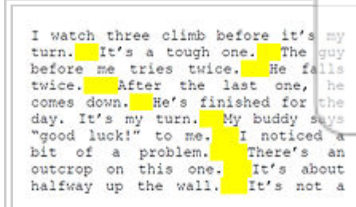
Regular expression

From Wikipedia, the free encyclopedia

In theoretical computer science and formal language theory, a regular expression (abbreviated regex or regexp and sometimes called a rational expression)[1][2] is a sequence of characters that define a search pattern, mainly for use in pattern matching with strings, or string matching, i.e. "find and replace"-like operations. The concept arose in the 1950s, when the American mathematician Stephen Kleene formalized the description of a *regular language*, and came into common use with the Unix text processing utilities ed, an editor, and grep (global regular expression print), a filter.



The regular expression `(?<=\.) {2,}(?=[A-Z])` matches at least two spaces occurring after period (.) and before an upper case letter as highlighted in the text above.

Regular expressions are so useful in computing that the various systems to specify regular expressions have evolved to provide both a *basic* and *extended* standard for the grammar and syntax; *modern* regular expressions heavily augment the standard. Regular expression processors are found in several search engines, search and replace dialogs of several word processors and text editors, and in the command lines of text processing utilities, such as sed and AWK.

Many programming languages provide regular expression capabilities, some built-in, for example Perl, JavaScript, Ruby, AWK, and Tcl, and others via a standard library, for example .NET languages, Java, Python, POSIX C and C++ (since C++11). Most other languages offer regular expressions via a library.

Contents

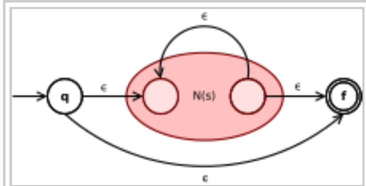
Patterns

Each character in a regular expression is understood to be: a metacharacter (with its special meaning), or a regular character (with its literal meaning). Together, they can be used to identify textual material of a given pattern, or process a number of instances of it. Pattern-matches can vary from a precise equality to a very general similarity (controlled by the metacharacters). The metacharacter syntax is designed specifically to represent prescribed targets in the most concise and flexible way to direct the automation of text processing of general text files, specific textual forms, or of random input strings.

A very simple use of a regular expression would be: to locate the same word spelled two different ways in a text editor (for example the regular expression `seriali[sz]e` matches both "serialise" and "serialize"). Wildcards could also achieve this, but are more limited in what they can pattern (having fewer metacharacters and a simple language-base).

A usual context of wildcard characters is in globbing similar names in a list of files, whereas regular expressions are usually employed in applications that pattern-match text strings in general. For example, the regexp `^[ \t]+|[ \t]+$` matches excess whitespace at the beginning or end of a line. An advanced regexp used to match any numeral is `^[+-]?(\d+(\.\d+)?|\.\d+)([eE][+-]?\d+)?$`. See *Examples* for more examples.

A regular expression processor translates a regular expression into a nondeterministic finite automaton (NFA), which is then made deterministic and 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 regex $s*$, where s denotes a simpler regex in turn, which has already been recursively translated to the NFA $N(s)$.



Translating the Kleene star "*s**": "zero or more of *s*".

History

Regular expressions originated in 1956, when mathematician Stephen Cole Kleene described regular languages using his mathematical notation called *regular sets*.[3] 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 syntax.

Regular expressions entered popular use from 1968 in two uses: pattern matching in a text editor[4] and lexical analysis in a compiler.[5] 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.[4][6][7][8] 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.[9] 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"[10]). 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.[5]

Many variations of these original forms of regular expressions were used in Unix[8] programs at Bell Labs in the 1970s, including vi, lex, sed, AWK, and expr, and in other programs such as Emacs. Regular expressions were subsequently adopted by a wide range of programs, with these early forms standardized in the POSIX.2 standard in 1992.

