Regular Expressions

On this page…

Overview

A regular expression is a string of characters that defines a certain pattern. You normally use a regular expression to search text for a group of words that matches the pattern. for example, while parsing program input or while processing a block of text.

The string `'Joh?n\w*'` is an example of a regular expression. It defines a pattern that starts with the letters `Jo`, is optionally followed by the letter `h` (indicated by `'h?'`), is then followed by the letter `n`, and ends with any number of *word characters*[1] (indicated by `'\w*'`). This pattern matches any of the following:

```
Jon, John, Jonathan, Johnny
```

Regular expressions provide a unique way to search a volume of text for a particular subset of characters within that text. Instead of looking for an exact character match as you would do with a function like `strfind`, regular expressions give you the ability to look for a particular *pattern* of characters.

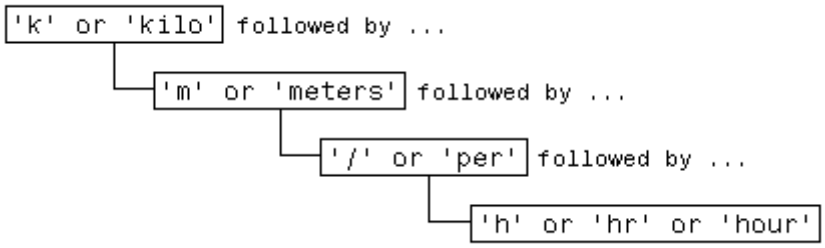
For example, several ways of expressing a metric rate of speed are:

```
km/h
km/hr
km/hour
kilometers/hour
kilometers per hour
```

You could locate any of the above terms in your text by issuing five separate search commands:

```
strfind(text, 'km/h');
strfind(text, 'km/hour');
     - etc. -
```

To be more efficient, however, you can build a single phrase that applies to all of these search strings:



Translate this phrase it into a regular expression (to be explained later in this section) and you have:

```
pattern = 'k(ilo)?m(eters)?(/|\sper\s)h(r|our)?';
```

Now locate one or more of the strings using just a single command:

```
text = ['The high-speed train traveled at 250 ', ...
   'kilometers per hour alongside the automobile ', ...
   'travelling at 120 km/h.'];

regexp(text, pattern, 'match')
ans =
    'kilometers per hour'    'km/h'
```

Back to Top

Calling Regular Expression Functions from MATLAB

This section covers the following topics:

Note The examples in this and some of the later sections of this documentation use expressions that can be difficult to decipher for anyone not previously exposed to them. The purpose of these initial examples is to introduce the basic use of regular expressions in MATLAB. Learning how to translate the expressions begins in the Metacharacters and Operators section.

MATLAB Regular Expression Functions

There are four MATLAB functions that support searching and replacing characters using regular expressions. The first three are similar in the input values they accept and the output values they return. For details, click the links in the table to see the corresponding function reference pages in the MATLAB Help.

Function	Description
`regexp`	Match regular expression.
`regexpi`	Match regular expression, ignoring case.
`regexprep`	Replace string using regular expression.
`regexptranslate`	Translate string into regular expression.

When calling any of the first three functions, pass the string to be parsed and the regular expression in the first two input arguments. When calling `regexprep`, pass an additional input that is an expression that specifies a pattern for the replacement string.

Returning the Desired Information

The `regexp` and `regexpi` functions return from 1 to 7 output values. providing the following information:

- The content or array indices of all matching strings
- The content of all nonmatching strings
- The content, names, or array indices of all tokens that were found

Unless you specify otherwise, MATLAB returns as many output values as you have output variables for. These are returned in the order shown by theReturn Value Summary table.

The following call to `regexp` returns all 7 outputs:

```
[matchStart, matchEnd, tokenIndices, matchStrings, ...
   tokenStrings, tokenName, splitStrings] = regexp(str, expr);
```

To specify fewer values to return, include an identifying keyword in the input argument list when you call `regexp` or `regexpi`. For example, the following statement uses two of these keywords, `match` and `start`:

```
[matchStrings, matchStart] = regexp(str, expr, 'match', 'start')
```

When you execute this statement, MATLAB assigns a cell array of all strings that match the pattern to variable `matchStrings`, and assigns an array of doubles containing the starting index of each match to variable `matchStart`.

