

Regular expression

Friends, Romans, countrymen, lend me your ears;
I come to bury Caesar, not to praise him.
The evil that men do lives after them;
The good is oft interred with their bones;
So let it be with Caesar. The noble Brutus
Hath told you Caesar was ambitious:
If it were so, it was a grievous fault,
And grievously hath Caesar answer'd it.

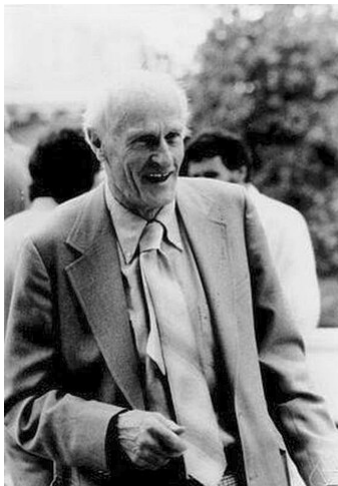
Blue highlights show the match results of the regular expression pattern: `/r[aeiou]+/g` (lower case *r* followed by one or more lower-case vowels).

A **regular expression** (shortened as **regex** or **regexp**),^[1] sometimes referred to as **rational expression**,^{[2][3]} is a sequence of [characters](#) that specifies a [match pattern](#) in [text](#). Usually such patterns are used by [string-searching algorithms](#) for "find" or "find and replace" operations on [strings](#), or for [input validation](#). Regular expression techniques are developed in [theoretical computer science](#) and [formal language](#) theory.

The concept of regular expressions began in the 1950s, when the American mathematician [Stephen Cole Kleene](#) formalized the concept of a [regular language](#). They came into common use with [Unix](#) text-processing utilities. Different [syntaxes](#) for writing regular expressions have existed since the 1980s, one being the [POSIX](#) standard and another, widely used, being the [Perl](#) syntax.

Regular expressions are used in [search engines](#), in search and replace dialogs of [word processors](#) and [text editors](#), in [text processing](#) utilities such as [sed](#) and [AWK](#), and in [lexical analysis](#). Regular expressions are supported in many programming languages. Library implementations are often called an "[engine](#)",^{[4][5]} and [many of these](#) are available for reuse.

History



[Stephen Cole Kleene](#), who introduced the concept

Regular expressions originated in 1951, when mathematician [Stephen Cole Kleene](#) described [regular languages](#) using his mathematical notation called *regular events*.^{[6][7]} These arose in [theoretical computer science](#), in the subfields of [automata theory](#) (models of computation) and the description and classification of [formal languages](#). Other early implementations of [pattern matching](#) include the [SNOBOL](#) language, which did not use regular expressions, but instead its own pattern matching constructs.

Regular expressions entered popular use from 1968 in two uses: pattern matching in a text editor^[8] and lexical analysis in a compiler.^[9] Among the first appearances of regular expressions in program form was when [Ken Thompson](#) built Kleene's notation into the editor [QED](#) as a means to match patterns in [text files](#).^{[8][10][11][12]} For speed, Thompson implemented regular expression matching by [just-in-time compilation](#) (JIT) to [IBM 7094](#) code on the [Compatible Time-Sharing System](#), an important early example of JIT compilation.^[13] He later added this capability to the Unix editor [ed](#), which eventually led to the popular search tool [grep](#)'s use of regular expressions ("grep" is a word derived from the command for regular expression searching in the ed editor: `g/re/p` meaning "Global search for Regular Expression and Print matching lines").^[14] Around the same time when Thompson developed QED, a group of researchers including [Douglas T. Ross](#) implemented a tool based on regular expressions that is used for lexical analysis in [compiler design](#).^[9]

Many variations of these original forms of regular expressions were used in [Unix](#)^[12] programs at [Bell Labs](#) in the 1970s, including [vi](#), [lex](#), [sed](#), [AWK](#), and [expr](#), and in other programs such as [Emacs](#) (which has its own, incompatible syntax and behavior). Regexes were subsequently adopted by a wide range of programs, with these early forms standardized in the [POSIX.2](#) standard in 1992.

In the 1980s, the more complicated regexes arose in [Perl](#), which originally derived from a regex library written by [Henry Spencer](#) (1986), who later wrote an implementation for [Tcl](#) called *Advanced Regular Expressions*.^[15] The Tcl library is a hybrid [NFA/DFA](#) implementation with improved performance characteristics. Software projects that have adopted Spencer's Tcl regular expression implementation include [PostgreSQL](#).^[16] Perl later expanded on Spencer's original library to add many new features.^[17] Part of the effort in the design of [Raku](#) (formerly named Perl 6) is to improve Perl's regex integration, and to increase their scope and capabilities to allow the definition of [parsing expression grammars](#).^[18] The result is a [mini-language](#) called [Raku rules](#), which are used to define Raku grammar as well as provide a tool to programmers in the language. These rules maintain existing features of Perl 5.x regexes, but also allow [BNF](#)-style definition of a [recursive descent parser](#) via sub-rules.

The use of regexes in structured information standards for document and database modeling started in the 1960s and expanded in the 1980s when industry standards like [ISO SGML](#) (precursored by ANSI "GCA 101-1983") consolidated. The kernel of the [structure specification language](#) standards consists of regexes. Its use is evident in the [DTD](#) element group syntax. Prior

to the use of regular expressions, many search languages allowed simple wildcards, for example "*" to match any sequence of characters, and "?" to match a single character. Relics of this can be found today in the [glob](#) syntax for filenames, and in the [SQL LIKE](#) operator.

Starting in 1997, [Philip Hazel](#) developed [PCRE](#) (Perl Compatible Regular Expressions), which attempts to closely mimic Perl's regex functionality and is used by many modern tools including [PHP](#) and [Apache HTTP Server](#).^[19]

Today, regexes are widely supported in programming languages, text processing programs (particularly [lexers](#)), advanced text editors, and some other programs. Regex support is part of the [standard library](#) of many programming languages, including [Java](#) and [Python](#), and is built into the syntax of others, including Perl and [ECMAScript](#). In the late 2010s, several companies started to offer hardware, [FPGA](#),^[20] [GPU](#)^[21] implementations of [PCRE](#) compatible regex engines that are faster compared to [CPU](#) implementations.

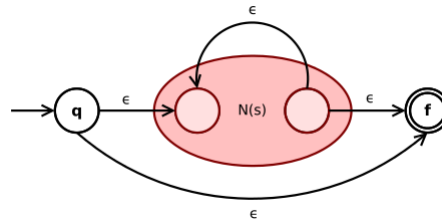
Patterns

The phrase *regular expressions*, or *regexes*, is often used to mean the specific, standard textual syntax for representing patterns for matching text, as distinct from the mathematical notation described below. Each character in a regular expression (that is, each character in the string describing its pattern) is either a [metacharacter](#), having a special meaning, or a regular character that has a literal meaning. For example, in the regex `b.`, 'b' is a literal character that matches just 'b', while '.' is a metacharacter that matches every character except a newline. Therefore, this regex matches, for example, 'b%', or 'bx', or 'b5'. Together, metacharacters and literal characters can be used to identify text of a given pattern or process a number of instances of it. Pattern matches may vary from a precise equality to a very general similarity, as controlled by the metacharacters. For example, `.` is a very general pattern, `[a-z]` (match all lower case letters from 'a' to 'z') is less general and `b` is a precise pattern (matches just 'b'). The metacharacter syntax is designed specifically to represent prescribed targets in a concise and flexible way to direct the automation of text processing of a variety of input data, in a form easy to type using a standard [ASCII keyboard](#).

A very simple case of a regular expression in this syntax is to locate a word spelled two different ways in a [text editor](#), the regular expression `seriali[sz]e` matches both "serialise" and "serialize". [Wildcard characters](#) also achieve this, but are more limited in what they can pattern, as they have fewer metacharacters and a simple language-base.

The usual context of wildcard characters is in [globbing](#) similar names in a list of files, whereas regexes are usually employed in applications that pattern-match text strings in general. For example, the regex `^[\t]+|[\t]+$` matches excess whitespace at the beginning or end of a

line. An advanced regular expression that matches any numeral is `[+-]?(\d+(\.\d*)?|\.\d+)([eE][+-]?\d+)?`.



Translating the Kleene star
(s^* means "zero or more of s ")

A **regex processor** translates a regular expression in the above syntax into an internal representation that can be executed and matched against a [string](#) representing the text being searched in. One possible approach is the [Thompson's construction algorithm](#) to construct a [nondeterministic finite automaton](#) (NFA), which is then [made deterministic](#) and the resulting [deterministic finite automaton](#) (DFA) is run on the target text string to recognize substrings that match the regular expression. The picture shows the NFA scheme $N(s^*)$ obtained from the regular expression s^* , where s denotes a simpler regular expression in turn, which has already been [recursively](#) translated to the NFA $N(s)$.

Basic concepts

A regular expression, often called a *pattern*, specifies a [set](#) of strings required for a particular purpose. A simple way to specify a finite set of strings is to list its [elements](#) or members. However, there are often more concise ways: for example, the set containing the three strings "Handel", "Händel", and "Haendel" can be specified by the pattern `H(ä|ae?)ndel`; we say that this pattern *matches* each of the three strings. However, there can be many ways to write a regular expression for the same set of strings: for example, `(Hän|Han|Haen)del` also specifies the same set of three strings in this example.

Most formalisms provide the following operations to construct regular expressions.

Boolean "or"

A [vertical bar](#) separates alternatives. For example, `gray|grey` can match "gray" or "grey".

Grouping

[Parentheses](#) are used to define the scope and precedence of the [operators](#) (among other uses). For example, `gray|grey` and `gr(a|e)y` are equivalent patterns which both describe the set of "gray" or "grey".

Quantification

A [quantifier](#) after an element (such as a [token](#), character, or group) specifies how many times the preceding element is allowed to repeat. The most common quantifiers are the [question mark](#) `?`, the [asterisk](#) `*` (derived from the [Kleene star](#)), and the [plus sign](#) `+` ([Kleene plus](#)).

?

The question mark indicates *zero or one* occurrences of the preceding element. For example, `colou?r` matches both "color" and "colour".

*

The asterisk indicates *zero or more* occurrences of the preceding element. For example, `ab*c` matches "ac", "abc", "abbc", "abbbc", and so on.

+

The plus sign indicates *one or more* occurrences of the preceding element. For example, `ab+c` matches "abc", "abbc", "abbbc", and so on, but not "ac".

{n}^[22]

The preceding item is matched exactly *n* times.

{min,}^[22]

The preceding item is matched *min* or more times.

{,max}^[22]

The preceding item is matched up to *max* times.