In the 1980s more complicated regular expressions arose in Perl, which originally derived from a regex library written by Henry Spencer (1986), who later wrote an implementation of *Advanced Regular Expressions* for Tcl.[11] The Tcl library is a hybrid NFA/DFA implementation with improved performance characteristics, earning praise from Jeffrey Friedl who said, "...it really seems quite wonderful."[12] Software projects that have adopted Spencer's Tcl regular expression implementation include PostgreSQL.[13] Perl later expanded on Spencer's original library to add many new features,[14] but has not yet caught up with Spencer's Advanced Regular Expressions implementation in terms of performance or Unicode handling.[15][16] Part of the effort in the design of Perl 6 is to improve Perl's regular expression integration, and to increase their scope and capabilities to allow the definition of parsing expression grammars.[17] The result is a mini-language called Perl 6 rules, which are used to define Perl 6 grammar as well as provide a tool to programmers in the language. These rules maintain existing features of Perl 5.x regular expressions, but also allow BNF-style definition of a recursive descent parser via sub-rules.

The use of regular expressions 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 regular expressions. Its use is evident in the DTD element group syntax.

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

Today regular expressions are widely supported in programming languages, text processing programs (particular lexers), advanced text editors, and some other programs. Regular expression 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. Implementations of regular expression functionality is often called a regular expression engine, and a number of libraries are available for reuse.

Basic concepts

A regular expression, often called a pattern, is an expression used to specify 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 to specify the desired set of strings. 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. In most formalisms, if there exists at least one regex that matches a particular set then there exists an infinite number of other regex that also match it—the specification is not unique. 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 a token (such as a character) or group specifies how often that preceding element is allowed to occur. 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}[18]	The preceding item is matched exactly *n* times.

{min,}[18]	The preceding item is matched *min* or more times.
{min,max}[18]	The preceding item is matched at least *min* times, but not more than *max* times.

These constructions can be combined to form arbitrarily complex expressions, much like one can construct arithmetical expressions from numbers and the operations +, −, ×, and ÷. For example, `H(ae?|ä)ndel` and `H(a|ae|ä)ndel` are both valid patterns which match the same strings as the earlier example, `H(ä|ae?)ndel`.

The precise syntax for regular expressions varies among tools and with context; more detail is given in the *Syntax* section.

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 and operator symbols that denote sets of strings and operations over these sets, respectively. The following definition is standard, and found as such in most textbooks on formal language theory.[19][20] Given a finite alphabet Σ, the following constants are defined as regular expressions:

- (*empty set*) ∅ denoting the set ∅.
- (*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 in R and a string in S. For example {"ab", "c"}{"d", "ef"} = {"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 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 set described by *R*. For example, {"0","1"}* is the set of all finite binary strings (including the empty string), and {"ab", "c"}* = {ε, "ab", "c", "abab", "abc", "cab", "cc", "ababab", "abcab", ... }.

To avoid parentheses it is assumed that the Kleene star has the highest priority, then concatenation and 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 ∪, +, or ∨ for alternation instead of the vertical bar.

Examples:

- `a|b*` denotes {ε, "a", "b", "bb", "bbb", ...}
- `(a|b)*` denotes the set of all strings with no symbols other than "a" and "b", including the empty string: {ε, "a", "b", "aa", "ab", "ba", "bb", "aaa", ...}
- `ab*(c|ε)` 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", ...}
- `(0|(1(01*0)*1))*` denotes the set of binary numbers that are multiples of 3: { ε, "0", "00", "11", "000", "011", "110", "0000", "0011", "0110", "1001", "1100", "1111", "00000", ... }

Expressive power and compactness

The formal definition of regular expressions is purposely parsimonious and avoids defining the redundant quantifiers ? and +, which can be expressed as follows: a+ = aa*, and a? = (a|ε). 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, as it can always be circumscribed by using the other operators. However, the process for computing such a representation is complex, and the result may require expressions of a size that is double exponentially larger.[21][22]

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 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

$$(a \mid b)^*a\underbrace{(a \mid b)(a \mid b)\cdots(a \mid b)}_{k-1 \text{ times}}.$$

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.[19]

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; 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).

