image literally, you must escape them with a backslash \.

```
'Sentence.'.scan /\./ #=> ["."]
'Sentence.'.scan /./
   #=> ["S", "e", "n", "t", "e", "n", "c", "e", "."]
```

Suppose you get a string from the user. You'd like to search for that string in the id attribute values in some HTML, then show the user the matching results, if any.

```
user_input1 = 'first' #=> "first"
regex = Regexp.compile('id="' + user_input1 + '"')
   #=> /id="first"/
'<span name="secret" id="first"/>'.scan regex
   #=> ["id=\"first\""]
```

Unfortunately, a user realizes that he can make a regex injection attack.

```
user_input2 = '|name=".*?"|' #=> "|name=\".*?\"|"
regex = Regexp.compile('id="' + user_input2 + '"')
   #=> /id="|name=".*?"|"/
'<span name="secret" id="first"/>'.scan regex
   #=> ["name=\"secret\"", "id=\"", "\""]
```

Luckily, most programming languages that support regex have a function that neutralizes all metacharacters. In Ruby, it's called `Regexp.escape()`.

```
user_input2 = '|name=".*?"|' #=> "|name=\".*?\"|"
regex = Regexp.compile('id="' + Regexp.escape(user_input2) + '"')
   #=> /id="\|name="\.\*\?"\|"/
'<span name="secret" id="first"/>'.scan regex #=> []
```

In some regex dialects—for example perl—it's possible to escape with a `\Q` to `\E` block. It works inside, as well as outside, character classes.

```
perl -e 'print "match" if "2.71828" =~ /\Q2.71\E/'
   #=> match
perl -e 'print "match" if "2-71828" =~ /\Q2.71\E/'
   #=> nothing, since '.' is escaped
perl -e 'print "match" if "2-71828" =~ /2.71/'
   #=> match
```

Two Takeaways

- In regular expressions, most characters match literally, but certain characters, such as `^`, `$`, `*`, `+`, and `?`, have special meanings (metacharacters) and do not match literally.
- To match a metacharacter literally, it needs to be escaped with a backslash `\`.

Character Code Points

“Unicode provides a unique number for every character, no matter what the platform, no matter what the program, no matter what the language.”

That’s what the Unicode Consortium writes on its website. The Unicode standard covers the characters from most writing systems of the world—modern and ancient. It also includes technical symbols, punctuation, and many other characters used in writing text. As of this book’s writing, Unicode supports more than 100 scripts and 100,000 characters.

In Part I, I explained alphabets. A regex automaton is equipped with an alphabet, that is, an ordered list of all possible input characters it understands. To understand what a character is, we first must sort out some other concepts:

- **Grapheme**: The smallest units of meaning in written language, for example, letters or numbers, are called graphemes. A single grapheme can be visualized in different ways, that is, the grapheme for the number seven can be crossed or not crossed. Thus, grapheme is something semantic, but not anything visual.
- **Glyph**: The different ways we can visualize a grapheme on paper or on screen are called glyphs. A glyph is a shape associated with an abstract character. For example, the grapheme for the Ohm sign may have a glyph that looks like this: Ω. Several different glyphs may represent the same grapheme.
- **Code Point**: A code point normally is assigned to an abstract character, which is a level above encodings of bits and bytes. Think of it as an index in an array of characters. In Unicode, the Ohm sign code point is U+2126, which is hexadecimal. Thus, the decimal index is 8486.
- **Character encoding**: A pairing of codes, for example, bitmaps or natural numbers and characters, is called character encoding. The most popular encodings are UTF-8, UTF-16, and UCS-2. Unicode can be implemented with

different encodings. The ohm sign is encoded as `0xE2 0x84 0xA6` in UTF-8 and as `0x2126` in UTF-16.

Most, but not all, regex dialects support Unicode. The easiest way to match a character is by writing it literally.

```
'a ; a - a , Ω . a : A'.scan /Ω/ #=> ["Ω"]
```

Of course, you also can match a Unicode code point.

```
'a ; a - a , Ω . a : A'.scan /\u2126/ #=> ["Ω"]
```

After `\u` comes four hexadecimal digits that add up to a code point number, but the syntax varies. Some regex dialects limit the number of digits in the hexadecimal number to one, two, four, or eight, while other dialects allow unlimited large numbers. Some have `\x`, while others have `\u`, like the previous code example. In some dialects, you actually write `\x{2126}` to match code point `U+2126`. With the `\x{}` syntax, you can choose how many digits you want.

- Python: `\xDF`
- Perl: `\xD`, `\xDF`, and `\x{2126}`
- Java and C#: `\xDF` and `\u2126`

It's also possible to point out a code point with octal numbers. This syntax varies as well between regex dialects. It may be one, two, three, and even four digits. The common denominator is that in all cases, it starts with a backslash `\`, directly followed by numbers. Some regex dialects (not Ruby) require the first digit to be zero.

```
'123 ABC'.scan /\101/ #=> ["A"]
'123 ABC'.scan /\63/ #=> ["3"]
'123 ABC'.scan /\063/ #=> ["3"]
```

The `\c` syntax allows you to address any of the first 26 characters in the ASCII character set. First, there's a `\c`, which is followed by a character from `A` to `Z` in the English alphabet. For example, considering that newline is the tenth character in the

character set in my Linux computer terminal, I can reference newline with `J`, which is the tenth letter of the English alphabet. The code then will be `\cJ`.

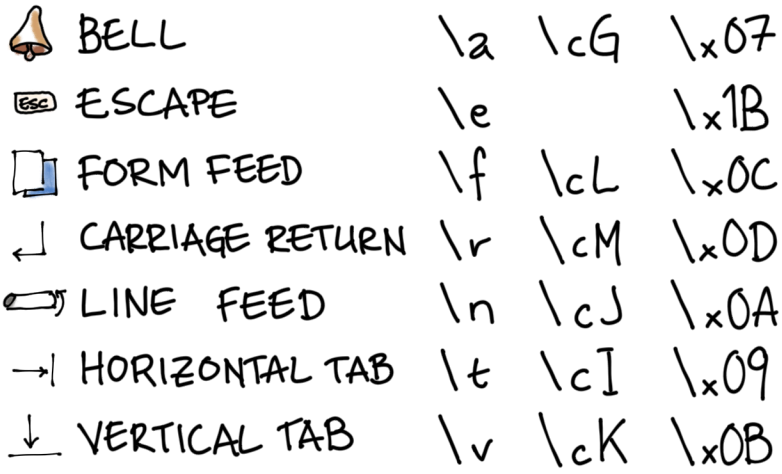
```
'123 45
6'.scan /\d\cJ\d/ #=> ["5\n6"]
```

Depending on the regex dialects that you're using and also what character encoding the input text uses, the pairing of characters and code points may vary. The first 128 code points are almost always the same, and if you use Unicode, everything, of course, is standard. It's not like herding cats, but you must be careful.

Two Takeaways

- Unicode assigns a unique code point (number) to every character, and some regex dialects allow matching characters using their Unicode code points, for example, `\u2126` for the Ohm Ω sign.
- Different regex dialects may have different syntax for specifying Unicode code points, for example, `\u`, `\x`, `\x{}`.

Character Aliases



The easiest way to match a character is, of course, literally. However, in some cases, it's not possible. For example, how do you match the escape sign `\`, the one that usually has code point 27? And is there a way to match compound characters, for example, the German ß, also called double-s?

Seven invisible (that is, they're not associated with any glyph) characters have their own aliases, each of which pairs a backslash `\` with an intuitive letter.

- `\a`—Sound alert, ASCII index `0x07`
- `\e`—Escape-character, ASCII index `0x1B` (Visible as `\.`)
- `\f`—Form feed, ASCII index `0x0C`
- `\n`—Newline, ASCII index `0x0A` on Windows/Linux and `0x0D` on OSX
- `\r`—Carriage return, ASCII index `0x0D` on Windows/Linux and `0x0A` on OSX
- `\R`—Any line break and also `CRLF` as a combo
- `\t`—Horizontal tab, ASCII index `0x09`
- `\v`—Vertical tab, ASCII index `0x0B`

Newline and carriage return are, of course, not the same.

```
'123 45
6'.scan /\d\n\d/ #=> ["5\n6"]
'123 45
6'.scan /\d\r\d/ #=> []
```

Matching line breaks may be tricky because conventions for how they are coded differs between operative systems. Sometimes `\R` comes to the rescue.

```
"a\nb a\r\nb a\n\rb a\rb".scan /a\rb/ #=> ["a\rb"]
"a\nb a\r\nb a\n\rb a\rb".scan /a\nb/ #=> ["a\nb"]
"a\nb a\r\nb a\n\rb a\rb".scan /a\Rb/
   #=> ["a\nb", "a\r\nb", "a\rb"]
```

An odd feature is POSIX's *collating sequences*. If a locale defines composite characters, for example, the German *ß* or Spanish *ll*, as a character, with its own place in the local alphabet, then you can match the combination with the syntax `[[.span-ll.]]`. In the Spanish alphabet, ll lies between l and m, and in the German alphabet, ß lies between s and t. Collating sequences also may be used in ranges; see the Generic Character Classes section of this book.

Speaking of odd POSIX features, I mustn't forget to mention *character equivalents*. In some locales, the two characters with and without diacritics are viewed as equivalent. If you want to match the Spanish ñ, as well as the ordinary `n`, then the *character equivalents* feature provides the syntax `[[=n=]]`. Another example is `[[=a=]]`, which (in some locales) matches `a`, à, and á.

> Diacritical marks like accents and umlauts have a long history, dating back to ancient Greece (or even earlier) where they were used to indicate pronunciation. They spread to other languages, including Latin, which heavily influenced the Romance languages. That's why you see accents in languages like Spanish and French. They inherited them from their ancestor, Latin. While you might not always see diacritics in old Latin texts, they were definitely used as far back as the first century AD.