{min,max}^[22]

The preceding item is matched at least *min* times, but not more than *max* times.

Wildcard

The wildcard `.` matches any character. For example,

`a.b` matches any string that contains an "a", and then any character and then "b".

`a.*b` matches any string that contains an "a", and then the character "b" at some later point.

These constructions can be combined to form arbitrarily complex expressions, much like one can construct arithmetical expressions from numbers and the operations +, −, ×, and ÷.

The precise [syntax](#) for regular expressions varies among tools and with context; more detail is given in § [Syntax](#).

Formal language theory

Regular expressions describe [regular languages](#) in [formal language theory](#). They have the same expressive power as [regular grammars](#).

Formal definition

Regular expressions consist of constants, which denote sets of strings, and operator symbols, which denote operations over these sets. The following definition is standard, and found as such in most textbooks on formal language theory.^{[23][24]} Given a finite [alphabet](#) Σ , the following constants are defined as regular expressions:

- (*empty set*) \emptyset denoting the set \emptyset .

- (*empty string*) ϵ denoting the set containing only the "empty" string, which has no characters at all.
- (*literal character*) a in Σ denoting the set containing only the character a .

Given regular expressions R and S , the following operations over them are defined to produce regular expressions:

- (*concatenation*) (RS) denotes the set of strings that can be obtained by concatenating a string accepted by R and a string accepted by S (in that order). For example, let R denote $\{ "ab", "c" \}$ and S denote $\{ "d", "ef" \}$. Then, (RS) denotes $\{ "abd", "abef", "cd", "cef" \}$.
- (*alternation*) $(R|S)$ denotes the *set union* of sets described by R and S . For example, if R describes $\{ "ab", "c" \}$ and S describes $\{ "ab", "d", "ef" \}$, expression $(R|S)$ describes $\{ "ab", "c", "d", "ef" \}$.
- (*Kleene star*) (R^*) denotes the smallest *superset* of the set described by R that contains ϵ and is *closed* under string concatenation. This is the set of all strings that can be made by concatenating any finite number (including zero) of strings from the set described by R . For example, if R denotes $\{ "0", "1" \}$, (R^*) denotes the set of all finite *binary strings* (including the empty string). If R denotes $\{ "ab", "c" \}$, (R^*) denotes $\{ \epsilon, "ab", "c", "abab", "abc", "cab", "cc", "ababab", "abcab", \dots \}$.

To avoid parentheses, it is assumed that the Kleene star has the highest priority followed by concatenation, then alternation. If there is no ambiguity, then parentheses may be omitted. For example, $(ab)c$ can be written as abc , and $a|(b(c^*))$ can be written as $a|bc^*$. Many textbooks use the symbols \cup , $+$, or \vee for alternation instead of the vertical bar.

Examples:

- $a|b^*$ denotes $\{ \epsilon, "a", "b", "bb", "bbb", \dots \}$
- $(a|b)^*$ denotes the set of all strings with no symbols other than "a" and "b", including the empty string: $\{ \epsilon, "a", "b", "aa", "ab", "ba", "bb", "aaa", \dots \}$
- $ab^*(c|\epsilon)$ denotes the set of strings starting with "a", then zero or more "b"s and finally optionally a "c": $\{ "a", "ac", "ab", "abc", "abb", "abbc", \dots \}$
- $(0|(1(01^*0)^*1))^*$ denotes the set of binary numbers that are multiples of 3: $\{ \epsilon, "0", "00", "11", "000", "011", "110", "0000", "0011", "0110", "1001", "1100", "1111", "00000", \dots \}$

Expressive power and compactness

The formal definition of regular expressions is minimal on purpose, and avoids defining $?$ and $+$ —these can be expressed as follows: $a^+ = aa^*$, and $a? = (a|\epsilon)$. Sometimes the *complement* operator is added, to give a *generalized regular expression*; here R^c matches all strings over Σ^* that do not match R . In principle, the complement operator is redundant, because

it does not grant any more expressive power. However, it can make a regular expression much more concise—eliminating a single complement operator can cause a [double exponential](#) blow-up of its length.^{[25][26][27]}

Regular expressions in this sense can express the regular languages, exactly the class of languages accepted by [deterministic finite automata](#). There is, however, a significant difference in compactness. Some classes of regular languages can only be described by deterministic finite automata whose size grows [exponentially](#) in the size of the shortest equivalent regular expressions. The standard example here is the languages L_k consisting of all strings over the alphabet $\{a,b\}$ whose k^{th} -from-last letter equals a . On the one hand, a regular expression describing L_4 is given by $(a \mid b)^* a (a \mid b)(a \mid b)(a \mid b)$.

Generalizing this pattern to L_k gives the expression: $(a \mid b)^* \underbrace{a(a \mid b)(a \mid b) \cdots (a \mid b)}_{k-1 \text{ times}}$.

On the other hand, it is known that every deterministic finite automaton accepting the language L_k must have at least 2^k states. Luckily, there is a simple mapping from regular expressions to the more general [nondeterministic finite automata](#) (NFAs) that does not lead to such a blowup in size; for this reason NFAs are often used as alternative representations of regular languages. NFAs are a simple variation of the type-3 [grammars](#) of the [Chomsky hierarchy](#).^[23]

In the opposite direction, there are many languages easily described by a DFA that are not easily described by a regular expression. For instance, determining the validity of a given [ISBN](#) requires computing the modulus of the integer base 11, and can be easily implemented with an 11-state DFA. However, converting it to a regular expression results in a 2,14 megabytes file.^[28]

Given a regular expression, [Thompson's construction algorithm](#) computes an equivalent nondeterministic finite automaton. A conversion in the opposite direction is achieved by [Kleene's algorithm](#).

Finally, it is worth noting that many real-world "regular expression" engines implement features that cannot be described by the regular expressions in the sense of formal language theory; rather, they implement *regexes*. See [below](#) for more on this.

Deciding equivalence of regular expressions

As seen in many of the examples above, there is more than one way to construct a regular expression to achieve the same results.

It is possible to write an [algorithm](#) that, for two given regular expressions, decides whether the described languages are equal; the algorithm reduces each expression to a [minimal deterministic](#)

[finite state machine](#), and determines whether they are [isomorphic](#) (equivalent).

Algebraic laws for regular expressions can be obtained using a method by Gischer which is best explained along an example: In order to check whether $(X+Y)^*$ and $(X^* Y^*)^*$ denote the same regular language, for all regular expressions X, Y , it is necessary and sufficient to check whether the particular regular expressions $(a+b)^*$ and $(a^* b^*)^*$ denote the same language over the alphabet $\Sigma=\{a,b\}$. More generally, an equation $E=F$ between regular-expression terms with variables holds if, and only if, its instantiation with different variables replaced by different symbol constants holds.^{[29][30]}

Every regular expression can be written solely in terms of the [Kleene star](#) and [set unions](#) over finite words. This is a surprisingly difficult problem. As simple as the regular expressions are, there is no method to systematically rewrite them to some normal form. The lack of axiom in the past led to the [star height problem](#). In 1991, [Dexter Kozen](#) axiomatized regular expressions as a [Kleene algebra](#), using equational and [Horn clause](#) axioms.^[31] Already in 1964, Redko had proved that no finite set of purely equational axioms can characterize the algebra of regular languages.^[32]

Syntax

A regex *pattern* matches a target *string*. The pattern is composed of a sequence of *atoms*. An atom is a single point within the regex pattern which it tries to match to the target string. The simplest atom is a literal, but grouping parts of the pattern to match an atom will require using `()` as metacharacters. Metacharacters help form: *atoms*; *quantifiers* telling how many atoms (and whether it is a [greedy quantifier](#) or not); a logical OR character, which offers a set of alternatives, and a logical NOT character, which negates an atom's existence; and backreferences to refer to previous atoms of a completing pattern of atoms. A match is made, not when all the atoms of the string are matched, but rather when all the pattern atoms in the regex have matched. The idea is to make a small pattern of characters stand for a large number of possible strings, rather than compiling a large list of all the literal possibilities.

Depending on the regex processor there are about fourteen metacharacters, characters that may or may not have their [literal](#) character meaning, depending on context, or whether they are "escaped", i.e. preceded by an [escape sequence](#), in this case, the backslash `\`. Modern and POSIX extended regexes use metacharacters more often than their literal meaning, so to avoid "backslash-osis" or [leaning toothpick syndrome](#), they have a metacharacter escape to a literal mode; starting out, however, they instead have the four bracketing metacharacters `()` and `{ }` be primarily literal, and "escape" this usual meaning to become metacharacters. Common standards implement both. The usual metacharacters are `{ } [] () ^ $. | * + ?` and `\`. The usual characters that become metacharacters when escaped are `dswDSW` and `N`.

Delimiters

When entering a regex in a programming language, they may be represented as a usual string literal, hence usually quoted; this is common in C, Java, and Python for instance, where the regex `re` is entered as `"re"`. However, they are often written with slashes as [delimiters](#), as in `/re/` for the regex `re`. This originates in [ed](#), where `/` is the editor command for searching, and an expression `/re/` can be used to specify a range of lines (matching the pattern), which can be combined with other commands on either side, most famously `g/re/p` as in [grep](#) ("global regex print"), which is included in most [Unix](#)-based operating systems, such as [Linux](#) distributions. A similar convention is used in [sed](#), where search and replace is given by `s/re/replacement/` and patterns can be joined with a comma to specify a range of lines as in `/re1/,/re2/`. This notation is particularly well known due to its use in [Perl](#), where it forms part of the syntax distinct from normal string literals. In some cases, such as `sed` and `Perl`, alternative delimiters can be used to avoid collision with contents, and to avoid having to escape occurrences of the delimiter character in the contents. For example, in `sed` the command `s/,/X,` will replace a `/` with an `X`, using commas as delimiters.

IEEE POSIX Standard

The [IEEE POSIX](#) standard has three sets of compliance: **BRE** (Basic Regular Expressions),^[33] **ERE** (Extended Regular Expressions), and **SRE** (Simple Regular Expressions). SRE is [deprecated](#),^[34] in favor of BRE, as both provide [backward compatibility](#). The subsection below covering the *character classes* applies to both BRE and ERE.

BRE and ERE work together. ERE adds `?`, `+`, and `|`, and it removes the need to escape the metacharacters `()` and `{ }`, which are *required* in BRE. Furthermore, as long as the POSIX standard syntax for regexes is adhered to, there can be, and often is, additional syntax to serve specific (yet POSIX compliant) applications. Although POSIX.2 leaves some implementation specifics undefined, BRE and ERE provide a "standard" which has since been adopted as the default syntax of many tools, where the choice of BRE or ERE modes is usually a supported option. For example, [GNU grep](#) has the following options: `"grep -E"` for ERE, and `"grep -G"` for BRE (the default), and `"grep -P"` for [Perl](#) regexes.

