Regular Expressions/Print version

Introduction

What is a regular expression?

A **regular expression** is a method of representing a string matching pattern. Regular expressions enable strings that match a particular pattern within textual data records to be located and modified and they are often used within utility programs and programming languages that manipulate textual data. Regular expressions are extremely powerful.

Example applications

Various software applications use regular expressions to locate, select or modify particular sections text. For example, a regular expression could be used to:

- replace the word "snake" with the word "serpent" throughout an entire piece of text
- locate pieces of text containing the words "fox" and "sheep" on the same line

Regular expression components

Regular expressions are made up of three types of components:

- anchors used to specify the position of the pattern in relation to a line of text.
- character sets used to match one or more characters in a single position.
- modifiers used to specify how many times a character set is repeated.

Syntax varies across application programs

The syntax of regular expressions varies across application programs. For example the shell uses a limited form of regular expression called shell regular expressions for filename substitution, whereas AWK uses a superset of extended regular expressions syntax.

Supporting Software

Regular expressions are supported by various software tools, including command line tools, plain text editors and programming languages. Most of these tools are available for various computing platforms, including Linux, Windows and Mac OS X. The tools use slightly different syntax styles. Let's look at some notable ones.

The tools:

- Command line tools
  - grep
  - egrep
  - sed
- Plain text editors
  - ed
  - vi
  - Emacs
- Programming languages
  - Awk
  - Java

- JavaScript
- .NET
- Perl
- PHP
- Ruby
- Tcl
- Python

A regular expression can be considered to be a little computer program that finds or isolates a subset of a larger set of text. In the same way that an ordinary computer program needs a computer to execute it, a regular expression needs a software application to interpret it — to give it meaning.

For example, a regular expression can be used to tell an editor to find the next occurrence of the word "Chapter" followed by several spaces and digits. Or you can use a regular expression to tell the UNIX grep command to show only those lines of a file that contain the word "Wiki" followed by either the word "Books" or the word-fragment "pedia". We will discuss the exact syntax of such regular expressions in the next chapter.

Syntaxes

There are several variants of regular expressions. These variants differ not only in their concrete syntax but also in their capabilities. Individual tools that support regular expressions also have their own peculiarities.

- ../Simple Regular Expressions/ - widely used for backwards compatibility, but deprecated on POSIX compliant systems.
- ../Basic Regular Expressions/ - used by some Unix shell tools
- ../Perl Compatible Regular Expressions/ - used by Perl and some application programs
- ../POSIX Basic Regular Expressions/ - provides extensions for consistency between utility programs. These extensions are not supported by some traditional implementations of Unix tools.
- ../POSIX Extended Regular Expressions/ - may be supported by some Unix utiities via the -E command line switch
- ../Non-POSIX Basic Regular Expressions/ - provides additional character classes not supported by POSIX
- ../Emacs Regular Expressions/ - used by the emacs editor
- ../Shell Regular Expressions/ - a limited form of regular expression used for pattern matching and filename substitution

Greedy expressions

Quantifiers such as * and + match as much as they can: they are greedy. For some uses, their greediness does not fit. For example, let as assume you want to find the first string enclosed in quotation marks, in the following text:

> These words include "cat", "mat", and "pat".

The pattern `".*"` matches the italicized part of the text below, that is, *"cat", "mat", and "pat"* instead of the desired *"cat"*:

> These words include *"cat", "mat", and "pat"*.

To fix this, some flavours of regular expressions provide non-greedy operators such as *?, +?, and }?. In PHP, adding a "U" at the end of the regexp makes the quantifier non-greedy, as in `/".*"/U`. In flavours that support neither of the two options, you can specify what is *not* to be matched, as in `("[^"]*")` to fix the discussed example. However, when dealing with bracketed expressions, `(\[\[^\]]*\]\])` fails to match on *A B C D E] F G*.

Comparison table

A comparison table or matrix that shows which features or flavors of regular expressions are available in which tool or programming language is available from regular-expressions.info [1].

Simple Regular Expressions

The **Simple Regular Expression** syntax is widely used on Unix based systems for the purposes of backwards compatibility. Most regular-expression-aware Unix utilities, such as *grep* and *sed*, use it by default while providing support for extended regular expressions with command line arguments (see below). This syntax is deprecated on POSIX compliant systems and should not be used by new utilities.