For information on these output values and selecting which outputs to return, see the `regexp` function reference page.

Modifying Parameters of the Search

You can fine-tune your regular expression parsing using the optional `mode` inputs: Case Sensitivity, Dot Matching, Anchor Type, and Spacing. These modes tell MATLAB whether or not to:

- Consider letter case when matching an expression to a string (Case Sensitivity mode).
- Include the newline (`\n`) character when matching the dot (`.`) metacharacter in a regular expression (Dot Matching mode).
- Consider the `^` and `$` metacharacters to represent the beginning and end of a string or the beginning and end of a line (Anchor Type mode).
- Ignore space characters and comments in the expression or to interpret them literally (Spacing mode).

Applying Modes. You can apply any of these modes in either of two ways:

- Apply the mode to *all* of a regular expression by passing the mode specifier in the argument list of the call. See Example 1, below.
- Apply the mode to *specific parts* of your expression by specifying the mode symbolically within the regular expression itself. See Example 2, below.

Example 1 — Applying Case Sensitivity Mode to the Entire String.

Create two slightly different strings, `s1` and `s2`. Then write an expression `expr` that you can use to match both of these strings, but only when ignoring case. (The expression operators `.+` match any consecutive series of any character between the `MAT` or `mat` phrases.)

```
s1 = 'Save your MATLAB data to a .mat file in C:\work\matlab';
s2 = 'Save your MATLAB data to a .MAT file in C:\work\matlab';
expr = '.+MAT.+mat.+mat.+';
```

Run `regexp` on both strings at the same time in `ignorecase` mode and examine the output in cell array `c`:

```
c = regexp({s1, s2}, expr, 'match', 'ignorecase');
c{:}
ans =
    'Save your MATLAB data to a .mat file in C:\work\matlab'
ans =
    'Save your MATLAB data to a .MAT file in C:\work\matlab'
```

Because of the `ignorecase` mode, there is a match for both strings. When you use `matchcase` mode instead, only the exact case match is accepted:

```
c = regexp({s1, s2}, expr, 'match', 'matchcase');
c{:}
ans =
    'Save your MATLAB data to a .mat file in C:\work\matlab'
ans =
     {}
```

Example 2 — Applying Case Sensitivity Mode Selectively. This example uses symbolic mode designators within the expression itself. The `(?i)` symbol tells `regexp` to ignore case for that part of the expression that immediately follows it. Similarly, the `(?-i)` symbol requires case to match for the part of the expression following it.

Here are three strings that vary slightly in case. Following that is the expression `expr` that employs the two states of the Case Sensitivity mode. Note that each of the `(?-i)` or `(?i)` symbols used in this expression applies only to the letters `MAT` or `mat` that immediately follow it:

```
s1 = 'Save your MATLAB data to a .mat file in C:\work\matlab';
s2 = 'Save your MATLAB data to a .MAT file in C:\work\MATLAB';
s3 = 'Save your MATLAB data to a .MAT file in C:\work\matlab';
expr = '.*(?-i)MAT.*(?i)mat.*(?-i)mat';
```

Run `regexp` on the three strings. According to the expression `expr`, the first and third instances of the letters '`mat`' must be in upper and lower case, respectively. Case is ignored for the second instance. Only strings `s1` and `s3` satisfy this condition:

```
c = regexp({s1,s2,s3}, expr, 'match');
c{:}
ans =
    'Save your MATLAB data to a .mat file in C:\work\mat'
ans =
     {}
ans =
    'Save your MATLAB data to a .MAT file in C:\work\mat'
```

Back to Top

Parsing Strings with Regular Expressions

MATLAB parses a string from left to right, "consuming" the string as it goes. If matching characters are found, `regexp` records the location and resumes parsing the string, starting just after the end of the most recent match. There is no overlapping of characters in this process. See Examples 2a and 2b under "Using the Lookahead Operator" if you need to match overlapping character groups.

There are three steps involved in using regular expressions to search text for a particular string:

1. Identify unique patterns in the string

   This entails breaking up the string you want to search for into groups of like character types. These character types could be a series of lowercase letters, a dollar sign followed by three numbers and then a decimal point, etc.
2. Express each pattern as a regular expression

   Use the *metacharacters* and operators described in this documentation to express each segment of your search string as a regular expression. Then combine these expression segments into the single expression to use in the search.
3. Call the appropriate search function

   Pass the string you want to parse to one of the search functions, such as `regexp` or `regexpi`, or to the string replacement function, `regexprep`.