Perl regexes have become a de facto standard, having a rich and powerful set of atomic expressions. Perl has no "basic" or "extended" levels. As in POSIX EREs, `()` and `{ }` are treated as metacharacters unless escaped; other metacharacters are known to be literal or symbolic based on context alone. Additional functionality includes [lazy matching](#), [backreferences](#), named capture groups, and [recursive](#) patterns.

POSIX basic and extended

In the [POSIX](#) standard, Basic Regular Syntax (**BRE**) requires that the [metacharacters](#) `()` and `{ }` be designated `\(\)` and `\{ \}`, whereas Extended Regular Syntax (**ERE**) does not.

Metacharacter	Description
<code>^</code>	Matches the starting position within the string. In line-based tools, it matches the starting position of any line.
<code>.</code>	Matches any single character (many applications exclude newlines , and exactly which characters are considered newlines is flavor-, character-encoding-, and platform-specific, but it is safe to assume that the line feed character is included). Within POSIX bracket expressions, the dot character matches a literal dot. For example, <code>a.c</code> matches "abc", etc., but <code>[a.c]</code> matches only "a", ".", or "c".
<code>[]</code>	<p>A bracket expression. Matches a single character that is contained within the brackets. For example, <code>[abc]</code> matches "a", "b", or "c". <code>[a-z]</code> specifies a range which matches any lowercase letter from "a" to "z". These forms can be mixed: <code>[abcx-z]</code> matches "a", "b", "c", "x", "y", or "z", as does <code>[a-cx-z]</code>.</p> <p>The <code>-</code> character is treated as a literal character if it is the last or the first (after the <code>^</code>, if present) character within the brackets: <code>[abc-]</code>, <code>[-abc]</code>. Backslash escapes are not allowed. The <code>]</code> character can be included in a bracket expression if it is the first (after the <code>^</code>) character: <code>[]abc]</code>.</p>
<code>[^]</code>	Matches a single character that is not contained within the brackets. For example, <code>[^abc]</code> matches any character other than "a", "b", or "c". <code>[^a-z]</code> matches any single character that is not a lowercase letter from "a" to "z". Likewise, literal characters and ranges can be mixed.
<code>\$</code>	Matches the ending position of the string or the position just before a string-ending newline. In line-based tools, it matches the ending position of any line.
<code>()</code>	Defines a marked subexpression. The string matched within the parentheses can be recalled later (see the next entry, <code>\n</code>). A marked subexpression is also called a block or capturing group. <i>BRE mode requires <code>\(\)</code>.</i>
<code>\n</code>	Matches what the <i>n</i> th marked subexpression matched, where <i>n</i> is a digit from 1 to 9. This construct is defined in the POSIX standard. ^[35] Some tools allow referencing more than nine capturing groups. Also known as a back-reference, this feature is supported in BRE mode.
<code>*</code>	Matches the preceding element zero or more times. For example, <code>ab*c</code> matches "ac", "abc", "abbbc", etc. <code>[xyz]*</code> matches "", "x", "y", "z", "zx", "zyx", "xyzy", and so on. <code>(ab)*</code> matches "", "ab", "abab", "ababab", and so on.
<code>{m,n}</code>	Matches the preceding element at least <i>m</i> and not more than <i>n</i> times. For example, <code>a{3,5}</code> matches only "aaa", "aaaa", and "aaaaa". This is not found in a few older instances of regexes. BRE mode requires <code>\{m,n\}</code> .

Examples:

- `.at` matches any three-character string ending with "at", including "hat", "cat", "bat", "4at", "#at" and " at" (starting with a space).

- `[hc]at` matches "hat" and "cat".
- `[^b]at` matches all strings matched by `.at` except "bat".
- `[^hc]at` matches all strings matched by `.at` other than "hat" and "cat".
- `^[hc]at` matches "hat" and "cat", but only at the beginning of the string or line.
- `[hc]at$` matches "hat" and "cat", but only at the end of the string or line.
- `\[.\\]` matches any single character surrounded by "[" and "]" since the brackets are escaped, for example: "[a]", "[b]", "[7]", "[@]", "[]", and "[]" (bracket space bracket).
- `s.*` matches s followed by zero or more characters, for example: "s", "saw", "seed", "s3w96.7", and "s6#h%(>>m n mQ".

According to Ross Cox, the POSIX specification requires ambiguous subexpressions to be handled in a way different from Perl's. The committee replaced Perl's rules with one that is simple to explain, but the new "simple" rules are actually more complex to implement: they were incompatible with pre-existing tooling and made it essentially impossible to define a "lazy match" (see below) extension. As a result, very few programs actually implement the POSIX subexpression rules (even when they implement other parts of the POSIX syntax).^[36]

Metacharacters in POSIX extended

The meaning of metacharacters [escaped](#) with a backslash is reversed for some characters in the POSIX Extended Regular Expression (**ERE**) syntax. With this syntax, a backslash causes the metacharacter to be treated as a literal character. So, for example, `\(\)` is now `()` and `\{ \}` is now `{ }`. Additionally, support is removed for `\n` backreferences and the following metacharacters are added:

Metacharacter	Description
<code>?</code>	Matches the preceding element zero or one time. For example, <code>ab?c</code> matches only "ac" or "abc".
<code>+</code>	Matches the preceding element one or more times. For example, <code>ab+c</code> matches "abc", "abbc", "abbbc", and so on, but not "ac".
<code> </code>	The choice (also known as alternation or set union) operator matches either the expression before or the expression after the operator. For example, <code>abc def</code> matches "abc" or "def".

Examples:

- `[hc]?at` matches "at", "hat", and "cat".
- `[hc]*at` matches "at", "hat", "cat", "hhat", "chat", "hcat", "cchchat", and so on.
- `[hc]+at` matches "hat", "cat", "hhat", "chat", "hcat", "cchchat", and so on, but not "at".
- `cat|dog` matches "cat" or "dog".

POSIX Extended Regular Expressions can often be used with modern Unix utilities by including the [command line](#) flag `-E`.

Character classes

The character class is the most basic regex concept after a literal match. It makes one small sequence of characters match a larger set of characters. For example, `[A-Z]` could stand for any uppercase letter in the English alphabet, and `\d` could mean any digit. Character classes apply to both POSIX levels.

When specifying a range of characters, such as `[a-z]` (i.e. lowercase `a` to uppercase `Z`), the computer's locale settings determine the contents by the numeric ordering of the character encoding. They could store digits in that sequence, or the ordering could be `abc...zABC...Z`, or `aAbBcC...zZ`. So the POSIX standard defines a character class, which will be known by the regex processor installed. Those definitions are in the following table:

Description	POSIX	Perl/Tcl	Vim	Java	ASCII
ASCII characters				<code>\p{ASCII}</code>	<code>[\x00-\x7F]</code>
Alphanumeric characters	<code>[:alnum:]</code>			<code>\p{Alnum}</code>	<code>[A-Za-z0-9]</code>
Alphanumeric characters plus "_"		<code>\w</code>	<code>\w</code>	<code>\w</code>	<code>[A-Za-z0-9_]</code>
Non-word characters		<code>\W</code>	<code>\W</code>	<code>\W</code>	<code>[^A-Za-z0-9_]</code>
Alphabetic characters	<code>[:alpha:]</code>		<code>\a</code>	<code>\p{Alpha}</code>	<code>[A-Za-z]</code>
Space and tab	<code>[:blank:]</code>		<code>\s</code>	<code>\p{Blank}</code>	<code>[\t]</code>
Word boundaries		<code>\b</code>	<code>\<</code> <code>\></code>	<code>\b</code>	<code>(?<=\W)(?=\W) (?<=\W)(?=\W)</code>
Non-word boundaries				<code>\B</code>	<code>(?<=\W)(?=\W) (?<=\W)(?=\W)</code>
Control characters	<code>[:cntrl:]</code>			<code>\p{Cntrl}</code>	<code>[\x00-\x1F\x7F]</code>
Digits	<code>[:digit:]</code>	<code>\d</code>	<code>\d</code>	<code>\p{Digit}</code> or <code>\d</code>	<code>[0-9]</code>
Non-digits		<code>\D</code>	<code>\D</code>	<code>\D</code>	<code>[^0-9]</code>
Visible characters	<code>[:graph:]</code>			<code>\p{Graph}</code>	<code>[\x21-\x7E]</code>
Lowercase letters	<code>[:lower:]</code>		<code>\l</code>	<code>\p{Lower}</code>	<code>[a-z]</code>
Visible characters and the space character	<code>[:print:]</code>		<code>\p</code>	<code>\p{Print}</code>	<code>[\x20-\x7E]</code>
Punctuation characters	<code>[:punct:]</code>			<code>\p{Punct}</code>	<code>[!\"#\$%&'()*+,-./:;<=>?@^_`{ }~]</code>
Whitespace characters	<code>[:space:]</code>	<code>\s</code>	<code>_s</code>	<code>\p{Space}</code> or <code>\s</code>	<code>[\t\r\n\v\f]</code>
Non-whitespace characters		<code>\S</code>	<code>\S</code>	<code>\S</code>	<code>[^ \t\r\n\v\f]</code>
Uppercase letters	<code>[:upper:]</code>		<code>\u</code>	<code>\p{Upper}</code>	<code>[A-Z]</code>
Hexadecimal digits	<code>[:xdigit:]</code>		<code>\x</code>	<code>\p{XDigit}</code>	<code>[A-Fa-f0-9]</code>

POSIX character classes can only be used within bracket expressions. For example,

`[:upper:]ab` matches the uppercase letters and lowercase "a" and "b".

An additional non-POSIX class understood by some tools is `[:word:]`, which is usually defined as `[:alnum:]` plus underscore. This reflects the fact that in many programming languages these are the characters that may be used in identifiers. The editor [Vim](#) further distinguishes *word* and *word-head* classes (using the notation `\w` and `\h`) since in many programming languages the characters that can begin an identifier are not the same as those that can occur in other positions: numbers are generally excluded, so an identifier would look like `\h\w*` or `[:alpha:]_ [:alnum:]_*` in POSIX notation.

Note that what the POSIX regex standards call *character classes* are commonly referred to as *POSIX character classes* in other regex flavors which support them. With most other regex flavors, the term *character class* is used to describe what POSIX calls *bracket expressions*.