The redundancy can be eliminated by using Kleene star and set union to find an interesting subset of regular expressions that is still fully expressive, but perhaps their use can be restricted. 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 with Kleene algebra;[23] see Kleene algebra#History for details.

Syntax

A regexp *pattern* matches a target *string*. The pattern is composed of a sequence of *atoms*. An atom is a single point within the regexp 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 back references 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 regular expression 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 regexp 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 regular expressions use metacharacters more often than their literal meaning, so to avoid "backslash-osis" it makes sense to have a metacharacter escape to a literal mode; but starting out, it makes more sense to 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 `dsw.DSW` and `N`.

Delimiters

When entering a regular expression 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 regular expression `re` is entered as `"re"`. However, they are often written with slashes as delimiters, as in `/re/` for the regular expression `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/regexp/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.

Standards

The IEEE POSIX standard has three sets of compliance: BRE,[24] ERE, and SRE for Basic, Extended, and Simple Regular Expressions. SRE is deprecated,[25] 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 regular expressions 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 regular expressions.

Perl regular expressions have become a de facto standard, having a rich and powerful set of atomic expressions. Perl has no "basic" or "extended" levels, where the `( )` and `{ }` may or may not have literal meanings. They are always metacharacters, as they are in "extended" mode for POSIX. To get their *literal* meaning, you escape them. Other metacharacters are known to be literal or symbolic based on context alone. Perl offers much more functionality: "lazy" regular expressions, backtracking, named capture groups, and recursive patterns, all of which are powerful additions to POSIX BRE/ERE. (See lazy quantification below.)

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
.	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, `a.c` matches "abc", etc., but `[a.c]` matches only "a", ".", or "c".
[ ]	A bracket expression. Matches a single character that is contained within the brackets. For example, `[abc]` matches "a", "b", or "c". `[a-z]` specifies a range which matches any lowercase letter from "a" to "z". These forms can be mixed: `[abcx-z]` matches "a", "b", "c", "x", "y", or "z", as does `[a-cx-z]`. The `-` character is treated as a literal character if it is the last or the first (after the ^, if present) character within the brackets: `[abc-]`, `[-abc]`. Note that backslash escapes are not allowed. The `]` character can be included in a bracket expression if it is the first (after the ^) character: `[]abc]`.
[^]	Matches a single character that is not contained within the brackets. For example, `[^abc]` matches any character other than "a", "b", or "c". `[^a-z]` matches any single character that is not a lowercase letter from "a" to "z". Likewise, literal characters and ranges can be mixed.
^	Matches the starting position within the string. In line-based tools, it matches the starting position of any line.
$	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.
( )	Defines a marked subexpression. The string matched within the parentheses can be recalled later (see the next entry, *n*). A marked subexpression is also called a block or capturing group. BRE mode requires **\(\)**.
n	Matches what the *n*th marked subexpression matched, where *n* is a digit from 1 to 9. This construct is vaguely defined in the POSIX.2 standard. Some tools allow referencing more than nine capturing groups.
*****	Matches the preceding element zero or more times. For example, `ab*c` matches "ac", "abc", "abbbc", etc. `[xyz]*` matches "", "x", "y", "z", "zx", "zyx", "xyzzy", and so on. `(ab)*` matches "", "ab", "abab", "ababab", and so on.
{*m*,*n*}	Matches the preceding element at least *m* and not more than *n* times. For example, `a{3,5}` matches only "aaa", "aaaa", and "aaaaa". This is not found in a few older instances of regular expressions. BRE mode requires **\{*m*,*n*\}**.

Examples:

- `.at` matches any three-character string ending with "at", including "hat", "cat", and "bat".
- `[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]" and "[b]".
- `s.*` matches s followed by zero or more characters, for example: "s" and "saw" and "seed".

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

Examples:

- [hc]+at matches "hat", "cat", "hhat", "chat", "hcat", "cchchat", and so on, but not "at".
- [hc]?at matches "hat", "cat", and "at".
- [hc]*at matches "hat", "cat", "hhat", "chat", "hcat", "cchchat", "at", and so on.
- 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 regular expression 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 the upper case 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 upper-case *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 regular expression processor installed. Those definitions are in the following table:

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

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.