The example shown in this section searches a record containing contact information belonging to a group of five friends. This information includes each person's name, telephone number, place of residence, and e-mail address. The goal is to extract specific information from one or more of the strings.

```
contacts = { ...
'Harry  287-625-7315  Columbus, OH  hparker@hmail.com'; ...
'Janice  529-882-1759  Fresno, CA  jan_stephens@horizon.net'; ...
'Mike  793-136-0975  Richmond, VA  sue_and_mike@hmail.net'; ...
'Nadine  648-427-9947  Tampa, FL  nadine_berry@horizon.net'; ...
'Jason  697-336-7728  Montrose, CO  jason_blake@mymail.com'};
```

The first part of the example builds a regular expression that represents the format of a standard e-mail address. Using that expression, the example then searches the information for the e-mail address of one of the group of friends. Contact information for Janice is in row 2 of the `contacts` cell array:

```
contacts{2}
ans =
   Janice  793-882-1759  Fresno, CA  jan_stephens@horizon.net
```

Step 1 — Identify Unique Patterns in the String

A typical e-mail address is made up of standard components: the user's account name, followed by an @ sign, the name of the user's internet service provider (ISP), a dot (period), and the domain to which the ISP belongs. The table below lists these components in the left column, and generalizes the format of each component in the right column.

Unique patterns of an email address	General description of each pattern
Start with the account name `jan_stephens ...`	One or more lowercase letters and underscores
Add '@' `jan_stephens@ ...`	`@` sign

Add the ISP `jan_stephens@horizon ...`	One or more lowercase letters, no underscores
Add a dot (period) `jan_stephens@horizon. ...`	Dot (period) character
Finish with the domain `jan_stephens@horizon.net`	`com` or `net`

Step 2 — Express Each Pattern as a Regular Expression

In this step, you translate the general formats derived in Step 1 into segments of a regular expression. You then add these segments together to form the entire expression.

The table below shows the generalized format descriptions of each character pattern in the left-most column. (This was carried forward from the right column of the table in Step 1.) The second column links to tables in this documentation that show the appropriate expressions to use in translating this description into a regular expression. The third column shows the operators or metacharacters chosen from those tables to represent the character pattern.

Description of each segment	Tables referenced	Related metacharacters
One or more lowercase letters and underscores	See Character Types, Quantifiers.	`[a-z_]+`
`@` sign	See Character Representation.	`@`
One or more lowercase letters, no underscores	See Character Types, Quantifiers.	`[a-z]+`
Dot (period) character	See Character Representation.	`\.`
`com` or `net`	See Grouping Operators.	`(com|net)`

Assembling these metacharacters into one string gives you the complete expression:

```
email = '[a-z_]+@[a-z]+\.(com|net)';
```

Step 3 — Call the Appropriate Search Function

In this step, you use the regular expression derived in Step 2 to match an e-mail address for one of the friends in the group. Use the `regexp` function to perform the search.

Here is the list of contact information shown earlier in this section. Each person's record occupies a row of the `contacts` cell array:

```
contacts = { ...
'Harry  287-625-7315  Columbus, OH  hparker@hmail.com'; ...
'Janice  529-882-1759  Fresno, CA  jan_stephens@horizon.net'; ...
'Mike  793-136-0975  Richmond, VA  sue_and_mike@hmail.net'; ...
```

```
'Nadine  648-427-9947  Tampa, FL  nadine_berry@horizon.net'; ...
'Jason  697-336-7728  Montrose, CO  jason_blake@mymail.com'};
```

This is the regular expression that represents an e-mail address, as derived in Step 2:

```
email = '[a-z_]+@[a-z]+\.(com|net)';
```

Call the `regexp` function, passing row 2 of the `contacts` cell array and the `email` regular expression. This returns the e-mail address for Janice.

```
regexp(contacts{2}, email, 'match')
ans =
    'jan_stephens@horizon.net'
```

> **Note** The last input passed to `regexp` in this command is the keyword `'match'` This keyword causes `regexp` to return the output as a string instead of as indices into the cell array.