When simple regular expression syntax is being used, most characters, except metacharacters are treated as literal characters and match only themselves (for example, "a" matches "a", "(bc" matches "(bc", etc).

Operators

Operator	Effect
.	The dot operator matches any single character.
[ ]	boxes enable a single character to be matched against character lists or character ranges.
[^]	A compliment box enables a single character not within in a character list or character range to be matched.
^	A caret anchor matches the start of the line (or any line, when applied in multiline mode)
$	A dollar anchor matches the start of the line (or any line, when applied in multiline mode)
( )	parentheses are used to define a marked subexpression. The matched text section can be recalled at a later time.
\n	Where *n* is a digit from 1 to 9; matches what the *n*th marked subexpression matched. This irregular construct has not been adopted in the extended regular expression syntax.
*	A single character expression followed by "*" matches zero or more copies of the expression. For example, "ab*c" matches "ac", "abc", "abbbc" etc. "[xyz]*" matches "", "x", "y", "zx", "zyx", and so on. • \n*, where *n* is a digit from 1 to 9, matches zero or more iterations of what the *n*th marked subexpression matched. For example, "(a.)c\1*" matches "abcab" and "abcabab" but not "abcac". • An expression enclosed in "\(" and "\)" followed by "*" is deemed to be invalid. In some cases (e.g. /usr/bin/xpg4/grep of SunOS 5.8), it matches zero or more iterations of the string that the enclosed expression matches. In other cases (e.g. /usr/bin/grep of SunOS 5.8), it matches what the enclosed expression matches, followed by a literal "*".

Examples

Examples:

- "^[hc]at"
  - Matches *hat* and *cat* but only at the beginning of a line.
- "[hc]at$"
  - Matches *hat* and *cat* but only at the end of a line.

Use in Tools

Tools and languages that utilize this regular expression syntax include:

- Grep
- sed

Basic Regular Expressions

Basic Regular Expressions: Note that particular implementations of regular expressions interpret the backslash symbol differently in front of some of the metacharacters. For example, egrep and perl interpret *unbackslashed* parentheses and vertical bars as metacharacters, reserving the backslashed versions to mean the literal characters themselves. Old versions of grep did not support the pipe alternation operator.

Operators

Operator	Effect
.	The dot operator matches any single character.
[ ]	boxes enable a single character to be matched against a character lists or character range.
[^]	A compliment box enables a single character not within in a character list or character range to be matched.
*	An asterisk specifies zero or more characters to match.
^	The caret anchor matches the beginning of the line
$	The dollar anchor matches the end of the line

Examples:

Example	Match
".at"	any three-character string like *hat*, *cat* or *bat*
"[hc]at"	*hat* and *cat*
"[^b]at"	all the matched strings from the regex ".at" except *bat*
"^[hc]at"	*hat* and *cat* but only at the beginning of a line
"[hc]at$"	*hat* and *cat* but only at the end of a line

Since many ranges of characters depends on the chosen locale setting (e.g., in some settings letters are organized as abc..yzABC..YZ while in some others as aAbBcC..yYzZ).

The ../Posix Basic Regular Expressions/ syntax provided extensions for consistency between utility programs such as grep, sed and awk. These extensions are not supported by some traditional implementations of Unix tools.

Use in Tools

Tools and languages that utilize this regular expression syntax include: TBD

Perl Compatible Regular Expressions

Perl has a richer and more predictable syntax than even the ../POSIX Extended Regular Expressions/ syntax. An example of its predictability is that \ always quotes a non-alphanumeric character. An example of something that is possible to specify with Perl but not POSIX is whether part of the match wanted to be greedy or not. For instance in the pattern /a.*b/, the .* will match as much as it can, while in the pattern /a.*?b/, .*? will match as little. So given

the string "a bad dab", the first pattern will match the whole string, and the second will only match "a b".

For these reasons, many other utilities and applications have adopted syntaxes that look a lot like Perl's. For example, Java, Ruby, Python, PHP, exim, BBEdit, and even Microsoft's .NET Framework all use regular expression syntax similar to that used in perl. Not all "Perl-compatible" regular expression implementations are identical, and many implement only a subset of Perl's features.

Examples

Conventions used in the examples: 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 [2]' for more details.

```
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
```

In the table heading below, "M-c" stands for "Metacharacters".

M-c	Description	Example All the if statements return a TRUE value.