Perl and PCRE

Because of its expressive power and (relative) ease of reading, many other utilities and programming languages have adopted syntax similar to Perl's—for example, [Java](#), [JavaScript](#), [Julia](#), [Python](#), [Ruby](#), [Qt](#), Microsoft's [.NET Framework](#), and [XML Schema](#). Some languages and tools such as [Boost](#) and [PHP](#) support multiple regex flavors. Perl-derivative regex implementations are not identical and usually implement a subset of features found in Perl 5.0, released in 1994. Perl sometimes does incorporate features initially found in other languages. For example, Perl 5.10 implements syntactic extensions originally developed in PCRE and Python.^[37]

Lazy matching

In Python and some other implementations (e.g. Java), the three common quantifiers (`*`, `+` and `?`) are [greedy](#) by default because they match as many characters as possible.^[38] The regex `".+"` (including the double-quotes) applied to the string

```
"Ganymede," he continued, "is the largest moon in the Solar System."
```

matches the entire line (because the entire line begins and ends with a double-quote) instead of matching only the first part, `"Ganymede, "`. The aforementioned quantifiers may, however, be made *lazy* or *minimal* or *reluctant*, matching as few characters as possible, by appending a question mark: `".+?"` matches only `"Ganymede, "`.^[38]

Possessive matching

In Java and Python 3.11+,^[39] quantifiers may be made *possessive* by appending a plus sign, which disables backing off (in a backtracking engine), even if doing so would allow the overall match to succeed.^[40] While the regex `".*"` applied to the string

```
"Ganymede," he continued, "is the largest moon in the Solar System."
```

matches the entire line, the regex `".*+"` does *not match at all*, because `".*+"` consumes the entire input, including the final `"`. Thus, possessive quantifiers are most useful with negated

character classes, e.g. `"[^"]*+"`, which matches `"Ganymede, "` when applied to the same string.

Another common extension serving the same function is atomic grouping, which disables backtracking for a parenthesized group. The typical syntax is `(?>group)`. For example, while `^(wi|w)i$` matches both `wi` and `wii`, `^(?>wi|w)i$` only matches `wii` because the engine is forbidden from backtracking and so cannot try setting the group to `"w"` after matching `"wi"`.^[41]

Possessive quantifiers are easier to implement than greedy and lazy quantifiers, and are typically more efficient at runtime.^[40]

IETF I-Regexp

IETF RFC 9485 describes "I-Regexp: An Interoperable Regular Expression Format". It specifies a limited subset of regular-expression idioms designed to be interoperable, i.e. produce the same effect, in a large number of regular-expression libraries. I-Regexp is also limited to matching, i.e. providing a true or false match between a regular expression and a given piece of text. Thus, it lacks advanced features such as capture groups, lookahead, and backreferences.^[42]

Patterns for non-regular languages

Many features found in virtually all modern regular expression libraries provide an expressive power that exceeds the [regular languages](#). For example, many implementations allow grouping subexpressions with parentheses and recalling the value they match in the same expression (*backreferences*). This means that, among other things, a pattern can match strings of repeated words like `"papa"` or `"WikiWiki"`, called *squares* in formal language theory. The pattern for these strings is `(.+)\\1`.

The language of squares is not regular, nor is it [context-free](#), due to the [pumping lemma](#). However, [pattern matching](#) with an unbounded number of backreferences, as supported by numerous modern tools, is still [context sensitive](#).^[43] The general problem of matching any number of backreferences is [NP-complete](#), and the execution time for known algorithms grows exponentially by the number of backreference groups used.^[44]

However, many tools, libraries, and engines that provide such constructions still use the term *regular expression* for their patterns. This has led to a nomenclature where the term regular expression has different meanings in [formal language theory](#) and pattern matching. For this reason, some people have taken to using the term *regex*, *regexp*, or simply *pattern* to describe the latter. [Larry Wall](#), author of the Perl programming language, writes in an essay about the design of Raku:

"Regular expressions" [...] are only marginally related to real regular expressions. Nevertheless, the term has grown with the capabilities of our pattern matching engines, so I'm not going to try to fight linguistic necessity here. I will, however, generally call them "regexes" (or "regexen", when I'm in an Anglo-Saxon mood).^[18]

Assertions

Assertion	Lookbehind	Lookahead
Positive	(?<= pattern)	(?= pattern)
Negative	(?<! pattern)	(?! pattern)
Look-behind and look-ahead assertions in Perl regular expressions		

Other features not found in describing regular languages include assertions. These include the ubiquitous `^` and `$`, used since at least 1970,^[45] as well as some more sophisticated extensions like lookaround that appeared in 1994.^[46] Lookarounds define the surrounding of a match and do not spill into the match itself, a feature only relevant for the use case of string searching. Some of them can be simulated in a regular language by treating the surroundings as a part of the language as well.^[47]

The look-ahead assertions `(?=...)` and `(?!...)` have been attested since at least 1994, starting with Perl 5.^[46] The look-behind assertions `(?<=...)` and `(?<!...)` are attested since 1997 in a commit by Ilya Zakharevich to Perl 5.005.^[48]

Implementations and running times

There are at least three different [algorithms](#) that decide whether and how a given regex matches a string.

The oldest and fastest relies on a result in formal language theory that allows every [nondeterministic finite automaton](#) (NFA) to be transformed into a [deterministic finite automaton](#) (DFA). The DFA can be constructed explicitly and then run on the resulting input string one symbol at a time. Constructing the DFA for a regular expression of size m has the time and memory cost of $O(2^m)$, but it can be run on a string of size n in time $O(n)$. Note that the size of the expression is the size after abbreviations, such as numeric quantifiers, have been expanded.

An alternative approach is to simulate the NFA directly, essentially building each DFA state on demand and then discarding it at the next step. This keeps the DFA implicit and avoids the exponential construction cost, but running cost rises to $O(mn)$. The explicit approach is called the

DFA algorithm and the implicit approach the NFA algorithm. Adding caching to the NFA algorithm is often called the "lazy DFA" algorithm, or just the DFA algorithm without making a distinction. These algorithms are fast, but using them for recalling grouped subexpressions, lazy quantification, and similar features is tricky.^{[49][50]} Modern implementations include the `re1-re2-sregex` family based on Cox's code.

The third algorithm is to match the pattern against the input string by [backtracking](#). This algorithm is commonly called NFA, but this terminology can be confusing. Its running time can be exponential, which simple implementations exhibit when matching against expressions like `(a|aa)*b` that contain both alternation and unbounded quantification and force the algorithm to consider an exponentially increasing number of sub-cases. This behavior can cause a security problem called [Regular expression Denial of Service](#) (ReDoS).

Although backtracking implementations only give an exponential guarantee in the worst case, they provide much greater flexibility and expressive power. For example, any implementation which allows the use of backreferences, or implements the various extensions introduced by Perl, must include some kind of backtracking. Some implementations try to provide the best of both algorithms by first running a fast DFA algorithm, and revert to a potentially slower backtracking algorithm only when a backreference is encountered during the match. GNU `grep` (and the underlying `gnulib` DFA) uses such a strategy.^[51]

Sublinear runtime algorithms have been achieved using [Boyer-Moore \(BM\) based algorithms](#) and related DFA optimization techniques such as the reverse scan.^[52] GNU `grep`, which supports a wide variety of POSIX syntaxes and extensions, uses BM for a first-pass prefiltering, and then uses an implicit DFA. Wu `agrep`, which implements approximate matching, combines the prefiltering into the DFA in BDM (backward DAWG matching). NR-`grep`'s BNBM extends the BDM technique with Shift-Or bit-level parallelism.^[53]

A few theoretical alternatives to backtracking for backreferences exist, and their "exponents" are tamer in that they are only related to the number of backreferences, a fixed property of some regexp languages such as POSIX. One naive method that duplicates a non-backtracking NFA for each backreference note has a complexity of $O(n^{2k+2})$ time and $O(n^{2k+1})$ space for a haystack of length n and k backreferences in the `RegExp`.^[54] A very recent theoretical work based on memory automata gives a tighter bound based on "active" variable nodes used, and a polynomial possibility for some backreferenced regexps.^[55]

Unicode

In theoretical terms, any token set can be matched by regular expressions as long as it is pre-defined. In terms of historical implementations, regexes were originally written to use [ASCII](#) characters as their token set though regex libraries have supported numerous other [character](#)

[sets](#). Many modern regex engines offer at least some support for [Unicode](#). In most respects it makes no difference what the character set is, but some issues do arise when extending regexes to support Unicode.

- **Supported encoding.** Some regex libraries expect to work on some particular encoding instead of on abstract Unicode characters. Many of these require the [UTF-8](#) encoding, while others might expect [UTF-16](#), or [UTF-32](#). In contrast, Perl and Java are agnostic on encodings, instead operating on decoded characters internally.
- **Supported Unicode range.** Many regex engines support only the [Basic Multilingual Plane](#), that is, the characters which can be encoded with only 16 bits. Currently (as of 2016) only a few regex engines (e.g., Perl's and Java's) can handle the full 21-bit Unicode range.
- **Extending ASCII-oriented constructs to Unicode.** For example, in ASCII-based implementations, character ranges of the form `[x-y]` are valid wherever *x* and *y* have [code points](#) in the range `[0x00,0x7F]` and `codepoint(x) ≤ codepoint(y)`. The natural extension of such character ranges to Unicode would simply change the requirement that the endpoints lie in `[0x00,0x7F]` to the requirement that they lie in `[0x0000,0x10FFFF]`. However, in practice this is often not the case. Some implementations, such as that of [gawk](#), do not allow character ranges to cross Unicode blocks. A range like `[0x61,0x7F]` is valid since both endpoints fall within the Basic Latin block, as is `[0x0530,0x0560]` since both endpoints fall within the Armenian block, but a range like `[0x0061,0x0532]` is invalid since it includes multiple Unicode blocks. Other engines, such as that of the [Vim](#) editor, allow block-crossing but the character values must not be more than 256 apart.^[56]
- **Case insensitivity.** Some case-insensitivity flags affect only the ASCII characters. Other flags affect all characters. Some engines have two different flags, one for ASCII, the other for Unicode. Exactly which characters belong to the POSIX classes also varies.
- **Cousins of case insensitivity.** As ASCII has case distinction, case insensitivity became a logical feature in text searching. Unicode introduced alphabetic scripts without case like [Devanagari](#). For these, [case sensitivity](#) is not applicable. For scripts like Chinese, another distinction seems logical: between traditional and simplified. In Arabic scripts, insensitivity to [initial](#), [medial](#), [final](#), and [isolated position](#) may be desired. In Japanese, insensitivity between [hiragana](#) and [katakana](#) is sometimes useful.
- **Normalization.** Unicode has [combining characters](#). Like old typewriters, plain base characters (white spaces, punctuation characters, symbols, digits, or letters) can be followed by one or more non-spacing symbols (usually diacritics, like accent marks modifying letters) to form a single printable character; but Unicode also provides a limited set of precomposed characters, i.e. characters that already include one or more combining characters. A sequence of a base character + combining characters should be matched with the identical single precomposed character (only some of these combining sequences can be precomposed into a single

Unicode character, but infinitely many other combining sequences are possible in Unicode, and needed for various languages, using one or more combining characters after an initial base character; these combining sequences *may* include a base character or combining characters partially precomposed, but not necessarily in canonical order and not necessarily using the canonical precompositions). The process of standardizing sequences of a base character + combining characters by decomposing these *canonically equivalent* sequences, before reordering them into canonical order (and optionally recomposing *some* combining characters into the leading base character) is called normalization.