Make the same call, but this time for the fifth person in the list:

```
regexp(contacts{5}, email, 'match')
ans =
     'jason_blake@mymail.com'
```

You can also search for the e-mail address of everyone in the list by using the entire cell array for the input string argument:

```
regexp(contacts, email, 'match');
```

Back to Top

Other Benefits of Using Regular Expressions

In addition to parsing single strings, you can also use the MATLAB regular expression functions for any of the following tasks:

- Parsing or Replacing with Multiple Expressions and Strings
- Replacing Parts of a String
- Matching with Tokens Taken from the String
- Matching and Replacing Strings Dynamically

Parsing or Replacing with Multiple Expressions and Strings

The MATLAB regular expression functions also work on multiple strings contained in a cell array. You can use multiple strings as the strings to be parsed, as regular expressions to match against the parse string(s), as replacement strings, or most combinations of these.

Replacing Parts of a String

String replacement with regular expressions requires the `regexprep` function. This function accepts two regular expressions in its input argument list. Each expression specifies a

character pattern to match in the string to be parsed. The function then replaces occurrences of the first pattern with occurrences of the second.

Matching with Tokens Taken from the String

A token is one or more characters selected from within the string being parsed that you can use to match other characters in the same string. The characters representing a token are not constants; they depend upon the contents of the parse string that match a part of the expression. You define a token by enclosing part of a regular expression in parentheses. You search for that token using the metacharacters `\1`, `\2`, etc. You can also use tokens in specifying a replacement string for the `regexprep` function. In this case, you refer to specific tokens using the metacharacters `$1`, `$2`, etc.

Matching and Replacing Strings Dynamically

With dynamic expressions, you can:

- Execute a MATLAB command within your expression parsing command.
- Execute a MATLAB command, and include the returned string in the match expression.
- Parse a regular expression and include the resulting string in the match expression.

Back to Top

Metacharacters and Operators

Much of the remainder of this section on regular expressions documents the various metacharacters and operators that you need to compose your expressions.

Category	Metacharacters and Operators
Character Type Operators	One of a certain group of characters (e.g., a character in a predefined set or range, a whitespace character, an alphabetic, numeric, or underscore character, or a character that is not in one of these groups.
Character Representation	Metacharacters that represent a special character (e.g., backslash, new line, tab, hexadecimal values, any untranslated literal character, etc.
Grouping Operators	A grouping of letters or metacharacters to apply a regular expression operator to.
Nonmatching Operators	Text included in an expression for the purpose of adding a comment statement, but not to be used as a pattern to find a match for.
Positional Operators	Location in the string where the characters or pattern must be positioned for there to be a match (e.g., start or end of the string, start or end of a word, an entire word).

Lookaround Operators	Characters or patterns that immediately precede or follow the intended match, but are not considered to be part of the match itself.
Quantifiers	Various ways of expressing the number of times a character or pattern is to occur for there to be a match (e.g., exact number, minimum, maximum, zero or one, zero or more, one or more, etc.)
Tokens	Characters or patterns selected from the string being parsed that you can use to match other characters in the string.
Named Capture	Operators used in assigning names to matched tokens, thus making your code more maintainable and the output easier to interpret.
Conditional Expressions	Operators that express conditions under which a certain match is considered to be is acceptable.
Dynamic Regular Expressions	Operators that include a subexpression or command that MATLAB parses or executes. MATLAB uses the result of that operation in parsing the overall expression.
String Replacement	Operators used with the `regexprep` function to specify the content of the replacement text.

Back to Top

Character Type Operators

Tables and examples in this and subsequent sections show the operators and syntax supported by the MATLAB `regexp`, `regexpi`, and `regexprep` functions. Expressions shown in the left column have special meaning and match one or more characters according to the usage described in the right column. Any character not having a special meaning, for example, any alphabetic character, matches that same character literally. To force one of the regular expression functions to interpret a sequence of characters literally (rather than as an operator) use the `regexptranslate` function.

Character types represent either a specific set of characters (e.g., uppercase) or a certain type of character (e.g., nonwhitespace).