.	Normally matches any character except a newline. Within square brackets the dot is literal.	`if ("Hello World\n" =~ m/...../) {`
		`  print "Yep"; # 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.	`if ("Hello World\n" =~ 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.	`if ("Hello World\n" =~ m/l+/) {`
		`  print "One or more \"l\"'s in the string\n";`
		`}`

?	Matches the preceding pattern element zero or one times.	`if ("Hello World\n" =~ 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 regexp that comes before to match as few times as possible.	`if ("Hello World\n" =~ m/(l.+?o)/) {`
		`  print "Yep"; # The non-greedy match with 'l' followed`
		`  # by one or more characters is 'llo' rather than 'llo wo'.`
		`}`

*	Matches the preceding pattern element zero or more times.	`if ("Hello World\n" =~ m/el*o/) {`
		`  print "There is an 'e' followed by zero to many ";`
		`  print "'l' followed by 'o' (eo, elo, ello, elllo)\n";`
		`}`

{M,N}	Denotes the minimum M and the maximum N match count.	`if ("Hello World\n" =~ m/l{1,2}/) {`
		` print "There is a substring with at least 1 ";`
		` print "and at most 2 l's in the string\n";`
		`}`

[...]	Denotes a set of possible character matches.	`if ("Hello World\n" =~ m/[aeiou]+/) {`
		`  print "Yep"; # Contains one or more vowels`
		`}`

|	Separates alternate possibilities.	`if ("Hello World\n" =~ m/(Hello|Hi|Pogo)/) {`
		`  print "At least one of Hello, Hi, or Pogo is ";`
		`  print "contained in the string.\n";`
		`}`

\b	Matches a word boundary.	`if ("Hello World\n" =~ m/llo\b/) {`
		`  print "There is a word that ends with 'llo'\n";`
		`}`

\w	Matches an alphanumeric character, including "_".	`if ("Hello World\n" =~ m/\w/) {`
		`  print "There is at least one alphanumeric ";`
		`  print "character in the string (A-Z, a-z, 0-9, _)\n";`
		`}`

\W	Matches a non-alphanumeric character, excluding "_".	`if ("Hello World\n" =~ m/\W/) {`
		`  print "The space between Hello and ";`
		`  print "World is not alphanumeric\n";`
		`}`

\s	Matches a whitespace character (space, tab, newline, form feed)	`if ("Hello World\n" =~ m/\s.*\s/) {`
		`  print "There are TWO whitespace characters, which may";`
		`  print " be separated by other characters, in the string.";`
		`}`

\S	Matches anything BUT a whitespace.	`if ("Hello World\n" =~ m/\S.*\S/) {`
		`  print "Contains two non-whitespace characters " .`
		`  "separated by zero or more characters.";`
		`}`

\d	Matches a digit, same as [0-9].	`if ("99 bottles of beer on the wall." =~ m/(\d+)/) {`
		`  print "$1 is the first number in the string'\n";`
		`}`

\D	Matches a non-digit.	`if ("Hello World\n" =~ m/\D/) {`
		`  print "There is at least one character in the string";`
		`  print " that is not a digit.\n";`
		`}`

^	Matches the beginning of a line or string.	`if ("Hello World\n" =~ m/^He/) {`
		`  print "Starts with the characters 'He'\n";`
		`}`

$	Matches the end of a line or string.	`if ("Hello World\n" =~ m/rld$/) {`
		`  print "Is a line or string ";`
		`  print "that ends with 'rld'\n";`
		`}`

\A	Matches the beginning of a string (but not an internal line).	`if ("Hello\nWorld\n" =~ m/\AH/) {`
		`  print "Yep"; # The string starts with 'H'.`
		`}`

\Z	Matches the end of a string (but not an internal line).	`if ("Hello\nWorld\n"; =~ m/d\n\Z/) {`
		`  print "Yep"; # Ends with 'd\\n'\n";`
		`}`

[^...]	Matches every character except the ones inside brackets.	`if ("Hello World\n" =~ m/[^abc]/) {`
		`  print "Yep"; # Contains a character other than a, b, and c.`
		`}`

Use in Tools

Tools and languages that utilize Perl regular expression syntax include:

- Java
- leafnode
- Perl
- Python
- PHP

Links

- Perl regular expressions [2] at perl.org
- Perl Compatible Regular Expressions library [3] at pcre.org
- Perl Regular Expression Syntax [4] at boost.org
- W:Regular_expressions#Standard_Perl

POSIX Basic Regular Expressions

The **POSIX Basic Regular Expression** syntax provided extensions to achieve consistency between utility programs such as grep, sed and awk. These extensions are not supported by some traditional implementations of Unix tools.

