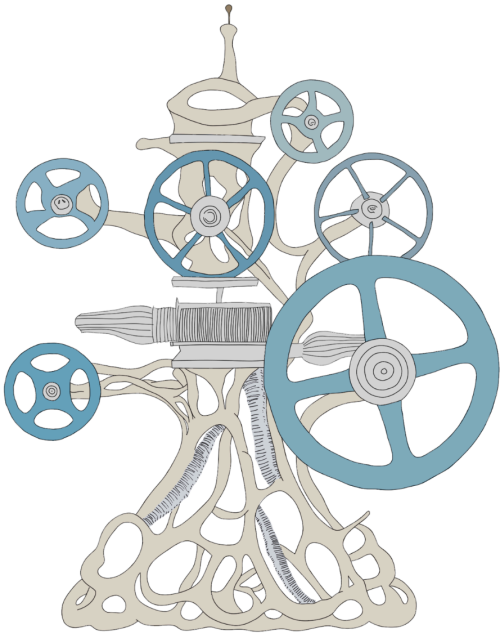
Regular Expressions Machinery

The Illustrated Guide



Staffan Nöteberg

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

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

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

Book Version: V1.0—January, 2025

PART 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.

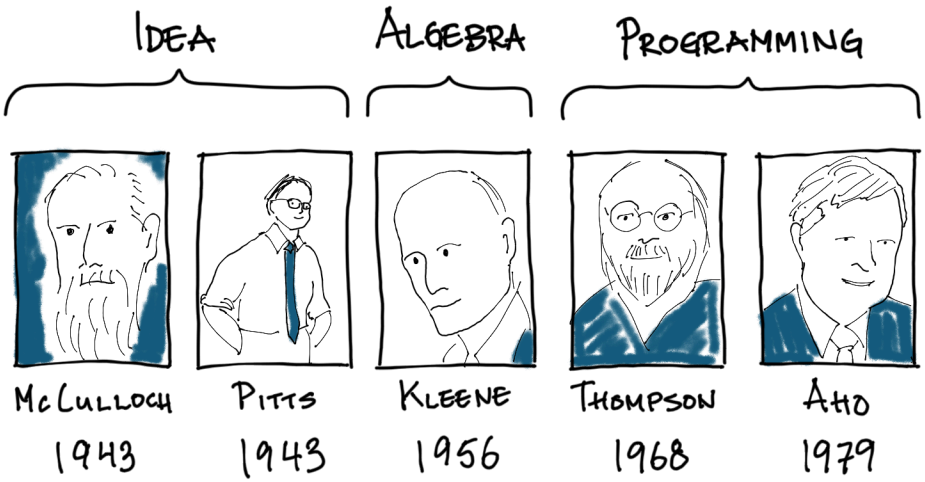
When it comes to details, the ultimate interface between humans and computers has long proven to be text. Visual programming languages are launched recurrently in our industry, but the market never takes off. We still must teach the computer all the details about how it should behave. Too many details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