Operator	Usage
.	Any single character, including white space
$[c_1c_2c_3]$	Any character contained within the brackets: c_1 or c_2 or c_3
$[^c_1c_2c_3]$	Any character not contained within the brackets: anything but c_1 or c_2 or c_3

[c_1-c_2]	Any character in the range of c_1 through c_2
\s	Any white-space character; equivalent to [\f\n\r\t\v]
\S	Any nonwhitespace character; equivalent to [^ \f\n\r\t\v]
\w	Any alphabetic, numeric, or underscore character. For English character sets, this is equivalent to [a-zA-Z_0-9].
\W	Any character that is not alphabetic, numeric, or underscore. For English character sets, this is equivalent to [^a-zA-Z_0-9].
\d	Any numeric digit; equivalent to [0-9]
\D	Any nondigit character; equivalent to [^0-9]

The following examples demonstrate how to use the character classes listed above. See the regexp reference page for help with syntax. Most of these examples use the following string:

```
str = 'The rain in Spain falls mainly on the plain.';
```

Any Character — .

The . operator matches any single character, including whitespace.

Example 1 — Matching Any Character. Use the dot (.) operator to locate sequences of five consecutive characters that end with 'ain'. The regular expression used in this example is

```
expr = '..ain';
```

Find each occurrence of the expression expr within the input string str. Return a vector of the indices at which any matches begin:

```
str = 'The rain in Spain falls mainly on the plain.';

startIndex = regexp(str, expr)
startIndex =
     4    13    24    39
```

Here is the input string with the returned startIndex values shown below it. Note that the dot operator not only matches the letters in the string, but whitespace characters as well:

```
The rain in Spain falls mainly on the plain.
   |         |           |               |
   4        13          24              39
```

If you would prefer to have MATLAB return the text of the matching substrings, use the 'match' qualifier in the command:

```
matchStr = regexp(str, expr, 'match')
```

```
matchStr =
    ' rain'    'Spain'    ' main'    'plain'
```

Example 2 — Returning Strings Rather than Indices. Here is the same example, this time specifying the command qualifier `'match'`. In this case, `regexp` returns the *text* of the matching strings rather than the starting index:

```
regexp(str, '..ain', 'match')
ans =
    ' rain'    'Spain'    ' main'    'plain'
```

Selected Characters — [c1c2c3]

Use $[c_1c_2c_3]$ in an expression to match selected characters `r`, `p`, or `m` followed by `'ain'`. Specify two qualifiers this time, `'match'` and `'start'`, along with an output argument for each, `mat` and `idx`. This returns the matching strings and the starting indices of those strings:

```
[mat idx] = regexp(str, '[rpm]ain', 'match', 'start')
mat =
    'rain'    'pain'    'main'
idx =
     5    14    25
```

Range of Characters — [c1 - c2]

Use $[c_1\text{-}c_2]$ in an expression to find words that begin with a letter in the range of `A` through `Z`:

```
[mat idx] = regexp(str, '[A-Z]\w*', 'match', 'start')
mat =
    'The'    'Spain'
idx =
     1    13
```

Word and White-Space Characters — \w, \s

Use `\w` and `\s` in an expression to find words that end with the letter `n` followed by a white-space character. Add a new qualifier, `'end'`, to return the `str` index that marks the end of each match:

```
[mat ix1 ix2] = regexp(str, '\w*n\s', 'match', 'start', 'end')
mat =
    'rain '    'in '    'Spain '    'on '
ix1 =
     5    10    13    32
ix2 =
     9    12    18    34
```

Numeric Digits — \d

Use `\d` to find numeric digits in the following string:

```
numstr = 'Easy as 1, 2, 3';

[mat idx] = regexp(numstr, '\d', 'match', 'start')
mat =
    '1'    '2'    '3'
idx =
     9    12    15
```

Back to Top

Character Representation

The following character combinations represent specific character and numeric values.

Operator	Usage
`\a`	Alarm (beep)
`\\`	Backslash
`\$`	Dollar sign
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\oN` or `\o{N}`	Character of octal value `N`
`\xN` or `\x{N}`	Character of hexadecimal value `N`
`\char`	If a character has special meaning in a regular expression, precede it with backslash (`\`) to match it literally.