History

Traditional Unix regular expression syntax followed common conventions that often differed from tool to tool. The POSIX Basic Regular Expressions syntax was developed by the IEEE, together with an extended variant called Extended Regular Expression syntax. These standards were designed mostly to provide backward compatibility with the traditional ../Simple Regular Expressions/ syntax, providing a common standard which has since been adopted as the default syntax of many Unix regular expression tools.

Syntax

In POSIX Basic Regular Expression syntax, most characters are treated as literals — they match only themselves (e.g., `a` matches "*a*"). The exceptions, listed below, are called *metacharacters* or *metasequences*.

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 `^`) 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*". As above, 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.

BRE: `\( \)` ERE: `( )`	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.
`\n`	Matches what the *n*th marked subexpression matched, where *n* is a digit from 1 to 9. This construct is theoretically **irregular** and was not adopted in the POSIX ERE syntax. 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*", "*abababab*", and so on.
BRE: `\{m,n\}` ERE: `{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.

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 "*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]*".

Character classes

The POSIX standard defines some classes or categories of characters as shown in the following table:

POSIX class	similar to	meaning
[:upper:]	[A-Z]	uppercase letters
[:lower:]	[a-z]	lowercase letters
[:alpha:]	[A-Za-z]	upper- and lowercase letters
[:alnum:]	[A-Za-z0-9]	digits, upper- and lowercase letters
[:digit:]	[0-9]	digits
[:xdigit:]	[0-9A-Fa-f]	hexadecimal digits
[:punct:]	[.,!?:...]	punctuation
[:blank:]	[\t]	space and TAB characters only
[:space:]	[\t\n\r\f\v]	blank (whitespace) characters
[:cntrl:]		control characters
[:graph:]	[^ \t\n\r\f\v]	printed characters
[:print:]	[^\t\n\r\f\v]	printed characters and space

Links:

- W:Regular_expression#Character_classes
- Character Classes that are Always Supported [5] at boost.org

Use in Tools

Tools and languages that utilize this regular expression syntax include:

- W:TextPad

Links

- POSIX Basic Regular Expressions [6] at regular-expressions.info
- POSIX Basic Regular Expression Syntax [7] at boost.org

POSIX Extended Regular Expressions

The more modern "extended" regular expressions can often be used with modern Unix utilities by including the command line flag "-E".

POSIX extended regular expressions are similar in syntax to the traditional Unix regular expressions, with some exceptions. The following metacharacters are added:

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 `^`) 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*". As above, 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.
BRE: `\( \)` ERE: `( )`	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.
`\n`	Matches what the *n*th marked subexpression matched, where *n* is a digit from 1 to 9. This construct is theoretically **irregular** and was not adopted in the POSIX ERE syntax. 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.
BRE: `\{m,n\}` ERE: `{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.

- **+** — Match the last "block" one or more times - "ba+" matches "ba", "baa", "baaa" and so on
- **?** — Match the last "block" zero or one times - "ba?" matches "b" or "ba"
- **|** — The choice (or set union) operator: match either the expression before or the expression after the operator - "abc|def" matches "abc" or "def".

Also, backslashes are removed: \{...\} becomes {...} and \(...\) becomes (...). Examples:

- "[hc]+at" matches with "hat", "cat", "hhat", "chat", "hcat", "ccchat" etc.

- "[hc]?at" matches "hat", "cat" and "at"
- "([cC]at)|([dD]og)" matches "cat", "Cat", "dog" and "Dog"

The characters (,),[,],.,*,?,+,^, and $ are special symbols and have to be escaped with a backslash symbol in order to be treated as literal characters. For example:

"a\.(\(|\))" matches with the string "a.)" or "a.("

Modern regular expression tools allow a quantifier to be specified as non-greedy, by putting a question mark after the quantifier: `(\[\[.*?\]\])`.

Use in Tools

Tools and languages that utilize this regular expression syntax include:

- AWK - uses a superset of the extended regular expression syntax

Links

- POSIX Basic Regular Expressions [6] at regular-expressions.info
- POSIX Extended Regular Expression Syntax [8] at boost.org

Non-POSIX Basic Regular Expressions

Non POSIX Basic Regular Expression Syntax: An additional non-POSIX class understood by some tools is [:word:], which is usually defined as [:alnum:] plus underscore. This form of regular expression is used to reflect the fact that in many programming languages these characters may be used in identifiers.