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

Perl

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, Python, Ruby, Microsoft's .NET Framework, and XML Schema. Some languages and tools such as Boost and PHP support multiple regular expression flavors. Perl-derivative regular expression 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.[26]

Lazy matching

The three common quantifiers (`*`, `+` and `?`) are greedy by default because they match as many characters as possible.[27] The regular expression `".*"` applied to the string

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

matches the entire sentence instead of only the first quotation. The aforementioned quantifiers may therefore be made *lazy* or *minimal*, matching as few characters as possible, by appending a question mark: ".*?" matches only the first quotation.[27]

Patterns for non-regular languages

Many features found in modern regular expression libraries provide an expressive power that far 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 back references, as supported by numerous modern tools, is still context sensitive.[28]

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* or simply *pattern* to describe the latter. Larry Wall, author of the Perl programming language, writes in an essay about the design of Perl 6:

> '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).[17]

Fuzzy regular expressions

Variants of regular expressions can be used for working with text in natural language, when it is necessary to take into account possible typos and spelling variants. For example, the text "Julius Caesar" might be a fuzzy match for:

- Gaius Julius Caesar
- Yulius Cesar
- G. Juliy Caezar

In such cases the mechanism implements some fuzzy string matching algorithm and possibly some algorithm for finding the similarity between text fragment and pattern.

This task is closely related to both full text search and named entity recognition.

Some software libraries work with fuzzy regular expressions:

- TRE — well-developed portable free project in C, which uses syntax similar to POSIX
- FREJ — open source project in Java with non-standard syntax (which utilizes prefix, Lisp-like notation), targeted to allow easy use of substitutions of inner matched fragments in outer blocks, but lacks many features of standard regular expressions.
- agrep — command-line utility (proprietary, but free for non-commercial usage).

Implementations and running times

There are at least three different algorithms that decide whether and how a given regular expression 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 $O(n)$.

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 $O(m\ n)$. 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.[29][30]

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.

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.

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, regular expressions were originally written to use ASCII characters as their token set though regular expression libraries have supported numerous other character sets. Many modern regular expression 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 regular expressions to support Unicode.

- Supported encoding. Some regular expression 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 regular expression engines support only the Basic Multilingual Plane, that is, the characters which can be encoded with only 16 bits. Currently, only a few regular expression 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 [0,0x10FFFF]. 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.[31]

- 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 letters can be followed by one of more non-spacing symbols (usually diacritics like accent marks) to form a single printing character, but also provides precomposed characters, i.e. characters that already include one or more combining characters. A sequence of a character + combining character should be matched with the identical single precomposed character. The process of standardizing sequences of characters + combining characters 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.[32] In Perl and the `java.util.regex` (https://docs.oracle.com/javase/8/docs/api/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 upper-case 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}`.

Uses

Regular expressions 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 regular expressions 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: Google Code Search, Exalead. However, Google Code Search has been shut down as of March 2013.[33]

Examples

A regular expression is a string that is used to describe or match a set of strings according to certain syntax rules. 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.

Despite this variability, and because regular expressions can be difficult to both explain and understand without examples. Interactive web sites for testing regular expressions are a useful resource for learning regular expressions by experimentation. This section provides a basic description of some of the properties of regular expressions by way of illustration.