Octal and Hexadecimal — \o, \x

Use `\x` and `\o` in an expression to find a comma (hex `2C`) followed by a space (octal `40`) followed by the character `2`:

```
numstr = 'Easy as 1, 2, 3';

[mat idx] = regexp(numstr, '\x2C\o{40}2', 'match', 'start')
mat =
```

```
    ', 2'
idx =
    10
```

Back to Top

Grouping Operators

When you need to use one of the regular expression operators on a number of consecutive elements in an expression, group these elements together with one of the grouping operators and apply the operation to the entire group. For example, this command matches a capital letter followed by a numeral and then an optional space character. These elements have to occur at least two times in succession for there to be a match. To apply the `{2,}` multiplier to all three consecutive characters, you can first make a group of the characters and then apply the `(?:)` quantifier to this group:

```
regexp('B5 A2 6F 63 R6 P4 B2 BC', '(?:[A-Z]\d\s?){2,}', 'match')
ans =
    'B5 A2 '    'R6 P4 B2 '
```

There are three types of explicit grouping operators that you can use when you need to apply an operation to more than just one element in an expression. Also in the grouping category is the alternative match (logical OR) operator, `|`. This creates two or more groups of elements in the expression and applies an operation to one of the groups.

Operator	Usage
`(expr)`	Group regular expressions and capture tokens.
`(?:expr)`	Group regular expressions, but do not capture tokens.
`(?>expr)`	Group atomically.
$expr_1$`|`$expr_2$	Match expression $expr_1$ or expression $expr_2$.

Grouping and Capture — (expr)

When you enclose an expression in parentheses, MATLAB not only treats all of the enclosed elements as a group, but also captures a token from these elements whenever a match with the input string is found. For an example of how to use this, see Using Tokens — Example 1.

Grouping Only — (?:expr)

Use `(?:expr)` to group a non-vowel (consonant, numeric, whitespace, punctuation, etc.) followed by a vowel in the palindrome `pstr`. Specify at least two consecutive occurrences (`{2,}`) of this group. Return the starting and ending indices of the matched substrings:

```
pstr = 'Marge lets Norah see Sharon''s telegram';
expr = '(?:[^aeiou][aeiou]){2,}';
```

```
[mat ix1 ix2] = regexp(pstr, expr, 'match', 'start', 'end')
mat =
    'Nora'    'haro'    'tele'
ix1 =
    12    23    31
ix2 =
    15    26    34
```

Remove the grouping, and the `{2,}` now applies only to `[aeiou]`. The command is entirely different now as it looks for a non-vowel followed by at least two consecutive vowels:

```
expr = '[^aeiou][aeiou]{2,}';

[mat ix1 ix2] = regexp(pstr, expr, 'match', 'start', 'end')
mat =
    'see'
ix1 =
    18
ix2 =
    20
```

Alternative Match — expr1|expr2

Use $p_1|p_2$ to pick out words in the string that start with `let` or `tel`:

```
regexpi(pstr, '(let|tel)\w+', 'match')
ans =
    'lets'    'telegram'
```

> **Note** The expressions `A|B` and `B|A` may return different answers. If there is a match with the first part of the expression (before the | symbol), then the second part (that follows the | symbol) is not considered.
>
> See the following example. Both calls to `regexp` parse the same string, and, except for the order of the OR conditions, the same expression. But the first call returns two values and the second returns just one:

```
string = 'one two';    expr1 = '(\w+\s\w+)';    expr2 = '(\w+)';

regexp(string, [expr1 '|' expr2], 'match')
ans =
    'one two'

regexp(string, [expr2 '|' expr1], 'match')
ans =
    'one'    'two'
```

Back to Top

Nonmatching Operators

The `comment` operator enables you to insert comments into your code to make it more maintainable. The text of the comment is ignored by MATLAB when matching against the input string.

Operator	Usage
`(?#comment)`	Insert a comment into the expression. Comments are ignored in matching.

Including Comments — (?#expr)

Use `(?#expr)` to add a comment to this expression that matches capitalized words in `pstr`. Comments are ignored in the process of finding a match:

```
regexp(pstr, '(?# Match words in caps)[A-Z]\w+', 'match')
ans =
    'Marge'    'Norah'    'Sharon'
```

Back to Top

Positional Operators

Positional operators in an expression match parts of the input string not by content, but by where they occur in the string (e.g., the first N characters in the string).

Operator	Usage
`^expr`	Match `expr` if it occurs at the beginning of the input string.
`expr$`	Match `expr` if it occurs at the end of the input string.
`\<expr`	Match `expr` when it occurs at the beginning of a word.
`expr\>`	Match `expr` when it occurs at the end of a word.
`\<expr\>`	Match `expr` when it represents the entire word.