Operators

Operator	Effect
.	The dot operator matches any single character.
[ ]	boxes enable a single character to be matched against a character lists or character range.
[^]	A compliment box enables a single character not within in a character list or character range to be matched.
*	An asterisk specifies zero or more characters to match.
^	The caret anchor matches the beginning of the line
$	The dollar anchor matches the end of the line

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.

(For an ASCII chart color-coded to show the POSIX classes, see ASCII [9].)

Use in Tools

Tools and languages that utilize this regular expression syntax include:

- vim

Emacs Regular Expressions

Notes on regular expressions used in text editor **Emacs**:

- For backslash escaping (magic vs literal), Emacs uses a mixture of BRE and ERE. Like in ERE, Emacs supports unescaped +, ?. Like in BRE, Emacs supports escaped \(, \), \|, \{, \}.
- GNU extensions to regular expressions supported by Emacs include \w, \W, \b, \B, \<, \>, \` , \' (start and end of buffer)
- No "\s" like in PCRE; whitespace is matched by "\s-".
- No "\d" like in PCRE; use [0-9] or [[:digit:]]
- No lookahead and no lookbehind like in PCRE
- Emacs regexp can match characters by syntax using mode-specific syntax tables ("\sc", "\s-", "\s ") or by categories ("\cc", "\cg").

Use in Tools

Tools and languages that utilize this regular expression syntax include:

- Emacs

Links

- Regular Expressions [10] at emacswiki.org
- Perl and Emacs regular expressions compared [11] at lemoda.net

Shell Regular Expressions

The Unix shell recognises a limited form of regular expressions used with filename substitution:

Operators

Operator	Effect
?	The hook operator specifies any single character.
[ ]	boxes enable a single character to be matched against a character lists or character range.
[!]	A compliment box enables a single character not within in a character list or character range to be matched.
*	An asterisk specifies zero or more characters to match.

Some operators behave differently in the shell: The asterisk and hook operators do not not need to follow a previous character in the shell and they exhibit non traditional regular expression behaviour.

Unsupported Constructs: Within the shell, a compliment box is formed using the pling symbol. The shell does not support the use of a careted box for character list exclusion. In the shell, a caret symbol within a box will simply be treated as one of the characters within the character list for matching.

Use in Tools

Tools and languages that utilize this regular expression syntax include:

- Bourne compatible shells

Implementation

Implementations and running times

There are at least two different algorithms that decide if (and how) a given string matches a regular expression.

The oldest and fastest relies on a result in formal language theory that allows every nondeterministic Finite State Machine (NFA) to be transformed into a deterministic finite state machine (DFA). The algorithm performs or simulates this transformation and then runs the resulting DFA on the input string, one symbol at a time. The latter process takes time linear to the length of the input string. More precisely, an input string of size n can be tested against a regular expression of size m in time $O(n+2^m)$ or $O(nm)$, depending on the details of the implementation. This algorithm is often referred to as DFA. It is fast, but can be used only for matching and not for recalling grouped subexpressions. There is a variant that can recall grouped subexpressions, but its running time slows down to $O(n^2m)$ [*citation needed*].

The other algorithm is to match the pattern against the input string by backtracking. (This algorithm is sometimes called NFA, but this terminology is highly 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 exponential number of subcases. More complex implementations identify and speed up various common cases where they would otherwise run slowly.

Even though backtracking implementations only give an exponential guarantee in the worst case, they allow much greater flexibility and provide more expressive power. For instance any implementation that allows the use of backreferences, or implements the various improvements that Perl introduced, must use a backtracking implementation.

Some implementations try to provide the best of both algorithms by first running a fast DFA match to see if the string matches the regular expression at all, and only in that case perform a potentially slower backtracking match.

Examples

1. REDIRECT ../Perl Compatible Regular Expressions/

Glossary

This is a glossary of the book.