- **New control codes.** Unicode introduced amongst others, [byte order marks](#) and text direction markers. These codes might have to be dealt with in a special way.
- **Introduction of character classes for Unicode blocks, scripts, and numerous other character properties.** Block properties are much less useful than script properties, because a block can have code points from several different scripts, and a script can have code points from several different blocks.^[57] In [Perl](#) and the [java.util.regex](https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/regex/package-summary.html) (<https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/regex/package-summary.html>) library, properties of the form `\p{InX}` or `\p{Block=X}` match characters in block `X` and `\P{InX}` or `\P{Block=X}` matches code points not in that block. Similarly, `\p{Armenian}`, `\p{IsArmenian}`, or `\p{Script=Armenian}` matches any character in the Armenian script. In general, `\p{X}` matches any character with either the binary property `X` or the general category `X`. For example, `\p{Lu}`, `\p{Uppercase_Letter}`, or `\p{GC=Lu}` matches any uppercase letter. Binary properties that are *not* general categories include `\p{White_Space}`, `\p{Alphabetic}`, `\p{Math}`, and `\p{Dash}`. Examples of non-binary properties are `\p{Bidi_Class=Right_to_Left}`, `\p{Word_Break=A_Letter}`, and `\p{Numeric_Value=10}`.

Language support

Most [general-purpose programming languages](#) support regex capabilities, either natively or via [libraries](#). Comprehensive support is included in:

- [C](#)^[58]
- [C++](#)^[59]
- [C#](#)^[60]
- [Java](#)^[61]
- [JavaScript](#)^[62]
- [OCaml](#)^[63]
- [Perl](#)^[64]
- [PHP](#)^[65]
- [Python](#)^[66]
- [Rust](#)^[67]

Uses



A blacklist on Wikipedia which uses regular expressions to identify bad titles

Regexes are useful in a wide variety of text processing tasks, and more generally [string processing](#), where the data need not be textual. Common applications include [data validation](#), [data scraping](#) (especially [web scraping](#)), [data wrangling](#), simple [parsing](#), the production of [syntax highlighting](#) systems, and many other tasks.

While regexes would be useful on Internet [search engines](#), processing them across the entire database could consume excessive computer resources depending on the complexity and design of the regex. Although in many cases system administrators can run regex-based queries internally, most search engines do not offer regex support to the public. Notable exceptions include [Google Code Search](#) and [Exalead](#). However, Google Code Search was shut down in January 2012.^[68]

Examples

The specific syntax rules vary depending on the specific implementation, [programming language](#), or [library](#) in use. Additionally, the functionality of regex implementations can vary between [versions](#).

Because regexes can be difficult to both explain and understand without examples, interactive websites for testing regexes are a useful resource for learning regexes by experimentation. This section provides a basic description of some of the properties of regexes by way of illustration.

The following conventions are used in the examples.^[69]

```
metacharacter(s) ;; the metacharacters column specifies the regex syntax
being demonstrated
=~ m//           ;; indicates a regex match operation in Perl
=~ s///          ;; indicates a regex substitution operation in Perl
```


Also worth noting is that these regexes are all Perl-like syntax. Standard [POSIX](#) regular expressions are different.

Unless otherwise indicated, the following examples conform to the [Perl](#) programming language, release 5.8.8, January 31, 2006. This means that other implementations may lack support for some parts of the syntax shown here (e.g. basic vs. extended regex, `\(\)` vs. `()`, or lack of `\d` instead of [POSIX](#) `[:digit:]`).

The syntax and conventions used in these examples coincide with that of other programming environments as well.^[70]

Meta-character(s)	Description	Example ^[71]
.	Normally matches any character except a newline. Within square brackets the dot is literal.	<pre>\$string1 = "Hello World\n"; if (\$string1 =~ m/...../) { print "\$string1 has length >= 5.\n"; }</pre> <p>Output:</p> <pre>Hello World has length >= 5.</pre>
()	Groups a series of pattern elements to a single element. When you match a pattern within parentheses, you can use any of <code>\$1</code> , <code>\$2</code> , ... later to refer to the previously matched pattern. Some implementations may use a backslash notation instead, like <code>\1</code> , <code>\2</code> .	<pre>\$string1 = "Hello World\n"; if (\$string1 =~ m/(H..)(o..)/) { print "We matched '\$1' and '\$2'.\n"; }</pre> <p>Output:</p> <pre>We matched 'Hel' and 'o W'.</pre>
+	Matches the preceding pattern element one or more times.	<pre>\$string1 = "Hello World\n"; if (\$string1 =~ m/l+/) { print "There are one or more consecutive letter 'l''s in \$string1.\n"; }</pre> <p>Output:</p> <pre>There are one or more consecutive letter "l"'s in Hello World.</pre>
?	Matches the preceding pattern element zero or one time.	<pre>\$string1 = "Hello World\n"; if (\$string1 =~ m/H.?e/) { print "There is an 'H' and a 'e' separated by "; print "0-1 characters (e.g., He Hue Hee).\n"; }</pre> <p>Output:</p> <pre>There is an 'H' and a 'e' separated by 0-1 characters (e.g., He Hue Hee).</pre>
*	Modifies the <code>*</code> , <code>+</code> , <code>?</code> or <code>{M,N}</code> 'd regex that comes before to match as few times as possible.	<pre>\$string1 = "Hello World\n"; if (\$string1 =~ m/(l.+?o)/)</pre>

		<pre>{ print "The non-greedy match with 'l' followed by one or "; print "more characters is 'llo' rather than 'llo Wo'.\\n"; }</pre> <p>Output:</p> <p>The non-greedy match with 'l' followed by one or more characters is 'llo' rather than 'llo Wo'.</p>
*	Matches the preceding pattern element zero or more times.	<pre>\$string1 = "Hello World\\n"; if (\$string1 =~ m/el*/o/) { print "There is an 'e' followed by zero to many "; print "'l' followed by 'o' (e.g., eo, elo, ello, ello).\\n"; }</pre> <p>Output:</p> <p>There is an 'e' followed by zero to many 'l' followed by 'o' (e.g., eo, elo, ello, ello).</p>
{M,N}	<p>Denotes the minimum M and the maximum N match count. N can be omitted and M can be 0: {M} matches "exactly" M times; {M,} matches "at least" M times; {0,N} matches "at most" N times.</p> <p>x* y+ z? is thus equivalent to x{0,} y{1,} z{0,1}.</p>	<pre>\$string1 = "Hello World\\n"; if (\$string1 =~ m/l{1,2}/) { print "There exists a substring with at least 1 "; print "and at most 2 l's in \$string1\\n"; }</pre> <p>Output:</p> <p>There exists a substring with at least 1 and at most 2 l's in Hello World</p>
[...]	Denotes a set of possible character matches.	<pre>\$string1 = "Hello World\\n"; if (\$string1 =~ m/[aeiou]+)/ { print "\$string1 contains one or more vowels.\\n"; }</pre> <p>Output:</p>

		<div>Hello World contains one or more vowels.</div>
<div> </div>	<div>Separates alternate possibilities.</div>	<div><pre>\$string1 = "Hello World\n"; if (\$string1 =~ m/(Hello Hi Pogo)/) { print "\$string1 contains at least one of Hello, Hi, or Pogo."; }</pre></div> <div>Output:</div> <div>Hello World contains at least one of Hello, Hi, or Pogo.</div>
<div>\b</div>	<div>Matches a zero-width boundary between a word-class character (see next) and either a non-word class character or an edge; same as <div>(^\w \w\$ \W\w \w\W) .</div></div>	<div><pre>\$string1 = "Hello World\n"; if (\$string1 =~ m/llo\b/) { print "There is a word that ends with 'llo'.\n"; }</pre></div> <div>Output:</div> <div>There is a word that ends with 'llo'.</div>
<div>\w</div>	<div>Matches an alphanumeric character, including "_"; same as <div>[A-Za-z0-9_]</div> in ASCII, and <div>[\p{Alphabetic}\p{GC=Mark}\p{GC=Decimal_Number}\p{GC=Connector_Punctuation}]</div> in Unicode,^[57] where the <div>Alphabetic</div> property contains more than Latin letters, and the <div>Decimal_Number</div> property contains more than Arab digits.</div>	<div><pre>\$string1 = "Hello World\n"; if (\$string1 =~ m/\w/) { print "There is at least one alphanumeric "; print "character in \$string1 (A-Z, a-z, 0-9, _).\n"; }</pre></div> <div>Output:</div> <div>There is at least one alphanumeric character in Hello World (A-Z, a-z, 0-9, _).</div>
<div>\W</div>	<div>Matches a <i>non</i>-alphanumeric character, excluding "_"; same as <div>[^A-Za-z0-9_]</div> in ASCII, and <div>[^\p{Alphabetic}\p{GC=Mark}\p{GC=Decimal_Number}\p{GC=Connector_Punctuation}]</div> in Unicode.</div>	<div><pre>\$string1 = "Hello World\n"; if (\$string1 =~ m/\W/) { print "The space between Hello and "; print "World is not alphanumeric.\n"; }</pre></div> <div>Output:</div>