Start and End of String Match — ^expr, expr$

Use `^expr` to match words starting with the letter `m` or `M` only when it begins the string, and `expr$` to match words ending with `m` or `M` only when it ends the string:

```
regexpi(pstr, '^m\w*|\w*m$', 'match')
```

```
ans =
    'Marge'    'telegram'
```

Start and End of Word Match — \<expr, expr\>

Use `\<expr` to match any words starting with `n` or `N`, or ending with `e` or `E`:

```
regexpi(pstr, '\<n\w*|\w*e\>', 'match')
ans =
    'Marge'    'Norah'    'see'
```

Exact Word Match — \<expr\>

Use `\<expr\>` to match a word starting with an n or N and ending with an `h` or `H`:

```
regexpi(pstr, '\<n\w*h\>', 'match')
ans =
    'Norah'
```

Back to Top

Lookaround Operators

Lookaround operators tell MATLAB to look either ahead or behind the current location in the string for a specified expression. If the expression is found, MATLAB attempts to match a given pattern.

This table shows the four lookaround expressions: lookahead, negative lookahead, lookbehind, and negative lookbehind.

Operator	Usage
`(?=expr)`	Look ahead from current position and test if `expr` is found.
`(?!expr)`	Look ahead from current position and test if `expr` is not found
`(?<=expr)`	Look behind from current position and test if `expr` is found.
`(?<!expr)`	Look behind from current position and test if `expr` is not found.

Lookaround operators do not change the current parsing location in the input string. They are more of a condition that must be satisfied for a match to occur.

For example, the following command uses an expression that matches alphabetic, numeric, or underscore characters (`\w*`) that meet the condition that they *look ahead to* (i.e., are immediately followed by) the letters `vision`. The resulting match includes only that part of

the string that matches the `\w*` operator; it does not include those characters that match the lookahead expression `(?=vision)`:

```
[s e] = regexp('telegraph television telephone', ...
               '\w*(?=vision)', 'start', 'end')
s =
    11
e =
    14
```

If you repeat this command and match one character beyond the lookahead expression, you can see that parsing of the input string resumes at the letter `v`, thus demonstrating that matching the lookahead operator has not consumed any characters in the string:

```
regexp('telegraph television telephone', ...
       '\w*(?=vision).', 'match')
ans =
    'telev'
```

> **Note** You can also use lookaround operators to perform a logical AND of two elements. See Using Lookaround as a Logical Operator.

Using the Lookahead Operator — expr(?=test)

Example 1 — Lookahead. Look ahead to a file name (`fileread.m`), and return only the name of the folder in which it resides, not the file name itself. Note that the lookahead part of the expression serves only as a condition for the match; it is not part of the match itself:

```
str = which('fileread')
str =
   C:\Program Files\MATLAB\toolbox\matlab\iofun\fileread.m

% Look ahead to a backslash (\\), followed by a file name (\w+)
% with an .m or .p extension (\.[mp]). Capture the letters
% that precede this sequence.
regexp(str, '\w+(?=\\\w+\.[mp])', 'match')
ans =
    'iofun'
```

Example 2a — Matching Sequential Character Groups. MATLAB parses a string from left to right, "consuming" the string as it goes. If matching characters are found, `regexp` records the location and resumes parsing the string from the location of the most recent match. There is no overlapping of characters in this process.

Find all sequences of 6 nonwhitespace characters in the input string shown below. Following the MATLAB default behavior, do not allow for overlap. That is, begin looking for your next match starting just after the *end* of the current match:

```
string = 'Locate several 6-char. phrases';
regexpi(string, '\S{6}')
```

```
ans =
     1     8    16    24
```

This statement finds the phrases:

```
Locate   severa   6-char   phrase
```

Example 2b — Using Lookahead to Match Overlapping Character Groups. If you need to find *every* sequence of characters that match a pattern, including sequences that overlap another, capture only the first character and look ahead for the remainder of the pattern. In other words, begin looking for your next match starting after the *next character* of the current match:

```
string = 'Locate several 6-char. phrases';
regexpi(string, '\S(?=\S{5})')
ans =
     1     8     9    16    17    24    25
```