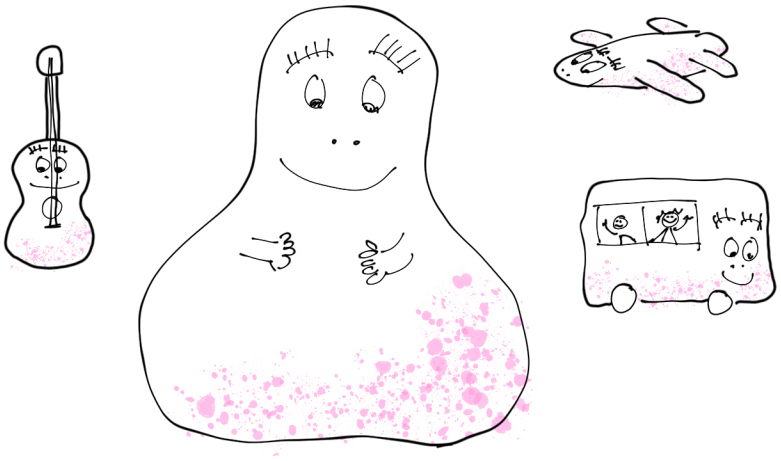
When I say that collating sequences and character equivalents are *odd features*, I don't mean they're bad or useless—just that they're rare. Collating sequences and character equivalents can be used only inside *generic character classes*. The outer pair of square brackets in the previous example are actually the generic character class boundaries.

Some composite glyphs can be coded both as single code points and as pairs of diacritical marks and main characters. For example, `à` can be encoded either as `U+00E0` or as the sequence `U+0061 U+0300`. Some regex dialects have a flag that makes both of these representations equivalent, that is, *canonically equivalent*.

Two Takeaways

- Some non-printable characters, like newline `\n` and escape `\e`, have aliases in regex.
- POSIX defines collating sequences, for example, `[[.span-ll.]]` and character equivalents, for example, `[[=n=]]` for matching composite characters or characters with diacritics.

The Period Symbol



The period symbol matches any character except line breaks.

Do you remember Barbapapa from Annette Tison and Talus Taylor's children's books and films from the 1970s? The hero was a pink, pear-shaped guy with the ability to take on almost any shape whatsoever. The equivalent in regex is *the period symbol*, that is, the punctuation mark sometimes called *dot* or *full stop*.

The period symbol is a character class—that is, a generic character. Rather than writing a literal character, like `2`, `a`, or `#`, we may use the period symbol to specify that we accept *almost* any character.

```
'mama 2 ##'.gsub /a|2|#/, '¤' #=> "m¤m¤ ¤ ¤¤"
'mama 2 ##'.gsub /./, '¤' #=> "¤¤¤¤¤¤¤¤¤"
```

However, two human problems surface when we use the period symbol, and they should be noted:

1. The character class period `.` and the kleene star function `*` are together and separately the most abused features of regex. If you use them frequently

without thinking, you'll often end up with overly general regexes—sometimes they are even incorrect. Every time you intend to write period `.`, asterisk `*`, or even the combo `.*`, you first may want to consider whether you really mean something more specific.

2. Most regex books, including the most popular ones, are unclear or even entirely incorrect, as they claim that *"period symbol matches any character."* In most cases, the period symbol matches *"any character except line breaks*," which is a very, very important difference.

Why doesn't the period symbol normally match line breaks? The original implementations of regex operated line by line. Programs like grep digest one line at a time, and trailing line breaks are filtered out before processing. Thus, no line breaks are left.

NASA engineer Larry Wall created Perl In the 1980s—the programming language that evangelized regex more than anything else. The original purpose was to make report processing easier. Thus, what could be more natural than continuing on the path of line-oriented work?

Another argument sometimes heard is the following: If the period symbol would match line breaks, then the meaning of the idiomatic combo `.*` would change. Perl set the agenda, and now, a few decades later, we only can accept that the period symbol typically doesn't match line breaks, no matter what you and I believe is natural and consistent.

```
"grey gr y gr\ny gr\ry gray".scan /gr.y/
   #=> ["grey", "gr y", "gr\ry", "gray"]
```

Can we force the period symbol to match line breaks? Yes, by setting a flag. Unfortunately, this flag has different names in different regex dialects.

In Perl, it's called *single-line mode*. Imagine what happens if the period symbol matches all characters, including line breaks. Input data becomes a long line in which the line break is a character like any other—hence, the name. Single-line mode should not be confused with what in Perl is called *multi-line mode*. Multi-line mode affects anchors `$` and `^`, and it's orthogonal with single-line mode.

To add more confusion, Ruby uses the term *multi-line* to mean what in Perl is called single-line mode. And what Perl calls multi-line mode is mandatory in Ruby—with no flag available. The best approach to this mess is to call this flag *period-match-all* no matter how it's written syntactically in a specific regex dialect. By the way, in Ruby, we add `m` next to the regex literal when we want the period symbol to match any character.

```
"grey gr y gr\ny gray gr\ry".scan /gr.y/
   #=> ["grey", "gr y", "gray", "gr\ry"]
"grey gr y gr\ny gray gr\ry".scan /gr.y/m
   #=> ["grey", "gr y", "gr\ny", "gray", "gr\ry"]
```

In some regex dialects, most notably JavaScript's, no flag exists for *period match all*. A workaround is to replace the period symbol with the idiom `[\s\S]`. This idiom matches exactly one character—either white space or anything that isn't whitespace. These two classes are, of course, 100% of all the characters—including line breaks.

```
JavaScript> 'grey gr y gr\ny gray gr\ry'
   .match(/gr.y/g);
[ 'grey', 'gr y', 'gray' ]
JavaScript> 'grey gr y gr\ny gray gr\ry'
   .match(/gr[\s\S]y/g);
[ 'grey', 'gr y', 'gr\ny', 'gray', 'gr\ry' ]
```

I argued previously that the period symbol often is abused. What does this mean? Imagine that we want to find every time string in a text, and we have the following requirements:

- Time should always include hours and minutes, sometimes even seconds.
- Hours, minutes, and seconds should always be written with two digits.
- We don't have to ignore impossible numbers, such as minute 61.
- In between hours, minutes, and seconds, there should be one of the separators period `.` or colon `:`.

The result of the simple regex `/\d\d.\d\d(.\d\d)?/` might surprise you.

```
'12:34 09.00 24.56.33'.scan /(\d\d.\d\d(.\d\d)?)/
   #=> [["12:34 09"], ["00 24.56"]]
```

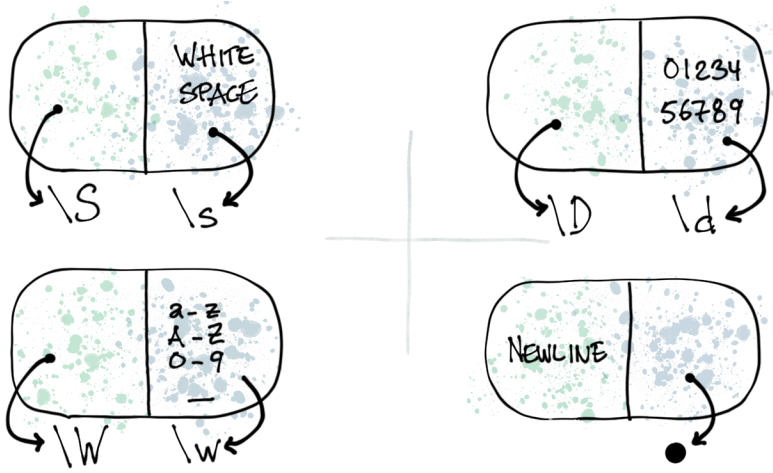
That's not what we wanted. The period symbol matches space! If we replace the item with the more specific character class `[.:]`, we aim more closely at our target. You mustn't forget that the period symbol inside a character class means that we literally want to match the period character.

```
'12:34 09.00 24.56.33'.scan /(\d\d[.:]\d\d([.:]\d\d)?)/
   #=> [["12:34"], ["09.00"], ["24.56.33"]]
```

Two Takeaways

- The period symbol `.` in regex matches any character except line breaks, but it can be made to match all characters in some dialects using a flag, for example, single-line mode in Perl.
- The period symbol, especially when combined with the kleene star in the idiomatic `.*`, should be used cautiously as it can lead to overly general or incorrect matches.

Shrthnd



Four character classes are so frequent that they have their own separate shorthand, that is, their own aliases. One of these four is the period symbol, which means *match any single character that isn't a line break*. But you knew that already. The other three—`\d`, `\s`, and `\w`—also have aliases for their complementary character classes, that is, the complement class matches everything not included in this particular character class. Thus, seven character classes exist overall:

- Period symbol `.` matches any character except line breaks.
- `\s` matches any whitespace character, such as tabs, line breaks, and spaces.
- `\S` matches any character that's not a whitespace; it's the complement of `\s`.
- `\d` matches any digit, for example, 3 or 9.
- `\D` matches any character that isn't a digit; it's the complement of `\d`.
- `\w` means *word character* and is normally equivalent to the character class `[a-zA-Z0-9_]`, but some regex dialects also include letters and digits from other scripts, for example, å or ñ.
- `\W` matches any character that's not matched by `\w`.

The character classes `\w` and `\W` rarely are what you really want and should be used carefully. They must be around for backward compatibility reasons. If your regex dialect supports Unicode, then the *letter* class `\p{L}`, *digit* class `\p{N}`, *not-a-letter* class `\P{L}`, and *not-a-digit* class `\P{N}` are preferred. More about this later in the Unicode categories section of this book.