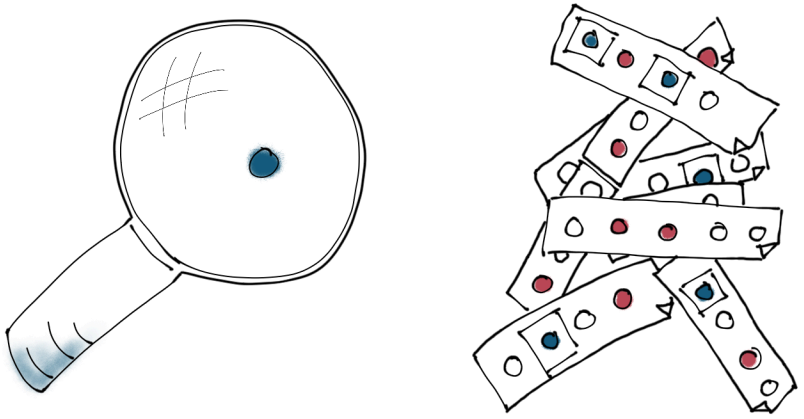
To search for a regular expression in ed, the user wrote `g/<regular expression>/p`. The letter `g` meant global search, and p meant print the result. The command—`g/re/p`—resulted in the stand-alone program grep, released in the fourth edition of Unix in 1973. However, grep didn't have a complete implementation 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 "">
```

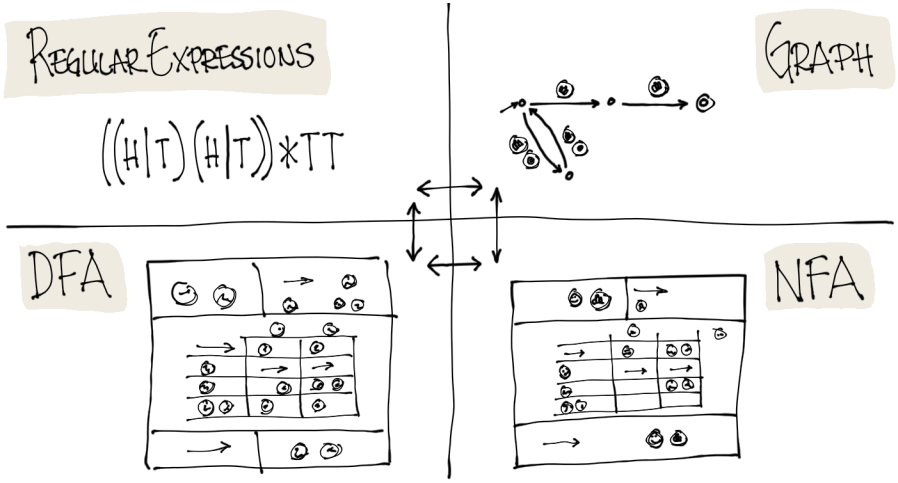
```
# string a not matched by empty regex ε
```

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.

Get a Grip on the Regex Machinery

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

In this illustrated guide, you gain precisely that understanding.

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

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

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

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

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

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

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

PART 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.

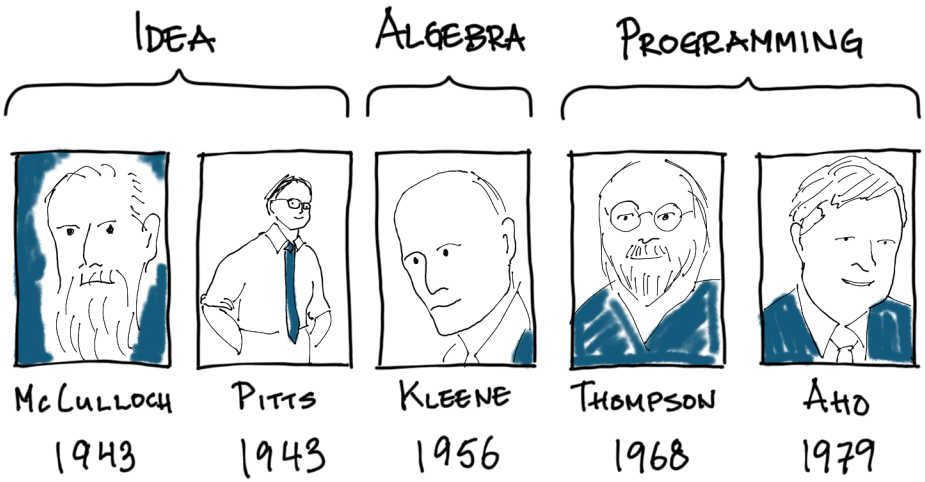
When it comes to details, the ultimate interface between humans and computers has long proven to be text. Visual programming languages are launched recurrently in our industry, but the market never takes off. We still must teach the computer all the details about how it should behave. Too many details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

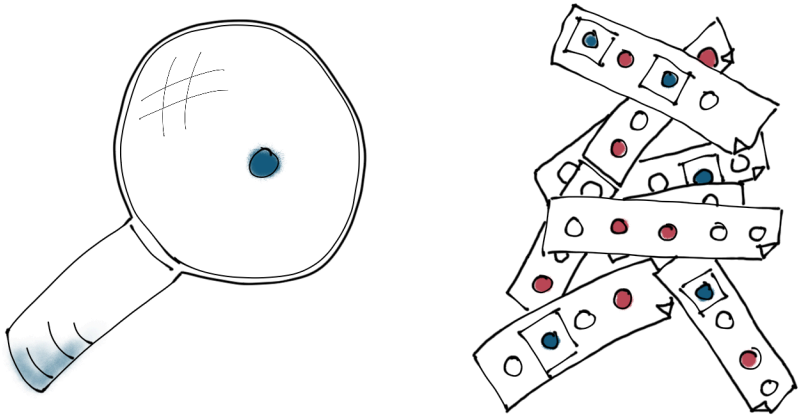
To search for a regular expression in ed, the user wrote `g/<regular expression>/p`. The letter `g` meant global search, and p meant print the result. The command—`g/re/p`—resulted in the stand-alone program grep, released in the fourth edition of Unix in 1973. However, grep didn't have a complete implementation 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 "">
```

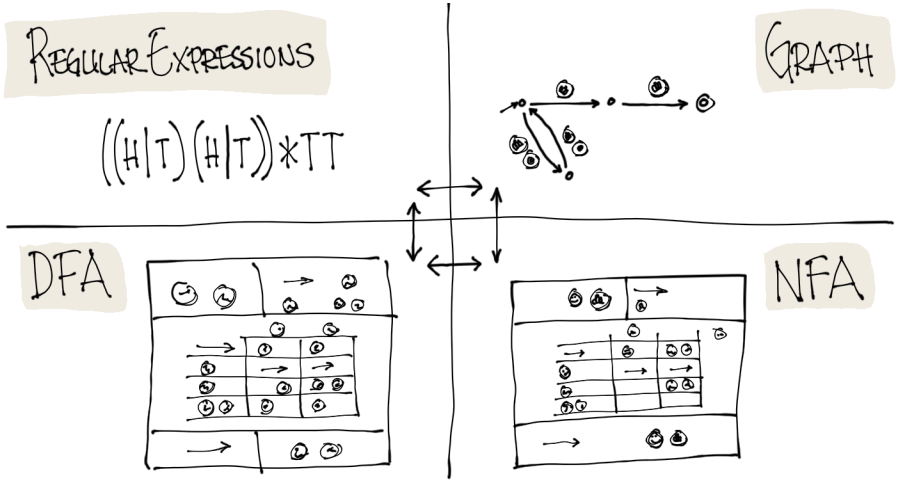
```
# string a not matched by empty regex ε
```

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.