		<p>The space between Hello and World is not alphanumeric.</p>
<code>\s</code>	<p>Matches a whitespace character, which in ASCII are tab, line feed, form feed, carriage return, and space;</p> <p>in Unicode, also matches no-break spaces, next line, and the variable-width spaces (amongst others).</p>	<pre>\$string1 = "Hello World\n"; if (\$string1 =~ m/\s.*\s/) { print "In \$string1 there are TWO whitespace characters, which may"; print " be separated by other characters.\n"; }</pre> <p>Output:</p> <pre>In Hello World there are TWO whitespace characters, which may be separated by other characters.</pre>
<code>\S</code>	<p>Matches anything <i>but</i> a whitespace.</p>	<pre>\$string1 = "Hello World\n"; if (\$string1 =~ m/\S.*\S/) { print "In \$string1 there are TWO non-whitespace characters, which"; print " may be separated by other characters.\n"; }</pre> <p>Output:</p> <pre>In Hello World there are TWO non- whitespace characters, which may be separated by other characters.</pre>
<code>\d</code>	<p>Matches a digit;</p> <p>same as <code>[0-9]</code> in ASCII;</p> <p>in Unicode, same as the <code>\p{Digit}</code> or <code>\p{GC=Decimal_Number}</code> property, which itself the same as the <code>\p{Numeric_Type=Decimal}</code> property.</p>	<pre>\$string1 = "99 bottles of beer on the wall."; if (\$string1 =~ m/(\d+)/) { print "\$1 is the first number in '\$string1'\n"; }</pre> <p>Output:</p> <pre>99 is the first number in '99 bottles of beer on the wall.'</pre>
<code>\D</code>	<p>Matches a non-digit;</p> <p>same as <code>[^0-9]</code> in ASCII or <code>\P{Digit}</code> in Unicode.</p>	<pre>\$string1 = "Hello World\n"; if (\$string1 =~ m/\D/) { print "There is at least one character in \$string1"; print " that is not a</pre>

		<div>digit.\n"; }</div> <div>Output:</div> <div>There is at least one character in Hello World that is not a digit.</div>
<div>^</div>	Matches the beginning of a line or string.	<div>\$string1 = "Hello World\n"; if (\$string1 =~ m/^He/) { print "\$string1 starts with the characters 'He'.\n"; }</div> <div>Output:</div> <div>Hello World starts with the characters 'He'.</div>
<div>\$</div>	Matches the end of a line or string.	<div>\$string1 = "Hello World\n"; if (\$string1 =~ m/rld\$/) { print "\$string1 is a line or string "; print "that ends with 'rld'.\n"; }</div> <div>Output:</div> <div>Hello World is a line or string that ends with 'rld'.</div>
<div>\A</div>	Matches the beginning of a string (but not an internal line).	<div>\$string1 = "Hello\nWorld\n"; if (\$string1 =~ m/\AH/) { print "\$string1 is a string "; print "that starts with 'H'.\n"; }</div> <div>Output:</div> <div>Hello World is a string that starts with 'H'.</div>
<div>\Z</div>	Matches the end of a string (but not an internal line). ^[72]	<div>\$string1 = "Hello\nWorld\n"; if (\$string1 =~ m/d\n\Z/) { print "\$string1 is a string "; }</div>

4/8/24, 11:12 PM

Regular expression - Wikipedia

		<pre>print "that ends with 'd\\n'.\\n"; }</pre> <div>Output:<div>Hello World is a string that ends with 'd\\n'.</div></div>
<div>[^...]</div>	Matches every character except the ones inside brackets.	<pre>\$string1 = "Hello World\\n"; if (\$string1 =~ m/[^abc]/) { print "\$string1 contains a character other than "; print "a, b, and c.\\n"; }</pre> <div>Output:<div>Hello World contains a character other than a, b, and c.</div></div>

Induction

Regular expressions can often be created ("induced" or "learned") based on a set of example strings. This is known as the [induction of regular languages](#) and is part of the general problem of [grammar induction](#) in [computational learning theory](#). Formally, given examples of strings in a regular language, and perhaps also given examples of strings *not* in that regular language, it is possible to induce a grammar for the language, i.e., a regular expression that generates that language. Not all regular languages can be induced in this way (see [language identification in the limit](#)), but many can. For example, the set of examples {1, 10, 100}, and negative set (of counterexamples) {11, 1001, 101, 0} can be used to induce the regular expression 1·0* (1 followed by zero or more 0s).

See also

- [Comparison of regular expression engines](#)
- [Extended Backus–Naur form](#)
- [Matching wildcards](#)
- [Regular tree grammar](#)
- [Thompson's construction](#) – converts a regular expression into an equivalent [nondeterministic finite automaton](#) (NFA)

Notes

1. Goyvaerts, Jan. "Regular Expression Tutorial - Learn How to Use Regular Expressions" (<http://web.archive.org/web/20161101212501/http://www.regular-expressions.info/tutorial.html>) [Regular-Expressions.info](#). Archived from the original (<http://www.regular-expressions.info/tutorial.html>) [on 2016-11-01](#). Retrieved 2016-10-31.
2. Mitkov, Ruslan (2003). *The Oxford Handbook of Computational Linguistics* (<https://books.google.com/books?id=yl6AnaKtVAkC&pg=PA754>) [Oxford University Press](#). p. 754. ISBN 978-0-19-927634-9. Archived (<https://web.archive.org/web/20170228030346/https://books.google.com/books?id=yl6AnaKtVAkC&pg=PA754>) [from the original on 2017-02-28](#). Retrieved 2016-07-25.
3. Lawson, Mark V. (17 September 2003). *Finite Automata* (https://books.google.com/books?id=MDQ_K7-z2AMC&pg=PA98) [CRC Press](#). pp. 98–100. ISBN 978-1-58488-255-8. Archived (https://web.archive.org/web/20170227195128/https://books.google.com/books?id=MDQ_K7-z2AMC&pg=PA98) [from the original on 27 February 2017](#). Retrieved 25 July 2016.
4. "How a Regex Engine Works Internally" (<https://www.regular-expressions.info/engine.html>) [regular-expressions.info](#). Retrieved 24 February 2024.
5. "How Do You Actually Use Regex?" (<https://www.howtogeek.com/devops/how-do-you-actually-use-regex/>) [howtogeek.com](#). Retrieved 24 February 2024.
6. Kleene 1951.
7. Leung, Hing (16 September 2010). "Regular Languages and Finite Automata" (<https://web.archive.org/web/20131205193130/https://www.cs.nmsu.edu/historical-projects/Projects/kleene.9.16.10.pdf>) [PDF](#) (PDF). *New Mexico State University*. Archived from the original (<https://www.cs.nmsu.edu/historical-projects/Projects/kleene.9.16.10.pdf>) [PDF](#) (PDF) [on 5 December 2013](#). Retrieved 13 August 2019. "The concept of regular events was introduced by Kleene via the definition of regular expressions."
8. Thompson 1968.
9. Johnson et al. 1968.
10. Kernighan, Brian (2007-08-08). "A Regular Expressions Matcher" (<http://www.cs.princeton.edu/courses/archive/spr09/cos333/beautiful.html>) [Beautiful Code](#). O'Reilly Media. pp. 1–2. ISBN 978-0-596-51004-6. Archived (<https://web.archive.org/web/20201007183137/https://www.cs.princeton.edu/courses/archive/spr09/cos333/beautiful.html>) [from the original on 2020-10-07](#). Retrieved 2013-05-15.
11. Ritchie, Dennis M. "An incomplete history of the QED Text Editor" (<https://web.archive.org/web/19990221023422/http://cm.bell-labs.com/who/dmr/qed.html>) [Archived from the original](#) (<http://cm.bell-labs.com/who/dmr/qed.html>) [on 1999-02-21](#). Retrieved 9 October 2013.
12. Aho & Ullman 1992, 10.11 Bibliographic Notes for Chapter 10, p. 589.
13. Aycok 2003, p. 98.
14. Raymond, Eric S. citing Dennis Ritchie (2003). "Jargon File 4.4.7: grep" (<https://web.archive.org/web/20110605165512/http://www.catb.org/jargon/html/G/grep.html>) [Archived from the original](#) (<http://catb.org/jargon/html/G/grep.html>) [on 2011-06-05](#). Retrieved 2009-02-17.
15. "New Regular Expression Features in Tcl 8.1" (<http://www.tcl.tk/doc/howto/regexp81.html>) [Archived](#) (<https://web.archive.org/web/20201007183137/http://www.tcl.tk/doc/howto/regexp81.html>) [from the original on 2020-10-07](#). Retrieved 2013-10-11.

16. "Documentation: 9.3: Pattern Matching" (<http://www.postgresql.org/docs/9.3/interactive/functions-matching.html>) [↗](#). *PostgreSQL*. Archived (<https://web.archive.org/web/20201007183140/https://www.postgresql.org/docs/9.3/functions-matching.html>) [↗](#) from the original on 2020-10-07. Retrieved 2013-10-12.
17. Wall, Larry (2006). "Perl Regular Expressions" (<http://perldoc.perl.org/perlre.html>) [↗](#). *perlre*. Archived (<https://web.archive.org/web/20091231010052/http://perldoc.perl.org/perlre.html>) [↗](#) from the original on 2009-12-31. Retrieved 2006-10-10.
18. Wall (2002)
19. "PCRE - Perl Compatible Regular Expressions" (<https://www.pcre.org/>) [↗](#). *www.pcre.org*. Retrieved 2024-04-07.
20. "GRegex – Faster Analytics for Unstructured Text Data" (<https://grovf.com/products/gregex>) [↗](#). *grovf.com*. Archived (<https://web.archive.org/web/20201007183139/https://grovf.com/products/gregex>) [↗](#) from the original on 2020-10-07. Retrieved 2019-10-22.
21. "CUDA grep" (<http://bkase.github.io/CUDA-grep/finalreport.html>) [↗](#). *bkase.github.io*. Archived (<https://web.archive.org/web/20201007183138/http://bkase.github.io/CUDA-grep/finalreport.html>) [↗](#) from the original on 2020-10-07. Retrieved 2019-10-22.
22. Kerrisk, Michael. "grep(1) - Linux manual page" (<https://man7.org/linux/man-pages/man1/grep.1.html>) [↗](#). *man7.org*. Retrieved 31 January 2023.
23. Hopcroft, Motwani & Ullman (2000)
24. Sipser (1998)
25. Gelade & Neven (2008, p. 332, Thm.4.1)
26. Gruber & Holzer (2008)
27. Based on Gelade & Neven (2008), a regular expression of length about 850 such that its complement has a length about 2^{32} can be found at [File:RegexComplementBlowup.png](#).
28. "Regular expressions for deciding divisibility" (<https://s3.boskent.com/divisibility-regex/divisibility-regex.html>) [↗](#). *s3.boskent.com*. Retrieved 2024-02-21.
29. Gischer, Jay L. (1984). (Title unknown) (Technical Report). Stanford Univ., Dept. of Comp. Sc.
30. Hopcroft, John E.; Motwani, Rajeev & Ullman, Jeffrey D. (2003). *Introduction to Automata Theory, Languages, and Computation*. Upper Saddle River, New Jersey: Addison Wesley. pp. 117–120. ISBN 978-0-201-44124-6. "This property need not hold for extended regular expressions, even if they describe no larger class than regular languages; cf. p.121."
31. Kozen (1991)
32. Redko, V.N. (1964). "On defining relations for the algebra of regular events" (<http://umj.imath.kiev.ua/article/?article=10002>) [↗](#). *Ukrainskii Matematicheskii Zhurnal* (in Russian). **16** (1): 120–126. Archived (<https://web.archive.org/web/20180329121016/http://umj.imath.kiev.ua/article/?article=10002>) [↗](#) from the original on 2018-03-29. Retrieved 2018-03-28.
33. ISO/IEC 9945-2:1993 *Information technology – Portable Operating System Interface (POSIX) – Part 2: Shell and Utilities*, successively revised as ISO/IEC 9945-2:2002 *Information technology – Portable Operating System Interface (POSIX) – Part 2: System Interfaces*, ISO/IEC 9945-2:2003, and currently ISO/IEC/IEEE 9945:2009 *Information technology – Portable Operating System Interface (POSIX) Base Specifications, Issue 7*
34. The Single Unix Specification (Version 2)
35. "9.3.6 BREs Matching Multiple Characters" (https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html#tag_09_03_06) [↗](#). *The Open Group Base Specifications Issue 7, 2018 edition*. The Open Group. 2017. Retrieved December 10, 2023.