The following conventions are used in the examples.[34]

```
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 regular expressions 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.[35]

Meta-character(s)	Description	Example[36]

`.`	Normally matches any character except a newline.
	Within square brackets the dot is literal.
		`$string1 = "Hello World\n";`
		`if ($string1 =~ m/...../) {`
		`  print "$string1 has length >= 5\n";`
		`}`

`( )`	Groups a series of pattern elements to a single element. When you match a pattern within parentheses, you can use any of `$1`, `$2`, ... later to refer to the previously matched pattern.	`$string1 = "Hello World\n";`
		`if ($string1 =~ m/(H..).(o..)/) {`
		`  print "We matched '$1' and '$2'\n";`
		`}`
		Output:
		`We matched 'Hel' and 'o W'`

`+`	Matches the preceding pattern element one or more times.	`$string1 = "Hello World\n";`
		`if ($string1 =~ m/l+/) {`
		`  print "There are one or more consecutive letter \"l\"'s in $string1\n";`
		`}`
		Output:
		`There are one or more consecutive letter "l"'s in Hello World`

`?`	Matches the preceding pattern element zero or one times.	`$string1 = "Hello World\n";`
		`if ($string1 =~ m/H.?e/) {`
		`  print "There is an 'H' and a 'e' separated by ";`
		`  print "0-1 characters (Ex: He Hoe)\n";`
		`}`

`?`	Modifies the `*`, `+`, `?` or `{M,N}`'d regex that comes before to match as few times as possible.	`$string1 = "Hello World\n";`
		`if ($string1 =~ m/(l.+?o)/) {`
		`  print "The non-greedy match with 'l' followed by one or\n";`
		`  print "more characters is 'llo' rather than 'llo Wo'.\n";`
		`}`
		Output:
		`The non-greedy match with 'l' followed by one or`
		`more characters is 'llo' rather than 'llo Wo'.`

Induction

Main article: Induction of regular languages

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
- List of regular expression software — applications which support regular expressions
- Regular tree grammar
- Thompson's construction algorithm — converts a regular expression into an equivalent NFA
- Nondeterministic finite automaton

Notes

1. Ruslan Mitkov (2003). *The Oxford Handbook of Computational Linguistics*. Oxford University Press. p. 754. ISBN 978-0-19-927634-9.
2. Mark V. Lawson (17 September 2003). *Finite Automata*. CRC Press. pp. 98–100. ISBN 978-1-58488-255-8.
3. Kleene 1956.
4. Thompson 1968.
5. Johnson et al. 1968.
6. Kernighan, Brian. "A Regular Expressions Matcher". *Beautiful Code*. O'Reilly Media. pp. 1–2. ISBN 978-0-596-51004-6. Retrieved 2013-05-15.
7. Ritchie, Dennis M. "An incomplete history of the QED Text Editor". Retrieved 9 October 2013.
8. Aho & Ullman 1992, 10.11 Bibliographic Notes for Chapter 10, p. 589.
9. Aycock 2003, 2. JIT Compilation Techniques, 2.1 Genesis, p. 98.
10. Raymond, Eric S. citing Dennis Ritchie (2003). "Jargon File 4.4.7: grep".
11. "New Regular Expression Features in Tcl 8.1". Retrieved 2013-10-11.
12. Friedl, Jeffrey. "The Mechanics of Expression Processing". *Mastering Regular Expressions*. O'Reilly Media. p. 182. ISBN 0-596-52812-4.
13. "PostgreSQL 9.3.1 Documentation: 9.7. Pattern Matching". Retrieved 2013-10-12.
14. Wall, Larry and the Perl 5 development team (2006). "perlre: Perl regular expressions".
15. "http://perldoc.perl.org/perlreguts.html#Unicode-and-Localisation-Support". Retrieved 2013-10-11.
16. Russ Cox (2007). "Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...)". Retrieved 2013-10-11.
17. Wall (2002)
18. grep(1) man page
19. Hopcroft, Motwani & Ullman (2000)
20. Sipser (1998)
21. Gelade & Neven (2008)
22. Gruber & Holzer (2008)
23. Kozen (1991)
24. 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*
25. The Single Unix Specification (Version 2)
26. "Perl Regular Expression Documentation". perldoc.perl.org. Retrieved January 8, 2012.
27. "Regular Expression Syntax". *Python 3.5.0 documentation*. Python Software Foundation. Retrieved 10 October 2015.