It's usually pretty straightforward to use a shorthand character classes.

```
'L8 love 2 u 4-ever'.scan /\d/ #=> ["8", "2", "4"]
```

Here are some other examples:

```
'123def789 müde'.scan /\d/
   #=> ["1", "2", "3", "7", "8", "9"]
'123def789 müde'.scan /\D/
   #=> ["d", "e", "f", " ", "m", "ü", "d", "e"]
'123def789 müde'.scan /\w/
   #=> ["1", "2", "3", "d", "e", "f", "7", "8", "9",
   #    "m", "d", "e"]
'123def789 müde'.scan /\W/ #=> [" ", "ü"]
'123def789 müde'.scan /\s/ #=> [" "]
'123def789 müde'.scan /\S/
   #=> ["1", "2", "3", "d", "e", "f", "7", "8", "9",
   #    "m", "ü", "d", "e"]
'123def789 müde'.scan /./
   #=> ["1", "2", "3", "d", "e", "f", "7", "8", "9",
   #    " ", "m", "ü", "d", "e"]
```

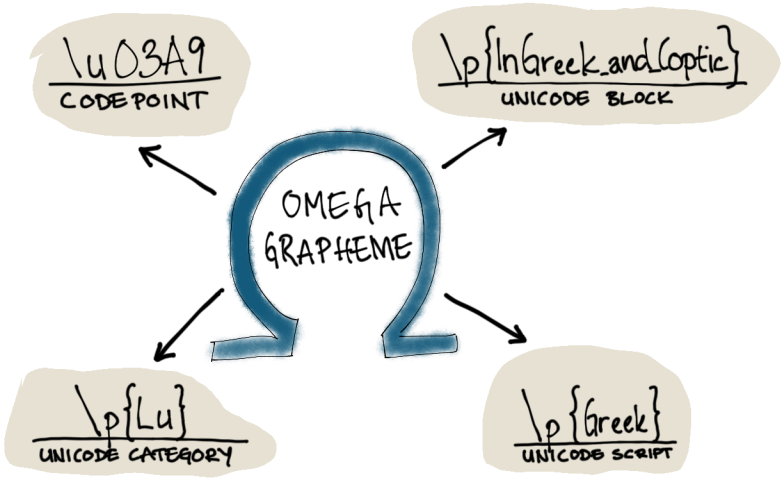
And, as noted many times in this book, the period symbol doesn't normally match line breaks.

```
"gr\ny".scan /./ #=> ["g", "r", "y"]
"gr\ny".scan /\s|\S/ #=> ["g", "r", "\n", "y"]
```

Two Takeaways

- Several shorthand character classes exist in regex, including `\d` for digits, `\s` for whitespace, and `\w` for word characters.
- Each of these three shorthand character classes has a complement, for example, `\D` for non-digits and `\S` for non-whitespace.

Unicode Categories, Scripts, and Blocks



We can divide Unicode code points in three ways. Each code point is a member of one category (also called property). They also belong to a block, and if they aren't unassigned, they belong to a script.

Let's start with *categories*. Unicode is divided into seven categories.

- Other: `\p{C}`
- Letter: `\p{L}`
- Mark: `\p{M}`
- Number: `\p{N}`
- Punctuation: `\p{P}`
- Symbol: `\p{S}`
- Separator: `\p{Z}`

```
'a; a-a, Ω. a: A€ŧ¥'.scan /\p{L}/
```

```
   #=> ["a", "a", "a", "Ω", "a", "A"]
'a; a-a, Ω. a: A€₺¥'.scan /\p{P}/
   #=> [";", "-", ",", ".", ":"]
'a; a-a, Ω. a: A€₺¥'.scan /\p{S}/
   #=> ["€", "₺", "¥"]
'a; a-a, Ω. a: A€₺¥'.scan /\p{Z}/
   #=> [" ", " ", " ", " "]
```

The seven categories are subdivided further into more than 30 subcategories. The syntax calls for adding an extra letter to the subcategory, for example, `l` for lowercase. Here are some examples of subcategories:

- Letter, Lowercase `\p{Ll}`
- Punctuation, Dash `\p{Pd}`
- Symbol, Currency `\p{Sc}`

```
'a; a-a, Ω. a: A€₺¥'.scan /\p{Ll}/
   #=> ["a", "a", "a", "a"]
'a; a-a, Ω. a: A€₺¥'.scan /\p{Lu}/
   #=> ["Ω", "A"]
'a; a-a, Ω. a: A€₺¥'.scan /\p{Sc}/
   #=> ["€", "₺", "¥"]
'a; a-a, Ω. a: A€₺¥'.scan /\p{Sm}/
   #=> []
```

Another decomposition, completely orthogonal to the categories, is *scripts*. When I wrote this book, Unicode contained roughly 70 ancient and 90 modern scripts, and more scripts are in the melting pot for encoding. All unassigned code points belong to exactly one script—no more, no less.

```
'a; a-a, Ω. a: A€'.scan /\p{Greek}/
   #=> ["Ω"]
'a; a-a, Ω. a: A€'.scan /\p{Latin}/
   #=> ["a", "a", "a", "a", "A"]
```

The third way to divide Unicode is in blocks. A *block* is a range of code points. Over 300 blocks already exist.

- `U+0600 - U+06FF == \p{InArabic}`
- `U+0000 - U+007F == \p{InBasic_Latin}`

In some regex dialects, the syntax is `\p{Is …}` instead of `\p{In …}`. Note that even unassigned code points can be part of a block.

By typing a capital `P`, we negate the properties, scripts, and blocks.

```
'a; a-a, Ω. a: A€'.scan /\P{Ll}/
   #=> [";", " ", "-", ",", " ", "Ω",
   # ".", " ", ":", " ", "A", "€"]
'a; a-a, Ω. a: A€'.scan /\P{Latin}/
   #=> [";", " ", "-", ",", " ", "Ω",
   # ".", " ", ":", " ", "€"]
```

You may recall that I warned you about the word character class `\w`, which doesn't always match our naïve expectations on what a word character could be. If our regex dialect supports Unicode, it's more likely that we're looking for, for example, all numbers and letters—even the letters not in the sequence `a-z`.

```
'a; a-a, Ω. a: A4€'.scan /[\p{L}\p{N}]/
   #=> ["a", "a", "a", "Ω", "a", "A", "4"]
```

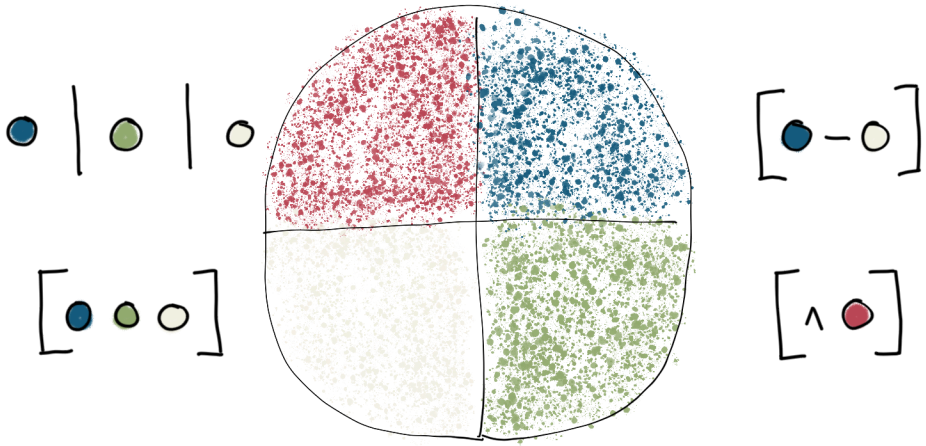
By now, you most certainly know that the period symbol normally matches anything except line breaks. A Unicode counterpart can be found in some regex dialects. It's written `\X`, and it even matches line breaks.

```
perl -e '"gr\ny" =~ /\X/; print $&'
   # g
perl -e '"gr\ny" =~ /\X+/; print $&'
   # gr
   # y
```

Two Takeaways

- Unicode characters are categorized into groups like letters `\p{L}`, numbers `\p{N}`, punctuation `\p{P}`, and symbols `\p{S}`, which can be used in regexes for more precise matching.
- Besides categories, Unicode characters are also organized into scripts (for example, Greek, Latin) and blocks (ranges of code points) providing further options for targeted matching in regexes.

Generic Character Class



Alternation (top left), character class (bottom left),
character class range (top right), and negated character class (bottom right).

Imagine that we must find all the vowels in a text. A no-nonsense approach is to simply list them in a long alternation.

```
'istanbul constantinople'.scan /a|e|i|o|u|y/
   #=> ["i", "a", "u", "o", "a", "i", "o", "e"]
```

What if we have a few (or many) characters and want to ensure that exactly one of these characters—no more, no less—is matched? In regex, we may use a more concise way to write exactly that: the *generic character class*. A *character class* is a designated subset of the *alphabet* (do you remember what an alphabet is?). We describe our subset inside a pair of square brackets `[]`. The description can be written in several ways. The plain syntax is simply to list the characters we want to include in our subset.

```
'istanbul constantinople'.scan /[aeiouy]/
   #=> ["i", "a", "u", "o", "a", "i", "o", "e"]
```

This was more crisp than when we made a verbose alternation in the previous example, but alternation is exactly what it means, that is, the exclusive disjunction or XOR in programmer lingo.

Here's another example, in which we find all letter characters in a hexadecimal number:

```
'12d343ea3'.scan /[abcdef]/ #=> ["d", "e", "a"]
```

Now, `a` to `f` comprise a *range*, an unbroken chain. We describe a range by writing the start character of the range, the end character, and in between them, a hyphen (minus).

```
'12d343ea3'.scan /[a-f]/ #=> ["d", "e", "a"]
```

What's a range then? In regex, the alphabet—all possible characters that may occur in a text—is ordered. As you know, each individual character is assigned a number that we call a *code point*, which also may be called an index. Each code point is unique, that is, multiple characters cannot have the same code point. You can think of the complete set of characters as an array in which each entry has an array index. Note that not only the letters are characters, for example, numbers, punctuation, and white spaces are characters as well. A range is a sequence of characters in our alphabet array. If this alphabet array is Unicode or ASCII, then the English lower letters `a`, `b`, `c`, etc. comprise a range. The digits `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, and `9` comprise another range.

Consider whether you understand the following:

```
'john.smith@company.com'.scan /[7-d]/
   #=> ["@", "c", "a", "c"]
```

In this case, we used the Unicode character database. The matched letters have these code points in Unicode:

- `7` has code point 55 in Unicode.
- `@` has code point 64 in Unicode.
- `d` has code point 100 in Unicode.

Why did the `@` symbol match, but not the period symbol? Because the period symbol has code point 46. When we enter the range 7 to d, we really mean all characters with a code point greater than or equal to 55 and less than or equal to 100.