36. Ross Cox (2009). "Regular Expression Matching: the Virtual Machine Approach" (<https://swtch.com/~rsc/regexp/regexp2.html>) [↗](#). *swtch.com*. "Digression: POSIX Submatching"
37. "Perl Regular Expression Documentation" (<http://perldoc.perl.org/perlre.html#PCRE%2fPython-Support>) [↗](#). *perldoc.perl.org*. Archived (<https://web.archive.org/web/20091231010052/http://perldoc.perl.org/perlre.html#PCRE%2fPython-Support>) [↗](#) from the original on December 31, 2009. Retrieved January 8, 2012.
38. "Regular Expression Syntax" (<https://web.archive.org/web/20180718132241/https://docs.python.org/3/library/re.html#regular-expression-syntax>) [↗](#). *Python 3.5.0 documentation*. Python Software Foundation. Archived from the original (<https://docs.python.org/3/library/re.html#regular-expression-syntax>) [↗](#) on 18 July 2018. Retrieved 10 October 2015.
39. SRE: Atomic Grouping (?>...) is not supported #34627 (<https://github.com/python/cpython/issues/34627>) [↗](#)
40. "Essential classes: Regular Expressions: Quantifiers: Differences Among Greedy, Reluctant, and Possessive Quantifiers" (<https://docs.oracle.com/javase/tutorial/essential/regex/quant.html#difs>) [↗](#). *The Java Tutorials*. Oracle. Archived (<https://web.archive.org/web/20201007183203/https://docs.oracle.com/javase/tutorial/essential/regex/quant.html#difs>) [↗](#) from the original on 7 October 2020. Retrieved 23 December 2016.
41. "Atomic Grouping" (<https://www.regular-expressions.info/atomic.html>) [↗](#). *Regex Tutorial*. Archived (<https://web.archive.org/web/20201007183204/https://www.regular-expressions.info/atomic.html>) [↗](#) from the original on 7 October 2020. Retrieved 24 November 2019.
42. Bormann, Carsten; Bray, Tim. "I-Regexp: An Interoperable Regular Expression Format" (<https://www.rfc-editor.org/rfc/rfc9485.html>) [↗](#). *IETF*. Internet Engineering Task Force. Retrieved 11 March 2024.
43. Cezar Câmpeanu; Kai Salomaa & Sheng Yu (Dec 2003). "A Formal Study of Practical Regular Expressions" (<http://137.149.157.5/Articles/index.php?aid=1>) [↗](#). *International Journal of Foundations of Computer Science*. **14** (6): 1007–1018. doi:10.1142/S012905410300214X (<https://doi.org/10.1142/S012905410300214X>) [↗](#). Archived (<https://web.archive.org/web/20150704141706/http://137.149.157.5/Articles/index.php?aid=1>) [↗](#) from the original on 2015-07-04. Retrieved 2015-07-03. Theorem 3 (p.9)
44. "Perl Regular Expression Matching is NP-Hard" (<https://perl.plover.com/NPC/>) [↗](#). *perl.plover.com*. Archived (<https://web.archive.org/web/20201007183205/https://perl.plover.com/NPC/>) [↗](#) from the original on 2020-10-07. Retrieved 2019-11-21.
45. Ritchie, D. M.; Thompson, K. L. (June 1970). *QED Text Editor* (<https://wayback.archive-it.org/all/20150203071645/http://cm.bell-labs.com/cm/cs/who/dmr/qedman.pdf>) [↗](#) (PDF). MM-70-1373-3. Archived from the original (<http://cm.bell-labs.com/cm/cs/who/dmr/qedman.pdf>) [↗](#) (PDF) on 2015-02-03. Retrieved 2022-09-05. Reprinted as "QED Text Editor Reference Manual", MHCC-004, Murray Hill Computing, Bell Laboratories (October 1972).
46. Wall, Larry (1994-10-18). "Perl 5: perlre.pod" (<https://github.com/Perl/perl5/blob/a0d0e21ea6ea90a22318550944fe6cb09ae10cda/pod/perlre.pod>) [↗](#). *GitHub*.
47. Wandering Logic. "How to simulate lookaheads and lookbehinds in finite state automata?" (<https://cs.stackexchange.com/a/40058>) [↗](#). *Computer Science Stack Exchange*. Archived (<https://web.archive.org/web/20201007183206/https://cs.stackexchange.com/questions/2557/how-to-simulate-backreferences-lookaheads-and-lookbehinds-in-finite-state-auto/40058>) [↗](#) from the original on 7 October 2020. Retrieved 24 November 2019.
48. Zakharevich, Ilya (1997-11-19). "Jumbo Regexp Patch Applied (with Minor Fix-Up Tweaks): Perl/perl5@c277df4" (<https://github.com/Perl/perl5/commit/c277df42229d99fecbc76f5da53793a409ac66e1>) [↗](#). *GitHub*.
49. Cox (2007)





















50. Laurikari (2009)

51. "gnulib/lib/dfa.c" (<https://web.archive.org/web/20210818191338/https://git.savannah.gnu.org/gitweb/?p=gnulib.git%3Ba%3Dblob%3Bf%3Dlib%2Fdfa.c>) [🔗](#). Archived from the original (<https://git.savannah.gnu.org/gitweb/?p=gnulib.git;a=blob;f=lib/dfa.c>) [🔗](#) on 2021-08-18. Retrieved 2022-02-12. "If the scanner detects a transition on backref, it returns a kind of "semi-success" indicating that the match will have to be verified with a backtracking matcher."
52. Kearns, Steven (August 2013). "Sublinear Matching With Finite Automata Using Reverse Suffix Scanning". arXiv:1308.3822 (<https://arxiv.org/abs/1308.3822>) [📄](#) [cs.DS (<https://arxiv.org/archive/cs/DS>) [🔗](#)].
53. Navarro, Gonzalo (10 November 2001). "NR-grep: a fast and flexible pattern-matching tool" (<https://users.dcc.uchile.cl/~gnavarro/ps/spe01.pdf>) [📄](#) (PDF). *Software: Practice and Experience*. **31** (13): 1265–1312. doi:10.1002/spe.411 (<https://doi.org/10.1002%2Fspe.411>) [🔗](#). S2CID 3175806 (<https://api.semanticscholar.org/CorpusID:3175806>) [🔗](#). Archived (<https://web.archive.org/web/20201007183210/https://users.dcc.uchile.cl/~gnavarro/ps/spe01.pdf>) [📄](#) (PDF) from the original on 7 October 2020. Retrieved 21 November 2019.
54. "travisdowns/polyregex" (<https://github.com/travisdowns/polyregex>) [🔗](#). *GitHub*. 5 July 2019. Archived (<https://web.archive.org/web/20200914170309/https://github.com/travisdowns/polyregex>) [🔗](#) from the original on 14 September 2020. Retrieved 21 November 2019.
55. Schmid, Markus L. (March 2019). "Regular Expressions with Backreferences: Polynomial-Time Matching Techniques". arXiv:1903.05896 (<https://arxiv.org/abs/1903.05896>) [📄](#) [cs.FL (<https://arxiv.org/archive/cs/FL>) [🔗](#)].
56. "Vim documentation: pattern" (<http://vimdoc.sourceforge.net/html/doc/pattern.html#/%5B%5D>) [🔗](#). Vimdoc.sourceforge.net. Archived (<https://web.archive.org/web/20201007183210/http://vimdoc.sourceforge.net/html/doc/pattern.html#/%5B%5D>) [🔗](#) from the original on 2020-10-07. Retrieved 2013-09-25.
57. "UTS#18 on Unicode Regular Expressions, Annex A: Character Blocks" (http://unicode.org/reports/tr18/#Character_Blocks) [🔗](#). Archived (https://web.archive.org/web/20201007183210/http://unicode.org/reports/tr18/#Character_Blocks) [🔗](#) from the original on 2020-10-07. Retrieved 2010-02-05.
58. "regex(3) - Linux manual page" (<https://man7.org/linux/man-pages/man3/regcomp.3.html>) [🔗](#). *man7.org*. Retrieved 2022-04-27.
59. "Regular expressions library - cppreference.com" (<https://en.cppreference.com/w/cpp/regex>) [🔗](#). *en.cppreference.com*. Retrieved 2022-04-27.
60. "Regular Expression Language - Quick Reference" (<https://learn.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference>) [🔗](#). *microsoft.com*. Retrieved 2024-02-20.
61. "Pattern (Java Platform SE 7)" (<https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html>) [🔗](#). *docs.oracle.com*. Retrieved 2022-04-27.
62. "Regular expressions - JavaScript" (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions) [🔗](#). *MDN*. Retrieved 2022-04-27.
63. "OCaml library: Str" (<https://v2.ocaml.org/api/Str.html>) [🔗](#). *v2.ocaml.org*. Retrieved 2022-08-21.
64. "perlre" (<https://perldoc.perl.org/perlre>) [🔗](#). *perldoc.perl.org*. Retrieved 2023-02-04.
65. "PHP: PCRE - Manual" (<https://www.php.net/manual/en/book.pcre.php>) [🔗](#). *www.php.net*. Retrieved 2023-02-04.