\A

: In some flavors, the beginning of a string but not of a line in the string

\b

: In some flavors, a word boundary

\B

: In some flavors, a complement to \b

BRE

: Basic regular expressions

\d

: In some flavors, a digit

\D

: In some flavors, a complement to \d

Emacs

: A scriptable text editor with support for regular expressions

ERE

: Extended regular expressions

GNU

: A project to create a free-as-in-freedom operating system, which provides extensions to regular expressions used in tools such as Grep or Sed

Greedy

: Of an operator, matching as much as it can

Grep

: A command-line tool for finding lines in a text files that match a regular expression

Java

: A byte-compiled programming language with support for regular expressions in its standard library since version 1.4

JavaScript

: A scripting languages for the web supported by web browsers, with built-in support for regular expressions

Metacharacter

: A character or sequence of characters with a special meaning, such as "." or "\+".

PCRE

: Perl compatible regular expressions

Perl

: An interpreted scripting language noted for its regular expressions

PHP

: An interpreted scripting language with support for regular expressions

Regex

: A regular expression

Regular expression

: A string containing special characters indicating patterns, intended to match literal strings

\s

: In some flavors, a whitespace character: space, tab, newline, form feed

\s-

: In Emacs, a whitespace character

\S

: In some flavors, a complement to \s

Sed

A non-interactive editor or command-line tool noted for its "s" command substituting strings that match a regular expression with other strings

\u13F

In some flavors, the character with the hexadecimal Unicode value of 13F.

Vim

A scriptable text editor with support for regular expressions

\w

In some flavors, an alphanumeric character, including "_"

\W

In some flavors, a complement to \w

\xF7

In some flavors, the character with the hexadecimal ASCII value of F7.

\x{13F}

In some flavors, the character with the hexadecimal Unicode value of 13F.

\Z

In some flavors, the end of a string but not of a line in the string

\<

In some flavors, an empty string before the beginning of a word

\>

In some flavors, an empty string after the end of a word

^

The beginning of a line

$

The end of a line

.

Any single character, but possibly not a newline

[

The opening of a character class

]

The closing of a character class

(

In some flavors, the opening of a group

)

In some flavors, the closing of a group

\(

In some flavors, the opening of a group

\)

In some flavors, the closing of a group

{

In some flavors, the opening of a match-count iterator

}
: In some flavors, the closing of a match-count iterator

\{
: In some flavors, the opening of a match-count iterator

\}
: In some flavors, the closing of a match-count iterator

|
: In some flavors, a marking of an alternative

\|
: In some flavors, a marking of an alternative

\1
: In some flavors, a backreference to the 1st group

\2
: In some flavors, a backreference to the 2nd group

*
: Any number of the previous

+
: In some flavors, one or more of the previous

\+
: In some flavors, one or more of the previous

?
: In some flavors, one or none of the previous

\?
: In some flavors, one or none of the previous

*?
: In some flavors, a non-greedy version of *

+?
: In some flavors, a non-greedy version of +

}?
: In some flavors, a non-greedy version of }

Links

External Links

- regular-expressions.info [12] - a regex tutorial site
- Text and Data Manipulation with Regular Expressions in .NET Development [13] — tutorial, reference and *Ready to use Regular Expression patterns for VB.NET, C#.NET and ASP.NET development*
- Emacs regular expressions [14] at cs.utah.edu
- W:Regular expression
- W:Comparison of regular expression engines
- W:List of regular expression software

References

[1] http://www.regular-expressions.info/refflavors.html
[2] http://perldoc.perl.org/perlre.html
[3] http://www.pcre.org/
[4] http://www.boost.org/doc/libs/1_34_1/libs/regex/doc/syntax_perl.html
[5] http://www.boost.org/doc/libs/1_44_0/libs/regex/doc/html/boost_regex/syntax/character_classes/std_char_clases.html
[6] http://www.regular-expressions.info/posix.html
[7] http://www.boost.org/doc/libs/1_44_0/libs/regex/doc/html/boost_regex/syntax/basic_syntax.html
[8] http://www.boost.org/doc/libs/1_44_0/libs/regex/doc/html/boost_regex/syntax/basic_extended.html
[9] http://billposer.org/Linguistics/Computation/ascii.html
[10] http://www.emacswiki.org/emacs/RegularExpression
[11] http://www.lemoda.net/emacs/perl-emacs-regex/index.html
[12] http://www.regular-expressions.info/
[13] http://www.tipsntracks.com/182/text-and-data-manipulation-with-regular-expressions-in-net-development.html
[14] http://www.cs.utah.edu/dept/old/texinfo/emacs18/emacs_17.html

Article Sources and Contributors

Regular Expressions/Print version *Source*: http://en.wikibooks.org/w/index.php?oldid=2536378 *Contributors*: Dan Polansky, LoStrangolatore

License