We may mix ranges and individual characters.

```
'john.smith@company.com'.scan /[j7-dh]/
   #=> ["j", "h", "h", "@", "c", "a", "c"]
```

If the hyphen comes first or last in the character class, it's matched literally, of course.

```
'pera-beyoğlu'.scan /[a-z]/
   #=> ["p", "e", "r", "a", "b", "e", "y", "o", "l", "u"]
'pera-beyoğlu'.scan /[-az]/ #=> ["a", "-"]
'pera-beyoğlu'.scan /[az-]/ #=> ["a", "-"]
```

Though sometimes forgotten, it's worth noting that a character class matches *exactly* one character, that is, not only *max one*, but also *at least one*. When we want to find all names that end with a digit, this isn't the correct regex:

```
'quick2 ball5 good4you1 money1'.scan /[a-z]+[0-9]/
   #=> ["quick2", "ball5", "good4", "you1", "money1"]
```

This is highly disappointing, considering that we actually wanted to get the whole `good4you1` in a single match. However, it's easily fixed, of course.

```
'quick2 ball5 good4you1 money1'.scan /[a-z0-9]+[0-9]/
   #=> ["quick2", "ball5", "good4you1", "money1"]
```

Two Takeaways

- A generic character class in regex, denoted by square brackets `[]`, allows matching a single character from a defined set of characters, for example, `[aeiou]` for vowels.
- Character classes can include ranges `[a-z]`.

Generic Character Class Negated and Tweaked

Sometimes it's easier to say what you don't want. It's been decades since I ate meat, and I have no allergies or food aversions. In response to the question of what I eat, it's easier to say *"anything except meat"* than to say *"pasta, fruit, olives, nuts, bread, beer, baklava..."* etc. Regex character classes support exactly that functionality: negated character classes.

Say you want to find all `class` attributes and their values in HTML code. Our first attempt looks good:

```
'<span rel="info" class="people">'.scan /class=".*"/
   #=> ["class=\"people\""]
```

Great! Unfortunately, there's another HTML snippet in which the attributes `class` and `rel` were swapped.

```
'<span class="people" rel="info">'.scan /class=".*"/
   #=> ["class=\"people\" rel=\"info\""]
```

We can solve this problem with negated character classes. A negated character class begins with a caret, `^`. We want to match exactly one character—anything except those we enumerate in our character class.

```
'<span class="people" rel="info">'.scan /class="[^"]*"/
   #=> ["class=\"people\""]
'<span rel="info" class="people">'.scan /class="[^"]*"/
   #=> ["class=\"people\""]
```

Of course, there may be several characters in a negated character class, and none will match.

```
'<span class="people" rel="info">'.scan /class=['"][^'"]*"/
   #=> ["class=\"people\""]
```

Using `[^"]*`, we express that we want to find zero or more—as many as possible—of anything except straight quotes.

> The metal type with quotation marks was first made in the mid-1500s. By the 1600s, printers were using them a lot. In the Baroque and Romantic periods, some printers even put quotation marks at the start of every line of a long quote. Eventually, they stopped doing this, but they kept the empty margin on the left side of the quote. This is how we got the indented block quotes that we use today.

Note that the caret needs to start the character class. Otherwise, the caret is matched literally, and no classes will be negated.

```
'Kadıköy^Chalcedon^Χαλκηδών'.scan /[^A-Za-z]/
   #=> ["ı", "ö", "^", "^",
   # "Χ", "α", "λ", "κ", "η", "δ", "ώ", "ν"]
'Kadıköy^Chalcedon^Χαλκηδών'.scan /[A-Z^a-z]/
   #=> ["K", "a", "d", "k", "y", "^", "C", "h",
   # "a", "l", "c", "e", "d", "o", "n", "^"]
```

Some regex dialects have three more types of character classes:

- A **character class union** is written as a nested character class. For example, `[[a-f][0-9]]` will match any lower hexadecimal digit.
- A **character class intersection** adds the meta-sequence double-ampersand, `&&`, to find what's common in the two sets: `[\w&&[\da-z]]`.
- **Character class subtraction** has at least two different syntaxes. In some regex dialects, we type a hyphen, `-`, before the set we want to subtract. The expression `[\da-z[g-z]]` will match any lower hexadecimal digit. In other regex dialects, an intersection is written as the main set and the

complement of what we want to remove (do you remember De Morgan's laws?): `[\da-z&&[^g-z]]`

The set operations union, intersection, and subtraction are a bit obscure. They only exist in a few regex dialects, and it might be even more difficult to find applications for them. However, applications do exist even though the previous examples are more esoteric.

Two Takeaways

- Negated character classes match any single character except those listed within the square brackets `[]` after the caret symbol `^`.
- Some regex dialects support character class unions (nested brackets), intersections (`&&`), and subtractions (`-` or `&&[^]`) for more complex set operations within character classes.

Generic Character Class Escape

The five literals which do not always match literally inside a character class.

Look at these two expressions. Why are the results so different?

```
'I know that 3 - 2 is 1'.scan /[a-z]/
   #=> ["k", "n", "o", "w", "t", "h", "a", "t", "i", "s"]
'I know that 3 - 2 is 1'.scan /[-az]/
   #=> ["a", "-"]
```

In the first case, the character class includes a range. In the second case, it's just a list of three different characters—a hyphen `-`, `a`, and `z`. All characters except five can be matched literally at any position inside a character class, that is, we must be careful with the following five inside a character class:

- A **slash, `\`,** indicates that the next character is a metacharacter, for example, newline `\n` or the digit character class shorthand `\d`.

- A **caret,** `^`**,** creates a negative character class—the complement of the characters listed—if the caret is positioned first in the character class.
- A **hyphen,** `-`, is put in between two characters to describe a range, which entails the hyphen is matched literally if positioned first or last in the character class.
- A **right square bracket,** `]`**,** marks the end of a character class.
- A **left square bracket,** `[`**,** marks the beginning of a character class union, subtraction, or intersection—but only if our regex dialect supports that functionality.

All other characters are matched literally inside character classes, even the period symbol.

```
'This is. That is.'.scan /[t.]/ #=> [".", "t", "."]
```

When we want to match `\`, `^`, — or `]`, we may prefix them with a backslash `\`. In most regex dialects, we also may write those in a position in our character class in which they cannot possibly have a functional meaning.

```
'1 - 2 ^ 3 \ 4 [ 5 ^ 6'.scan /[3^]/ #=> ["^", "3", "^"]
'1 - 2 ^ 3 \ 4 [ 5 ^ 6'.scan /[^3]/
   #=> ["1", " ", "-", " ", "2", " ", "^", " ", " ", "\\",
   # " ", "4", " ", "[", " ", "5", " ", "^", " ", "6"]
'1 - 2 ^ 3 \ 4 [ 5 ^ 6'.scan /[\^3]/ #=> ["^", "3", "^"]
'1 - 2 ^ 3 \ 4 [ 5 ^ 6'.scan /[]3]/ #=> ["3"]
'1 - 2 ^ 3 \ 4 [ 5 ^ 6'.scan /[3]]/ #=> []
'1 - 2 ^ 3 \ 4 [ 5 ^ 6'.scan /[3\]]/ #=> ["3"]
'1 - 2 ^ 3 \ 4 [ 5 ^ 6'.scan /[-24]]/ #=> []
'1 - 2 ^ 3 \ 4 [ 5 ^ 6'.scan /[-24]/ #=> ["-", "2", "4"]
'1 - 2 ^ 3 \ 4 [ 5 ^ 6'.scan /[2-4]/ #=> ["2", "3", "4"]
'1 - 2 ^ 3 \ 4 [ 5 ^ 6'.scan /[24-]/ #=> ["-", "2", "4"]
'1 - 2 ^ 3 \ 4 [ 5 ^ 6'.scan /[2\-4]/ #=> ["-", "2", "4"]
```

Shorthand character classes, except the period symbol, work inside character classes as well.

```
'1 - 2 ^ 3 \ 4 [ 5 ^ 6'.scan /[\da-z]/
   #=> ["1", "2", "3", "4", "5", "6"]
'1 - 2 ^ 3 \ 4 [ 5 ^ 6'.scan /[\Sa-z]/
   #=> ["1", "-", "2", "^", "3", "\\", "4", "[", "5",
   # "^", "6"]
'1 - 2 ^ 3 \ 4 [ 5 ^ 6'.scan /[\w-]/
   #=> ["1", "-", "2", "3", "4", "5", "6"]
```

As you've seen, some characters are matched literally in regex, and some are interpreted as metacharacters, which is why we must prefix (escape) the latter with a backslash `\` when we want these characters to be matched literally. However, you might have noticed something peculiar: The escape rules *inside* and *outside* the character class differ. We discussed earlier how only five characters are metacharacters *inside* a character class. A dozen characters are interpreted as metacharacters *outside* a character class. Here's a revealing example:

```
'Five $ (dollar) + one gull { *|. ?'.scan /(|)*?.{0}+$/
   #=> [[nil]]
'Five $ (dollar) + one gull { *|. ?'.scan /[(|)*?.{+$]/
   #=> ["$", "(", ")", "+", "{", "*", "|", ".", "?"]
```

Two Takeaways

- Inside a character class, certain characters like `\`, `^`, `-`, and `]` may have special meanings and might need to be escaped with a backslash `\` to be matched literally.
- However, these characters can often be matched literally without escaping if they are placed in a position where they cannot have a special meaning, for example, placing the hyphen `-` at the beginning or end of the class.