66. "re – Regular expression operations" (<https://docs.python.org/3/library/re.html>) *docs.python.org*. Retrieved 2023-02-24.
67. "Regex on crates.io" (<https://crates.io/crates/regex>) *Crates.io*. Archived (<https://web.archive.org/web/20221129000056/https://crates.io/crates/regex>) from the original on 2022-11-29. Retrieved 2023-02-24.
68. Horowitz, Bradley (24 October 2011). "A fall sweep" (<https://googleblog.blogspot.com/2011/10/fall-sweep.html>) *Google Blog*. Archived (<https://web.archive.org/web/20181021074737/https://googleblog.blogspot.com/2011/10/fall-sweep.html>) from the original on 21 October 2018. Retrieved 4 May 2019.
69. The character 'm' is not always required to specify a Perl match operation. For example, `m/[^abc]/` could also be rendered as `/[^abc]/`. The 'm' is only necessary if the user wishes to specify a match operation without using a forward-slash as the regex delimiter. Sometimes it is useful to specify an alternate regex delimiter in order to avoid "delimiter collision". See 'perldoc perlre (<http://perldoc.perl.org/perlre.html>) Archived (<https://web.archive.org/web/20091231010052/http://perldoc.perl.org/perlre.html>) 2009-12-31 at the Wayback Machine' for more details.
70. E.g., see *Java in a Nutshell*, p. 213; *Python Scripting for Computational Science*, p. 320; *Programming PHP*, p. 106.
71. All the if statements return a TRUE value
72. Conway, Damian (2005). "Regular Expressions, End of String" (<https://www.scribd.com/doc/15491004/Perl-Best-Practices>) *Perl Best Practices*. O'Reilly. p. 240. ISBN 978-0-596-00173-5. Archived (<https://web.archive.org/web/20201007183212/https://www.scribd.com/doc/15491004/Perl-Best-Practices-Standards-and-Styles-for-Developing-Maintainable-Code>) from the original on 2020-10-07. Retrieved 2017-09-10.

References

- Aho, Alfred V. (1990). "Algorithms for finding patterns in strings". In van Leeuwen, Jan (ed.). *Handbook of Theoretical Computer Science, volume A: Algorithms and Complexity*. The MIT Press. pp. 255–300.
- Aho, Alfred V.; Ullman, Jeffrey D. (1992). "Chapter 10. Patterns, Automata, and Regular Expressions" (<http://infolab.stanford.edu/~ullman/focs/ch10.pdf>) (PDF). *Foundations of Computer Science* (<http://infolab.stanford.edu/~ullman/focs.html>) Archived (<https://web.archive.org/web/20201007183211/http://infolab.stanford.edu/~ullman/focs.html>) from the original on 2020-10-07. Retrieved 2013-12-14.
- Aycock, John (June 2003). "A brief history of just-in-time" (<https://www.cs.tufts.edu/comp/150C/MF/papers/aycock03jit.pdf>) (PDF). *ACM Computing Surveys*. **35** (2): 97–113. CiteSeerX 10.1.1.97.3985 (<https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.97.3985>) . doi:10.1145/857076.857077 (<https://doi.org/10.1145/857076.857077>) . S2CID 15345671 (<https://api.semanticscholar.org/CorpusID:15345671>) .
- "Regular Expressions". *The Single UNIX Specification, Version 2* (<http://pubs.opengroup.org/onlinepubs/007908799/xbd/re.html>) . The Open Group. 1997. Archived (<https://web.archive.org/web/20201007183212/https://pubs.opengroup.org/onlinepubs/007908799/xbd/re.html>) from the original on 2020-10-07. Retrieved 2011-12-13.
- "Chapter 9: Regular Expressions" (http://pubs.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap09.html) *The Open Group Base Specifications* (6). The Open Group. 2004. IEEE Std 1003.1, 2004 Edition. Archived (https://web.archive.org/web/20111202145637/http://pubs.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap09.html) from the original on 2011-12-02. Retrieved 2011-12-13.

- Cox, Russ (2007). "Regular Expression Matching Can Be Simple and Fast" (<https://web.archive.org/web/20100101190447/http://swtch.com/~rsc/regexp/regexp1.html>) . Archived from the original (<http://swtch.com/~rsc/regexp/regexp1.html>)  on 2010-01-01. Retrieved 2008-04-27.
- Forta, Ben (2004). *Sams Teach Yourself Regular Expressions in 10 Minutes*. Sams. ISBN 978-0-672-32566-3.
- Friedl, Jeffrey E. F. (2002). *Mastering Regular Expressions* (<http://regex.info/>) . O'Reilly. ISBN 978-0-596-00289-3. Archived (<https://web.archive.org/web/20050830113350/http://regex.info/>)  from the original on 2005-08-30. Retrieved 2005-04-26.
- Gelade, Wouter; Neven, Frank (2008). *Succinctness of the Complement and Intersection of Regular Expressions* (<https://web.archive.org/web/20110718225605/http://drops.dagstuhl.de/opus/volltexte/2008/1354/>) . *Proceedings of the 25th International Symposium on Theoretical Aspects of Computer Science (STACS 2008)*. pp. 325–336. arXiv:0802.2869 (<https://arxiv.org/abs/0802.2869>) . Archived from the original (<http://drops.dagstuhl.de/opus/volltexte/2008/1354>)  on 2011-07-18. Retrieved 2009-06-15.
- Goyvaerts, Jan; Levithan, Steven (2009). *Regular Expressions Cookbook*. [O'reilly]. ISBN 978-0-596-52068-7.
- Gruber, Hermann; Holzer, Markus (2008). *Finite Automata, Digraph Connectivity, and Regular Expression Size* (<http://www.hermann-gruber.com/data/icalp08.pdf>)  (PDF). *Proceedings of the 35th International Colloquium on Automata, Languages and Programming (ICALP 2008)*. Lecture Notes in Computer Science. Vol. 5126. pp. 39–50. doi:10.1007/978-3-540-70583-3_4 (https://doi.org/10.1007/978-3-540-70583-3_4) . ISBN 978-3-540-70582-6. Archived (<https://web.archive.org/web/20110711163607/http://www.hermann-gruber.com/data/icalp08.pdf>)  (PDF) from the original on 2011-07-11. Retrieved 2011-02-03.
- Habibi, Mehran (2004). *Real World Regular Expressions with Java 1.4*. Springer. ISBN 978-1-59059-107-9.
- Hopcroft, John E.; Motwani, Rajeev; Ullman, Jeffrey D. (2000). *Introduction to Automata Theory, Languages, and Computation* (2nd ed.). Addison-Wesley.
- Johnson, Walter L.; Porter, James H.; Ackley, Stephanie I.; Ross, Douglas T. (1968). "Automatic generation of efficient lexical processors using finite state techniques" (<https://doi.org/10.1145/2F364175.364185>) . *Communications of the ACM*. **11** (12): 805–813. doi:10.1145/364175.364185 (<https://doi.org/10.1145/364175.364185>) . S2CID 17253809 (<https://api.semanticscholar.org/CorpusID:17253809>) .
- Kleene, Stephen C. (1951). "Representation of Events in Nerve Nets and Finite Automata". In Shannon, Claude E.; McCarthy, John (eds.). *Automata Studies* (https://www.rand.org/content/dam/rand/pubs/research_memoranda/2008/RM704.pdf)  (PDF). Princeton University Press. pp. 3–42. Archived (https://web.archive.org/web/20201007183213/https://www.rand.org/content/dam/rand/pubs/research_memoranda/2008/RM704.pdf)  (PDF) from the original on 2020-10-07. Retrieved 2017-12-10.
- Kozen, Dexter (1991). "A completeness theorem for Kleene algebras and the algebra of regular events". [1991] *Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*. pp. 214–225. doi:10.1109/LICS.1991.151646 (<https://doi.org/10.1109/2FLICS.1991.151646>) . hdl:1813/6963 (<https://hdl.handle.net/1813/6963>) . ISBN 978-0-8186-2230-4. S2CID 19875225 (<https://api.semanticscholar.org/CorpusID:19875225>) .
- Laurikari, Ville (2009). "TRE library 0.7.6" (<https://web.archive.org/web/20100714224701/http://laurikari.net/tre/>) . Archived from the original (<https://www.laurikari.net/tre/>)  on 2010-07-14. Retrieved 2009-04-01.
- Liger, François; McQueen, Craig; Wilton, Paul (2002). *Visual Basic .NET Text Manipulation Handbook*. Wrox Press. ISBN 978-1-86100-730-8.

- Sipser, Michael (1998). "Chapter 1: Regular Languages" (<https://archive.org/details/introductiontoth00sips/page/31>) . *Introduction to the Theory of Computation*. PWS Publishing. pp. 31–90 (<https://archive.org/details/introductiontoth00sips/page/31>) . ISBN 978-0-534-94728-6.
- Stubblebine, Tony (2003). *Regular Expression Pocket Reference*. O'Reilly. ISBN 978-0-596-00415-6.
- Thompson, Ken (1968). "Programming Techniques: Regular expression search algorithm" (<http://doi.org/10.1145%2F363347.363387>) . *Communications of the ACM*. **11** (6): 419–422. doi:10.1145/363347.363387 (<https://doi.org/10.1145%2F363347.363387>) . S2CID 21260384 (<https://api.semanticscholar.org/CorpusID:21260384>) .
- Wall, Larry (2002). "Apocalypse 5: Pattern Matching" (<http://dev.perl.org/perl6/doc/design/apo/A05.html>) . Archived (<https://web.archive.org/web/20100112232513/http://dev.perl.org/perl6/doc/design/apo/A05.html>)  from the original on 2010-01-12. Retrieved 2006-10-11.

External links

-  Media related to [Regex](#) at Wikimedia Commons
- [Regular Expressions](https://curlie.org/Computers/Programming/Languages/Regular_Expressions) (https://curlie.org/Computers/Programming/Languages/Regular_Expressions)  at [Curlie](#)
- ISO/IEC 9945-2:1993 *Information technology – Portable Operating System Interface (POSIX) – Part 2: Shell and Utilities* (http://www.iso.org/iso/catalogue_detail.htm?csnumber=17841) 
- ISO/IEC 9945-2:2002 *Information technology – Portable Operating System Interface (POSIX) – Part 2: System Interfaces* (http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=37313) 
- ISO/IEC 9945-2:2003 *Information technology – Portable Operating System Interface (POSIX) – Part 2: System Interfaces* (http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=38790) 
- ISO/IEC/IEEE 9945:2009 *Information technology – Portable Operating System Interface (POSIX) Base Specifications, Issue 7* (http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=50516) 
- Regular Expression, IEEE Std 1003.1-2017, Open Group (http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html) 