28. Cezar Câmpeanu and Kai Salomaa, and Sheng Yu (Dec 2003). "A Formal Study of Practical Regular Expressions". *International Journal of Foundations of Computer Science* 14 (6): 1007–1018. Theorem 3 (p.9)
29. Cox (2007)
30. Laurikari (2009)
31. "Vim documentation: pattern". Vimdoc.sourceforge.net. Retrieved 2013-09-25.
32. "UTS#18 on Unicode Regular Expressions, Annex A: Character Blocks". Retrieved 2010-02-05.
33. http://stackoverflow.com/questions/7778034/replacement-for-google-code-search

34. 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)' for more details.
35. e.g., see *Java in a Nutshell* — Page 213, *Python Scripting for Computational Science* — Page 320, Programming PHP — Page 106
36. Note that all the if statements return a TRUE value
37. Conway, Damian (2005). "Regular Expressions, End of String". *Perl Best Practices*. O'Reilly. p. 240. ISBN 978-0-596-00173-5.

References

- Aho, Alfred V. (1990). van Leeuwen, Jan, ed. *Algorithms for finding patterns in strings. 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" (PDF). *Foundations of Computer Science*.
- "Regular Expressions". *The Single UNIX ® Specification, Version 2* (The Open Group). 1997.
- "Chapter 9: Regular Expressions". *The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition* (The Open Group). 2004.
- Cox, Russ (2007). "Regular Expression Matching Can Be Simple and Fast".
- Forta, Ben (2004). *Sams Teach Yourself Regular Expressions in 10 Minutes*. Sams. ISBN 0-672-32566-7.
- Friedl, Jeffrey (2002). *Mastering Regular Expressions*. O'Reilly. ISBN 0-596-00289-0.
- Gelade, Wouter; Neven, Frank (2008). *Succinctness of the Complement and Intersection of Regular Expressions. Proceedings of the 25th International Symposium on Theoretical Aspects of Computer Science (STACS 2008)*: 325—336.
- 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* (PDF). *Proceedings of the 35th International Colloquium on Automata, Languages and Programming (ICALP 2008)* 5126: 39—50. doi:10.1007/978-3-540-70583-3_4.
- Habibi, Mehran (2004). *Real World Regular Expressions with Java 1.4*. Springer. ISBN 1-59059-107-0.
- Hopcroft, John E.; Motwani, Rajeev; Ullman, Jeffrey D. (2000). *Introduction to Automata Theory, Languages, and Computation* (2nd ed.). Addison-Wesley.
- Johnson, W. L.; Porter, J. H.; Ackley, S. I.; Ross, D. T. (1968). "Automatic generation of efficient lexical processors using finite state techniques". *Communications of the ACM* 11 (12): 805—813. doi:10.1145/364175.364185.
- Kleene, Stephen C. (1956). Shannon, Claude E.; McCarthy, John, eds. *Representation of Events in Nerve Nets and Finite Automata. Automata Studies* (Princeton University Press). pp. 3—42.
- Kozen, Dexter (1991). "A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events". *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science (LICS 1991)*: 214—225.
- Laurikari, Ville (2009). "TRE library 0.7.6".
- Liger, Francois; Craig McQueen; Paul Wilton (2002). *Visual Basic .NET Text Manipulation Handbook*. Wrox Press. ISBN 1-86100-730-2.
- Sipser, Michael (1998). "Chapter 1: Regular Languages". *Introduction to the Theory of Computation*. PWS Publishing. pp. 31—90. ISBN 0-534-94728-X.
- Stubblebine, Tony (2003). *Regular Expression Pocket Reference*. O'Reilly. ISBN 0-596-00415-X.
- Thompson, K. (1968). "Programming Techniques: Regular expression search algorithm". *Communications of the ACM* 11 (6): 419—422. doi:10.1145/363347.363387.
- Wall, Larry (2002). "Apocalypse 5: Pattern Matching".

External links

Wikibooks has a book on the topic of: *Regular Expressions*

The Wikibook *R Programming* has a page on the topic of: *Text Processing*

- Regular Expressions (https://www.dmoz.org/Computers/Programming/Languages/Regular_Expressions) at DMOZ
- 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)

Categories: Automata (computation) | Formal languages | Pattern matching | Programming constructs | Regular expressions