Posix Character Class

The POSIX counterpart is older than the Unicode character class constants. It's only about a dozen classes. I say *about* because it varies with different locales and dialects:

- `[:alnum:]` English alphanumeric characters, equivalent to `[a-zA-Z0-9]` or `[\d\p{Ll}]`
- `[:alpha:]` English alphabetic characters, equivalent to `[a-zA-Z]` or `\p{L}`
- `[:ascii:]` The seven-bit ASCII characters, equivalent to `[\x00-\x7F]`
- `[:blank:]` Spaces and tabs, but not line breaks, equivalent to `[ \t]`
- `[:cntrl:]` Control characters, equivalent to `[\x00-\x1F\x7F]`
- `[:digit:]` Digits, equivalent to `\d`
- `[:graph:]` Visible characters, that is, not spaces, control characters
- `[:lower:]` Lowercase alphabetic characters, equivalent to `[a-z]` or `\p{Ll}`
- `[:print:]` Same as `[:graph:]`, but including the space characters
- `[:punct:]` Punctuation characters
- `[:space:]` Whitespace characters, equivalent to `\s`
- `[:upper:]` Uppercase alphabetic characters, equivalent to `[A-Z]` or `\p{Lu}`
- `[:word:]` Word characters, equivalent to `\w`
- `[:xdigit:]` Hexadecimal digits, equivalent to `[A-Fa-f0-9]`

POSIX character classes can be used only inside *generic character classes*.

```
'abc123efg'.scan /[[:digit:]]/ #=> ["1", "2", "3"]
'abc123efg'.scan /[:digit:]/ #=> ["g"]
```

Some dialects redefine POSIX character classes to also include letters in Unicode other than `a-z` as alphabetic characters.

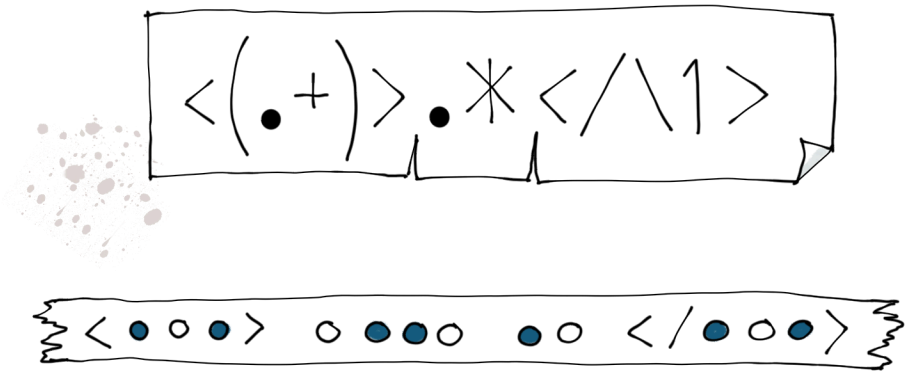
```
'abc123efgå '.scan /[[:lower:]]/
   #=> ["a", "b", "c", "e", "f", "g", "å"]
'abc123efgå '.scan /[\d\p{L}]/
   #=> ["a", "b", "c", "1", "2", "3", "e", "f", "g", "å"]
'abc123efgå '.scan /[[:alnum:]]/
   #=> ["a", "b", "c", "1", "2", "3", "e", "f", "g", "å"]
```

Java supports its own mix of Unicode and POSIX character class syntax. Classes have POSIX names, but the Unicode syntax looks like this: `\p{Graph}`. Of course, these may be used outside generic character classes, as well as inside them.

Two Takeaways

- POSIX character classes, for example `[:digit:]` and `[:alpha:]`, provide named sets of characters for common character types, but they must be used within generic character classes `[]`.
- The exact behavior and availability of POSIX character classes may vary depending on the regex dialect and locale.

Grouping



Parentheses are metacharacters in regex, that is, they're not matched as literals.

```
'12(3)4'.gsub /(\d)\d/, '¤' #=> "¤(3)4"
```

When we want them to match literally, we may escape the parentheses with a backslash `\`.

```
'12(3)4'.gsub /\(\d\)\d/, '¤' #=> "12¤"
```

Above all, we may use parentheses to override the order of operations. Like mathematics, parentheses have the authority to cancel the natural operator precedence in regex. Positive closure `+` takes precedence over concatenation. Parentheses override that rule in the second example below:

```
'1234'.gsub /1\d+/, '¤' #=> "¤"
'1234'.gsub /(1\d)+/, '¤' #=> "¤34"
```

However, in regex, we also may use parentheses to give a subexpression an identity. We count the left parentheses from left to right in our expression and give them indices 1, 2, 3, etc. Here's an analysis of the expression `/a((bc)((d)e))/`:

- Index 1: `(bc)((d)e)`
- Index 2: `bc`
- Index 3: `(d)e`
- Index 4: `e`

These subexpressions—1, 2, 3, and 4—are known as *groups*. Note that a group may contain many characters, a single character, or even be empty.

Two Takeaways

- Parentheses `()` in regexes are primarily used for grouping subexpressions, which can be used to apply quantifiers or alternations to multiple elements as a unit.
- Parentheses also create capturing groups, which allow extracting the matched portion of the input corresponding to the grouped subexpression.

Capture and Back Reference

The groups constructed from the parentheses aren't just esoteric—we may benefit from them. The groups are captured, stored, and can be referenced later.

```
'abcde' =~ /a((bc)((d)e))/ #=> 0
$1 #=> "bcde"
$2 #=> "bc"
$3 #=> "de"
$4 #=> "d"
```

The reference number is based on the order of the group's left parenthesis. Note how they're ordered in this example where nested groups are present.

We even may refer to a group inside our regex. Imagine that you want to find all occasions in which an arbitrary character is repeated.

```
'Rūmiyyat al-kubra'.sub /(.)\1/, "¤"
   #=> "Rūmi¤at al-kubra"
```

The generic regex `/(.)\1/` isn't as cryptic as it may look. The period symbol matches any character that isn't a line break. The parentheses around the period symbol capture whatever was matched by the period symbol in Group №1. With `\1`, we refer back to the thing captured in Group №1. So, we wanted to match precisely the character that the period symbol matched with `\1` no matter what it was.

In the same way, we may match words or parts of words that repeat themselves.

```
'You may may do that'.sub /(\S+)\s\1/, "¤"
   #=> "You ¤ do that"
```

Regex dialects based on deterministic finite automata (DFA) normally cannot capture groups at all, but most regex dialects are based on non-deterministic finite automata (NFA).

It's possible to use parentheses, `()`, and refuse to capture a group. It sometimes may provide a marginally better performance and keep the indices lower. The price we pay is a cluttered regex. The syntax for a non-capturing group is a question mark, `?`, followed by a colon, `:`, immediately after the left parenthesis.

```
'abcde' =~ /a((?:bc)((d)e))/ #=> 0
$1 #=> "bcde"
$2 #=> "de"
$3 #=> "d"
```

You might have noticed the recurring syntax with question marks and some additional characters directly after the left parenthesis. Here are some examples:

- Atomic grouping: `(?>`
- Comments: `(?#`
- Conditional: `(?(`
- Mode modifiers: `(?i:`, `(?s:`, and `(?m:`
- Named capture: `(?<`, `(?P<`, or `(?'`
- Negative Lookahead: `(?!`
- Positive Lookahead: `(?=`
- Positive Lookbehind: `(?<=`
- Negative Lookbehind: `(?<!`

Captured groups can be referenced in the replacement text of a search-and-replace. For example, you can format a date from the U.S. to the more logical ISO 8601.

```
'12/31/1999'.sub %r!(\d\d)/(\d\d)/(\d{4})!, '\3-\1-\2'
   #=> "1999-12-31"
```

The regex lies between the exclamation marks, and '`\3-\1-\2`' is what matches are replaced with. As in this repeated character example, we refer to the three groups with `\1`, `\2`, and `\3`.

A few regex dialects support *forward references*, but most treat them as errors. Some people (not you, of course) find it challenging to understand the following:

```
'aba'.match /(\2b|(a)){2}/ #=> nil
'aab'.match /(\2b|(a)){2}/ #=> #<MatchData "aab">
```

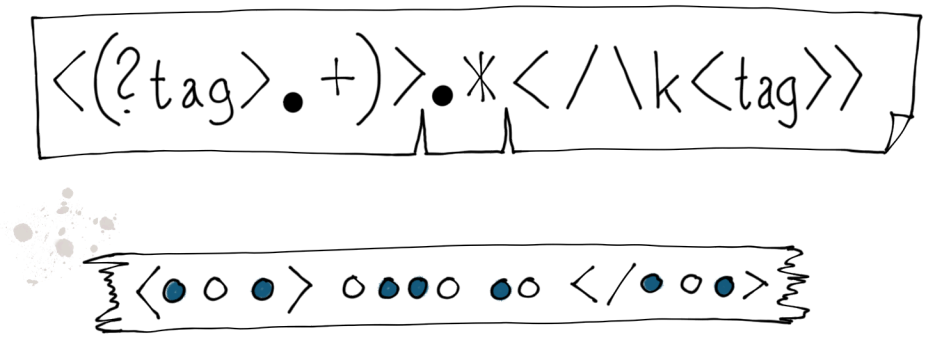
Even more obscure are *relative backreferences*, in which the index is negative.

```
'abb'.match /(a)(b\k<-2>)/ #=> nil
'aba'.match /(a)(b\k<-2>)/ #=> #<MatchData "aba">
```

Two Takeaways

- Capturing groups are numbered based on the order of their opening parentheses, and the captured content can be referenced using backreferences like `\1`, `\2`, etc.
- Backreferences can be used both within the regex itself, for example to match repeated words, and in replacement strings during search-and-replace operations.

Named groups



Even the capture can be named. Eventually, it adds more clarity when we refer back to that more explanatory name later.

```
'12/31/1999'.sub \
   %r!(?<month>\d\d)/(?<day>\d\d)/(?<year>\d{4})!,
   '\k<year>-\k<month>-\k<day>'
   #=> "1999-12-31"
'You may may do that'.sub /(?<word>\S+)\s\k<word>/, '\k<word>'
   #=> "You may do that"
```

The *named capture* syntax varies. Sometimes we name a capture with `(?P<name> …)` or `(?'name' …)`, instead of `(?<name> …)`, as we do above. The backreference part need not be `\k<name>` either. In some regex dialects, it's `(?P=name)`, `\k'name'`, or even `${name}` etc.

Look at this example in C#, which matches vowels preceded by `v` and consonants preceded by `c`:

```
csharp> Regex regex = new Regex("v(?'letter'[aeiouy])|" +
   "c(?'letter'[b-df-hj-np-tv-xz])");
csharp> regex.Match("va");
va
csharp> regex.Match("vb");
csharp> regex.Match("ca");
csharp> regex.Match("cb");
cb
```

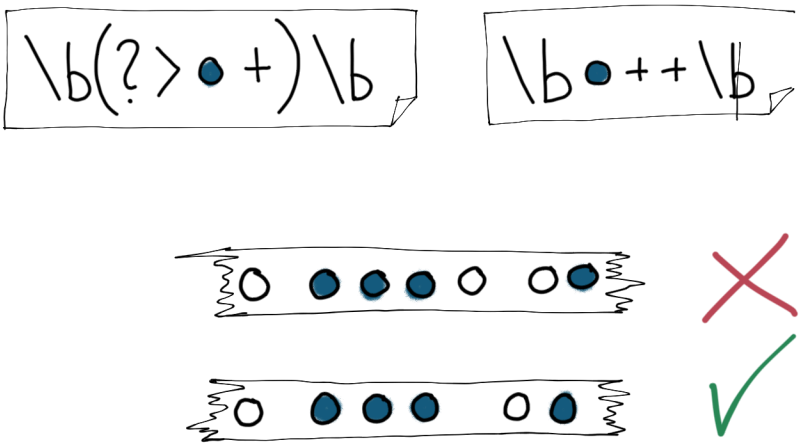
Be careful not to mix numeric and named backreferences. Some dialects ignore named groups whenever groups are numbered. Others count all groups. Note the difference here between Ruby and C#:

```
csharp> Regex.Replace("ab", "(?<first>.)(.)",
   "Group 1 is '$1'");
"Group 1 is 'b'"
ruby> 'ab'.sub /(?<first>.)(.)/, "Group 1 is '" + $1 + "'"
   #=> "Group 1 is 'a'"
```

Two Takeaways

- Named groups allow assigning meaningful names to capturing groups, improving readability and maintainability of regexes, especially when dealing with many groups.
- The syntax for named groups and backreferences may vary across regex dialects, for example `(?<name> …)`, `(?P<name> …)`, `\k<name>`, or `\k'name'`.

Atomic Groups



Although it's not trivial to understand *how* atomic groupings work, it's easier than understanding *why* and *when* to use them. The short answer to *why* is that the regex automaton fails faster. Atomic groups refuse to backtrack, and this sometimes may improve performance.

Here follow two variants of a little tangled regex that searches for strings of at least two `a` followed by one `b`. The first and third expressions contain the atomic grouping `(?> …)`, while the second and fourth don't:

```
'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab'.scan /(?>a+a+)+b/
   #=> ["aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab"]
'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab'.scan /(a+a+)+b/
   #=> [["aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"]]
'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'.scan /(?>a+a+)+b/
   #=> []
'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'.scan /(a+a+)+b/
   #=> []
```

The most peculiar thing about these examples isn't that the last two fail, but that the last one on my computer took 25,000 times longer to execute than the first three. Why? It turns out that no `b` is present at the end of the input, then the regex automaton starts to backtrack. With atomic grouping, we explicitly state that we don't want backtracking. So, the question *why*, is answered with *performance*. Do previous discussions about *possessive quantifiers* ring a bell?

Here's a simple example:

```
'123° 456 789°'.scan /\d+°/ #=> ["123°", "789°"]
```

We're looking for some digits followed by the degree symbol, °. With `123°` and `789°`, it's a perfect match, but what happens with `456` in the middle? The regex automaton first matches `456` with the subexpression `\d+`, then discovers that there's a space instead of the degree symbol directly after `456`. That makes the regex automaton instead only try to match `45` with `\d+`. Considering that `6` isn't a degree symbol either, the automaton now will test to match `\d+` with only `4`. Nor is `5` a degree symbol, so now the automaton finally realizes that this doesn't work.

Suppose we encapsulate `\d+` in an atomic group. The automaton then gives up right away when it detects that `456` isn't followed by a degree symbol `°` because atomic groups refuse to backtrack.

```
'123° 456 789°'.scan /(?>\d+)°/ #=> ["123°", "789°"]
```

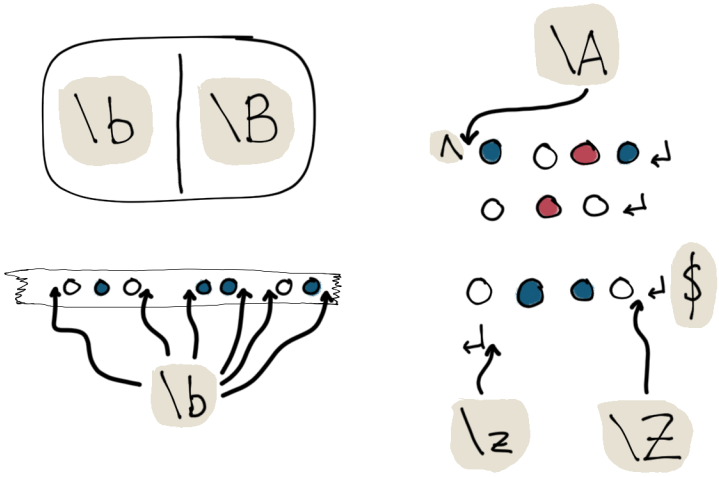
Atomic groups have strong kinship with possessive quantifiers. I said earlier that greedy and reluctant quantifiers match the same text, but in a different order. However, atomic groups match only a subset. It's more efficient when we don't find anything anyway. However, if you're not careful, you might miss something that happens to be correct.

```
'123° 456 789°'.scan /(?>.+)°/ #=> []
```

Two Takeaways

- Atomic groups `(?> …)` prevent backtracking within the group, which can improve performance, especially in cases where backtracking would lead to many failed attempts.
- However, atomic groups should be used with caution, as they may miss valid matches by refusing to backtrack.

Anchors



Here's the big thing with anchors: they don't match any characters in the input string. However, they're still useful because they match a position, that is, their match is of zero length:

- **Caret** `^` matches the position before the first character in the input string and usually also directly after a line break.
- **Dollar** `$` matches the position after the last character in the input string and usually also directly before a line break.
- `\A` matches only at the start of the input string and ignores line breaks.
- `\Z` matches only at the end of the input string and ignores line breaks.
- `\b` matches word boundaries, mostly based on what `\w` matches

Some regex dialects differ slightly from the following. Check this before applying anchors.

Let's start with caret `^` and dollar `$`. Note how an extra newline `\n` affects these patterns:

```
'This to the fair Critias.'.scan /[A-Z][a-z]+/
   #=> ["This", "Critias"]
'This to the fair Critias.'.scan /^[A-Z][a-z]+/
   #=> ["This"]
"This to the fair\nCritias.".scan /^[A-Z][a-z]+/
   #=> ["This", "Critias"]
'This to the fair Critias.'.scan /[A-Z][a-z]+$/
   #=> []
"This to the fair Critias\n.".scan /[A-Z][a-z]+$/
   #=> ["Critias"]
'This to the fair Critias.'.scan /^[A-Z][a-z]+$/
   #=> []
```

Now let's see what happens if we use the same regular expressions, only changing from caret `^` and dollar `$` to `\A` and `\Z`:

```
'This to the fair Critias.'.scan /[A-Z][a-z]+/
   #=> ["This", "Critias"]
'This to the fair Critias.'.scan /\A[A-Z][a-z]+/
   #=> ["This"]
"This to the fair\nCritias.".scan /\A[A-Z][a-z]+/
   #=> ["This"]—no Critias match after line break
'This to the fair Critias.'.scan /[A-Z][a-z]+\Z/
   #=> []
"This to the fair Critias\n.".scan /[A-Z][a-z]+\Z/
   #=> []—no Critias match before line break
'This to the fair Critias.'.scan /\A[A-Z][a-z]+\Z/
   #=> []
```

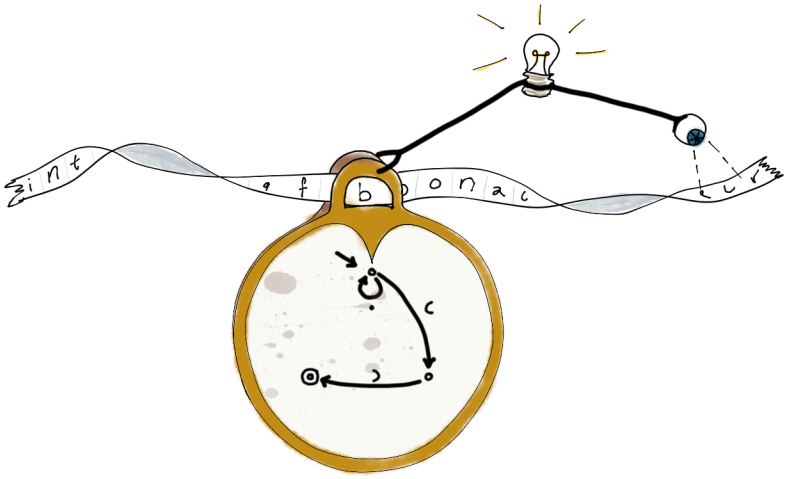
The \b anchor matches word breaks—which sounds useful, but may not always be what you expect. The reason is that in most regex dialects, the definition of a word break \b is based on what's matched by the word character class \w.

```
'This to the fair Critias.'.scan /\w+/
   #=> ["This", "to", "the", "fair", "Critias"]
'This to the fair Critias.'.scan /t\w+/
   #=> ["to", "the", "tias"]
'This to the fair Critias.'.scan /\bt\w+\b/
   #=> ["to", "the"]
```

Two Takeaways

- Anchors in regexes assert positions within the input string without consuming characters, such as the beginning of the string ^, the end of the string $, or word boundaries \b.
- Some anchors, like ^ and $, may behave differently in multi-line mode, where they can match the beginning or end of individual lines within the input.

Lookarounds



"This clock-shaped machine solves the slash-slash problem. The square window at the top of the machine reads a tape—from left to right—consisting of the input string, i.e., the C code. We have a state machine of type Nondeterministic Finite Automaton (NFA). To visualize and represent the automaton, we use a transition graph, in which the vertices represent states and the edges represent transitions. The initial state is identified by an incoming unlabeled arrow not originating at any vertex. The acceptance state is surrounded by a circle. This is the graph that you see printed on the clock dial. Now, whenever a new symbol is read from the tape, the clock dial rotates so that the drooping peak, just below the reading window, points at the current state. Ah, and this particular machine also has a special lookahead feature: it's a long arm with an eye in the end and a light bulb. This eye can look ahead and tell if there's any double slash. If there is, the bulb will glow and the machine will understand that it doesn't matter if the input ends in a pair of parentheses—it's in a comment anyway." Source: De Morgan to the Rescue.

Last, but certainly not least, are lookarounds. They're actually often very useful and sometimes irreplaceable. Like the anchors we just saw, lookarounds don't consume anything from the input string. They differ from the other anchors in that they can match any pattern, asserting that the pattern exists or doesn't exist. The response to that existential doubt is, of course, of boolean type. Either the pattern exists or it doesn't. Nothing from the input string is consumed.

There are four lookaround operators.

- **Positive lookahead** `(?= )` checks whether a pattern is present to the right
- **Negative lookahead** `(?! )` checks whether a pattern isn't present to the right
- **Positive lookbehind** `(?<= )` checks whether a pattern is present to the left
- **Negative lookbehind** `(?<! )` checks whether a pattern isn't present to the left

Note that lookbehinds look a little bit like left arrows pointing forward or backward.

Here's a challenge for the lookahead function: Which words are followed by the word `Thor`?

```
'Cheese slicer invented by carpenter Thor'. scan  \
   /\b\w+?\sThor/
   #=> ["carpenter Thor"] -- too bad, Thor included
'Cheese slicer invented by carpenter Thor'. scan  \
   /\b\w+?(?=\s+Thor\b)/
   #=> ["carpenter"] -- yes, Thor extracted from match
```

Remember that `\w` matches any word letter, and `\b` is the anchor for word breaks.

Which fragments starting with `c` immediately follow a whitespace `\s`?

```
'The cheese slicer invented by carpenter Thor'. scan /c\w+/
   #=> ["cheese", "cer", "carpenter"]
'The cheese slicer invented by carpenter Thor'. scan /\sc\w+/
   #=> [" cheese", " carpenter"]
```

Ouch! We accidentally matched the whitespace preceding our words, which is why we rather use lookbehind.

```
'The cheese slicer invented by carpenter Thor'. \
   scan /(?<=\s)c\w+/
   #=> ["cheese", "carpenter"] - preceded by whitespace
'The cheese slicer invented by carpenter Thor'. \
```

```
scan /(?<!\s)c\w+/
#=> ["cer"] - starting with c, but no whitespace before
```

Now that you're getting good with lookahead and lookbehind, let me finish with a story about Alice and Bob. Together, they were tasked with writing a function that validates a number of properties of a new password:

- At least eight characters long
- At least one uppercase letter
- At least one lowercase letter
- At least one digit
- At least one of the following special characters: `@$!%*?&`
- No other types of characters

Bob quickly whipped up an imperative algorithm.

```
def validate_password (password)
  has_uppercase = false
  has_lowercase = false
  has_digit = false
  has_special_char = false

  password.each_char do |char|
    if char >= 'A' && char <= 'Z'
      has_uppercase = true
    elsif char >= 'a' && char <= 'z'
      has_lowercase = true
    elsif char >= '0' && char <= '9'
      has_digit = true
    elsif "@$!%*?&".include?(char)
      has_special_char = true
    end
  end
```

```
  has_uppercase && has_lowercase && has_digit && has_special_char
end
```

Bob: *“Short and efficient! Right?”*

Alice: *“Does the algorithm check that the password is at least eight characters long?”*

Bob: *“Oh right. I forgot, but I’ll quickly fix it by adding code to the beginning of the function.”*

```
  if password.length < 8
    return false
  end
```

Alice: *“Arbitrary special characters, such as 'ö,' are not allowed in the passwords according to the spec we received. Does the algorithm check for that?”*

Bob: *“No, I forgot that too. Now I’ve added it with…”*

```
def validate_password (password)
  if password.length < 8
    return false
  end

  has_uppercase = false
  has_lowercase = false
  has_digit = false
  has_special_char = false

  allowed_chars = ('A'..'Z').to_a + ('a'..'z').to_a + ('0'..'9').to_a +
"@$!%*?&".chars
```

```
  password.each_char do |char|
    if !allowed_chars.include?(char)
      return false
    end

    if char >= 'A' && char <= 'Z'
      has_uppercase = true
    elsif char >= 'a' && char <= 'z'
      has_lowercase = true
    elsif char >= '0' && char <= '9'
      has_digit = true
    elsif "@$!%*?&".include?(char)
      has_special_char = true
    end
  end

  has_uppercase && has_lowercase && has_digit && has_special_char
end
```

Bob: *“Unfortunately, all the 'if' and 'else' make it difficult to get an overview of the flow. Is there any way to make the code more readable?”*

Alice: *“We could use regex.”*

```
def validate_password (password)
  pattern =
/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}$/
  !!(password =~ pattern)
end
```

Two Takeaways

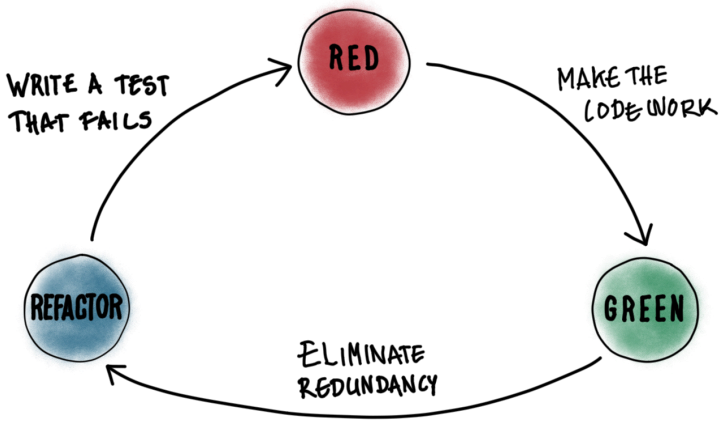
- Lookarounds allow checking for patterns before or after the current matching position without including those patterns in the actual match.
- There are four types of lookarounds: positive lookahead `(?= …)`, negative lookahead `(?! …)`, positive lookbehind `(?<= …)`, and negative lookbehind `(?<! …)`.

Part IV: Test-Driven Regex Development

In this part, you'll learn about a technique called test-driven development (TDD) and how you can benefit from it when developing and maintaining your regexes. TDD helps you write better code, especially when you work with regexes.

With TDD, you ensure that your regexes execute as intended—both today and in the future—and that they are easy to maintain and update. It's a method that I think you'll find very useful, and it'll save you a lot of time and headaches in the long run.

Wishful Thinking and Test-Driven Development



Wishful thinking in programming involves assuming you already have the functions or code snippets to solve parts of a problem, even if you haven't written them yet. For example, you might write calls to an API that doesn't exist. It's a mind hack that gets you to "begin with the end in mind." Instead of getting bogged down in implementation details, you can look at the bigger picture and focus on what the client code of the yet-to-be-written code really needs.

Imagine you need to write a regex that validates that a string seems to be an email address. Even though you won't follow everything in the RFC 5322 standard, there are still several things to check:

- There is exactly one `@` symbol.
- There is at least one period to the right of the `@` symbol.
- There are alphanumeric characters to the left of the `@` symbol.
- Etc.

This creates a list of requirements. For each requirement, we'll write down examples of correct as well as incorrect email addresses. Let's start with examples of "There is exactly one `@` symbol."

- `abc@abc.com` (pass)
- `abc.abc.com` (fail)
- `abc@abc@com` (fail)

Great! Now, let's test these examples in IRB. First, we must make sure that the unit test framework is present.

```
require "test/unit/assertions"
include Test::Unit::Assertions
```

With the test framework, we can test the truth of statements by calling the `assert` function. This is where wishful thinking comes in. We simply pretend—that is, "wish"—that the `is_email` function already exists and that it validates that its input has exactly one `@` symbol.

```
assert is_email('abc@abc.com') #=> undefined method `is_email'
```

Ouch! Or ... good! The message "`undefined method 'is_email'`" tells us that `is_email` doesn't exist, which we already knew. We start by writing the simplest possible `is_email` function to get rid of the "`undefined method`" error.

```
def is_email(email) false end
```

Now we can run our test again.

```
assert is_email('abc@abc.com') #=> <false> is not true
```

Since `is_email` always returns `false`, the test fails and gives us the [logic tautology](#) "`<false> is not true`." The easiest way to make it pass is through a boolean makeover: always return `true` instead of always return `false`.

```
def is_email(email) true end
```

However, now that the function always returns `true`, checking an inadequate address example with `assert_false` will cause problems.

```
assert is_email('abc@abc.com') #=> nil
assert_false is_email('abc.abc.com') #=> <false> expected but was <true>
```

Our positive test—the old one—passed, but the negative test—the new one—fails because `is_email` always returns `true`. Sorry, IRB! We promise to do better by writing an improved implementation of the function.

```
def is_email(email) /.*@.*/ === email end
```

This regex checks if the input string contains an `@` symbol. Now both tests should pass.

```
assert is_email('abc@abc.com') #=> nil
assert_false is_email('abc.abc.com') #=> nil
```

Wow! Both tests pass—that is, return `nil`— and we've just developed a function embryo using wishful thinking and TDD. By isolating our regex into its own function, we can test it—and even regression-test it—without other potential sources of error.

Let's continue with the third example.

```
assert_false is_email('abc@abc@com') #=> <false> expected but was <true>
```

When we adjust `is_email` to make our third test pass, the first two tests must still pass. We can modify the regex to ensure it matches strings with exactly one `@` symbol. Anchors and negated character classes do the trick.

```
def is_email(email) /^[^@]*@[^@]*$/ === email end
assert is_email('abc@abc.com') #=> nil
assert_false is_email('abc.abc.com') #=> nil
assert_false is_email('abc@abc@com') #=> nil
```

Yes! All three tests pass.

In its purest form, TDD consists of three steps, often called Red–Green–Refactor.

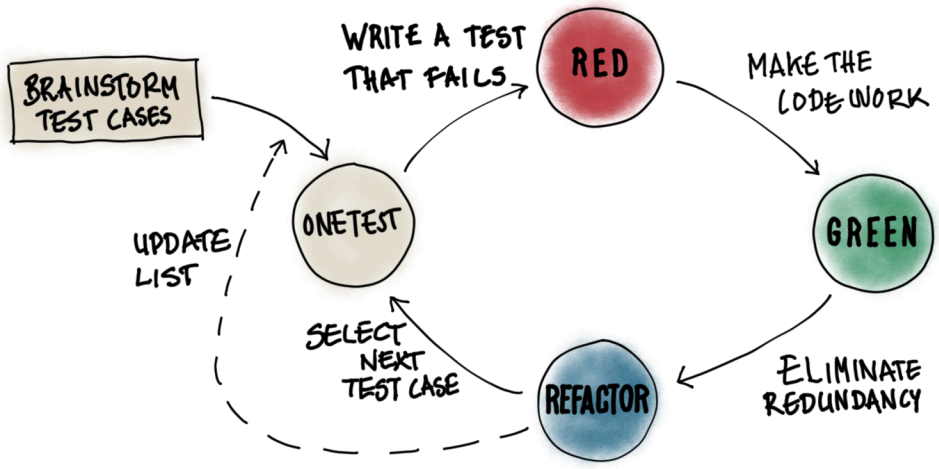
1. **Red:** Write a test that assumes functionality you don't have yet.
2. **Green:** Write the code that makes the test in step 1 pass.
3. **Refactor:** Improve the structure of the old and new code.

If it feels backward to start by writing a test for code that doesn't exist—that is, step 1—you can think of the test as a requirement specification.

Two Takeaways

- In wishful thinking, you assume you have the necessary functions or code snippets to solve parts of a problem, even if you haven't actually written them yet.
- TDD is a method in which you start by writing tests for code that doesn't exist yet, which helps drive a minimal API with needs-driven implementations.

TDD and Regression Testing



Besides driving minimal APIs with needs-driven implementations, TDD also creates the conditions for regression testing. Regression tests are invaluable because they act as a safety net. Imagine that in the future you add new functionality or change existing code. You might accidentally introduce disastrous side effects into other parts of the system that you aren't even aware of. Regression tests catch these side effects before they can cause problems.

Regression tests also give increased confidence to programmers who want to improve code in the future. Refactoring code means changing the code's structure without changing its functionality. With a suite of regression tests, you can be sure that refactoring hasn't introduced any unexpected bugs. These tests ensure that the code still works as it should after the changes.

To make what we have done so far more practical and sustainable, we can collect all these small tests in a test suite. This suite is saved in a version-controlled source code file, just like the production code. This makes it easy to keep track of the tests, share them with other developers, and see how they have evolved over time.

We can also have a dedicated server process that continuously runs all the tests. If any test suddenly fails, the server sends an alarm. This way, we can quickly detect and fix any problems before they affect users.

Red–Green–Refactor is a good foundation for TDD, but to make the process even smoother and more efficient, we can add two more steps to the cycle. The extended cycle then becomes:

1. **Brainstorm test cases:** Before we start coding, we write a list of all conceivable test cases in plain English. We list all the requirements, just like we did with "There is exactly one `@` symbol." We write this list directly in our test file, but we comment it out so that it doesn't affect execution.
2. **Select next test case:** We choose one of the commented-out test cases from the list. We translate the selected test case into code with `assert`, `assert_false`, or any other assertion function. We make sure to include both positive and negative tests to cover different scenarios.
3. **Red:** Now we write the test itself in code, as in the example with `assert is_email('abc@abc.com')`. Since the `is_email` function is not yet correctly implemented, the test will fail (become "red").
4. **Green:** We implement the simplest possible code that makes the test pass (become "green").
5. **Refactor:** We improve and optimize the code while ensuring that all tests still pass.

By adding "Brainstorm test cases" and "Select next test case," we get a more structured and thoughtful process. We start with an overview of all test cases, which helps us capture all the important aspects of functionality. Then we can focus on one test case at a time and work our way through the list. This makes it easier to keep track of what needs to be tested and ensures that we don't miss anything important. We can also add more test cases if we get sudden insights during the TDD cycle.

Two Takeaways

- Regression tests catch bugs that may be accidentally introduced when we add new functionality or change existing code.
- Regression tests give more confidence to programmers who want to improve code in the future by ensuring that refactoring hasn't introduced any unexpected bugs.

TDD and Legacy Code

Inspired by TDD, we can use a similar approach when dealing with legacy code. Imagine finding code in the future that someone wrote long ago.

```
require 'date'

def days_until_date(date_string)
  # This is legacy code, do not modify!
  # The regex below is essential for the algorithm to work correctly.
  match = date_string.match(/(\d+)-(\d+)-(\d+)/)
  raise ArgumentError, "Invalid date format" unless match
  p, q, r = match.captures.map(&:to_i)
  date = Date.new(p, q, r)
  (date - Date.today).to_i
end
```

Judging by its name, the function `days_until_date` returns how many days are left until a certain date. But it seems buggy. Imagine further that today—still in the future—it's Christmas Day 2031. Therefore, you expect to receive the answer that there are six days left until New Year's Eve 2031. You experimentally call the function to see if you receive what you expect.

```
days_until_date('31-12-2031') #=> `initialize': invalid date (Date::Error)
```

But you receive a `Date::Error` instead of the integer six—or any other number, for that matter. You suspect that the regex is wrong, so you do something very wise: You isolate the regex with the Extract Function refactoring pattern into a function and assign the name `date_match` to the function.

```
require 'date'

def date_match(date_string)
```

```
  date_string.match(/(\d+)-(\d+)-(\d+)/)
end

def days_until_date(date_string)
  # This is legacy code, do not modify!
  match = date_match(date_string)
  raise ArgumentError, "Invalid date format" unless match
  p, q, r = match.captures.map(&:to_i)
  date = Date.new(p, q, r)
  (date - Date.today).to_i
end
```

Now you can write tests that explore only the regex.

```
assert_nil date_match('31-12-2031') #=> <#<MatchData> was expected to be nil
```

After trying a test that fails—since `days_until_date` didn't accept `'31-12-2031'`, date_match should not match, as is obviously the case here—you realize that you should perhaps write the date parameter in the ISO 8601 format (YYYY-MM-DD), which means that the year should come first, and the month should come second.

```
days_until_date('2031-12-31') #=> 6
```

(If you—the reader—tries this code on another day than Christmas Day 2031, you'll of course receive another number)

Your assumption holds. When starting with the year, you finally get a satisfying value in return.

This means that the regex should be stricter. Time to move on to the red step of the TDD cycle. Instead of writing tests that pass, you write tests that show how you want `date_match` to work.

```
assert_not_nil date_match('2031-12-31') #=> nil
assert_nil date_match('31-12-2031') #=> <#<MatchData> was expected to be nil
assert_nil date_match('2031-31-12') #=> <#<MatchData> was expected to be nil
assert days_until_date('2031-12-31') #=> nil
```

```
assert_raise(ArgumentError) { days_until_date('31-12-2031') }
  #=> <ArgumentError> exception expected but was AssertionFailedError
```

Three out of five tests failed. Now you are ready to rewrite the regex.

```
def date_match(date_string)
  date_string.match(/(\d{4})-([01][0-9])-([0-3][0-9])/)
end
```

Now all five tests in your test suite pass.

```
assert_not_nil date_match('2031-12-31') #=> nil
assert_nil date_match('31-12-2031') #=> nil
assert_nil date_match('2031-31-12') #=> nil
assert days_until_date('2031-12-31') #=> nil
assert_raise(ArgumentError) { days_until_date('31-12-2031') }
   #=> #<ArgumentError: Invalid date format>
```

This gives you the courage to refactor.

```
def days_until_date(date_string)
  # This is legacy code, do not modify!
  match = date_match(date_string)
  raise ArgumentError, "Invalid date format" unless match
  year, month, day = match.captures.map(&:to_i)
  (Date.new(year, month, day) - Date.today).to_i
end
```

And before you close this TDD session, you, of course, run the test suite one more time to ensure that you didn't mistakenly introduce any new errors.

Two Takeaways

- When working with legacy code, it is often necessary to refactor it to make it testable.
- We can do this by extracting the regex into its own function and writing tests to explore and validate the function's behavior.

Afterword

> *"Some people, when confronted with a problem, think, 'I know, I'll use regular expressions.' Now they have two problems."*

Jamie Zawinski wrote that in August 1997. The second problem, of course, is the potential complexity and difficulty of using regular expressions effectively.

However, having read this book, you are no longer one of those people. You've gained the knowledge and skills to use regular expressions effectively, and the original problem is no longer a threat. In fact, with your new command of regular expressions, you can turn that problem into an opportunity. You are now a regex hero, equipped to tackle complex text-matching challenges with confidence and skill.

But never forget that regex isn't a silver bullet. Some problems can be solved with awesome regex solutions, while others can't be solved with regexes on any level.

And that brings us to the end of this book. Thanks for reading.

Appendix A: Precedence

To create effective and reliable regexes, it's crucial to grasp the order of operations. While most regex flavors adhere to a similar precedence, variations can occur. Context and search strategies may also play a role. Here's the typical order of precedence:

Priority	Operator	Symbol	Position
1	Escape	`\<symbol>`	Prefix
2	Character class	`[]`	Surround
3	Grouping	`()`	Surround
4	Back-reference	`\<index>`	Prefix
5	Quantifier	`?`, `+`, `*`, `{}`	Postfix
6	Quantifier modifier	`?`, `+`	Postfix
7	Concatenation	*"Invisible"*	Infix
8	Anchoring	`^` `$`	Prefix / Postfix
9	Alternation	`|`	Infix

Remember, the table offers helpful guidance, not rigid rules. Parentheses can clarify the order of operations, but excessive use can clutter your regex and complicate grouping indices. Instead, consider adding unit tests with specific examples to your test suite. These tests will verify that your regexes match and don't match as expected, improving code reliability and maintainability.

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