This statement finds the phrases:

```
Locate   severa   everal   6-char   -char.   phrase   hrases
```

Using the Negative Lookahead Operator — expr(?!test)

Example — Negative Lookbehind and Lookahead. Generate a series of sequential numbers:

```
n = num2str(5:15)
n =
   5   6   7   8   9  10  11  12  13  14  15
```

Use both the negative lookbehind and negative lookahead operators together to precede only the single-digit numbers with zero:

```
regexprep(n, '(?<!\d)(\d)(?!\d)', '0$1')
ans =
   05   06   07   08   09  10  11  12  13  14  15
```

Using the Lookbehind Operator — (?<=test)expr

Example 1 — Positive and Negative Lookbehind Operators. Using the lookbehind operator, find the letter `r` that is preceded by the letter `u`:

```
str = 'Neural Network Toolbox';

startIndex = regexp(str, '(?<=u)r', 'start')
startIndex =
     4
```

Using the negative lookbehind operator, find the letter `r` that is *not* preceded by the letter `u`:

```
startIndex = regexp(str, '(?<!u)r', 'start')
```

```
startIndex =
    13
```

Example 2 — Lookbehind. Return the names and 7-digit telephone numbers for those people in the list that are in the 617 area code. The lookbehind `(?<=^617-)` finds those lines that begin with the number 617:

```
phone_list = {...
'978-389-2457 Kevin';       '617-922-3091 Ruth'; ...
'781-147-1748 Alan';        '508-643-9648 George'; ...
'617-774-6642 Lisa';        '617-241-0275 Greg'; ...
'413-995-9114 Jason';       '781-276-0482 Victoria'};
len = length(phone_list);

ph617 = regexp(phone_list, '(?<=^617-).*', 'match');

for k=1:len
str = char(ph617{k});
if ~isempty(str),   fprintf('   %s\n', str),   end
end
```

MATLAB returns the three numbers that have a 617 area code:

```
   922-3091 Ruth
   774-6642 Lisa
   241-0275 Greg
```

Using the Negative Lookbehind Operator— (?<!test)expr

Example — Negative Lookbehind. This example uses negative lookbehind to find those tasks that are not labelled as `Done` or `Pending`, Create a list of tasks, each with status information to the left:

```
tasks = {...
'ToDo     3892457';       'Done     9223091'; ...
'Pending  1471748';       'Maybe    7746642'; ...
'ToDo     2410275';       'Pending  4723596'; ...
'ToDo     9959114';       'Maybe    2760482'; ...
'ToDo     3080027';       'Done     1221941'};
count = length(tasks);
```

The regular expression looks for those task numbers that do not have a `Done` or `Pending` status. Note that you can use the `or` (|) operator in a lookaround to check for more than one condition:

```
doNow = regexp(tasks, '(?<!^(Done|Pending).*)\d+', 'match');
```

Now print out the results:

```
disp 'The following tasks need attention:'
for k=1:count
```

```
    s = char(doNow{k});
    if ~isempty(s),   fprintf('   %s\n', s),   end
end
```

The output displays all but the `Done` and `Pending` tasks:

```
The following tasks need attention:
   3892457
   7746642
   2410275
   9959114
   2760482
   3080027
```

Using Lookaround as a Logical Operator

One way in which a lookahead operation can be useful is to perform a logical AND between two conditions. This example initially attempts to locate all lowercase consonants in a text string. The text string is the first 50 characters of the help for the `normest` function:

```
helptext = help('normest');
str = helptext(1:50)
str =
 NORMEST Estimate the matrix 2-norm.
    NORMEST(S
```

Merely searching for non-vowels (`[^aeiouAEIOU]`) does not return the expected answer, as the output includes capital letters, space characters, and punctuation:

```
c = regexp(str, '[^aeiouAEIOU]', 'match')
c =
  Columns 1 through 12
    ' '  'N'  'R'  'M'  'S'  'T'  ' '  's'  't'  'm' 't'

      -- etc. --
```

Try this again, using a lookahead operator to create the following AND condition:

```
(lowercase letter) AND (not a vowel).
```

This time, the result is correct:

```
c = regexp(str, '(?=[a-z])[^aeiou]', 'match')
c =
  's'  't'  'm '  't'  't'  'h'  'm'  't'  'r'  'x'
     'n'  'r'  'm'
```

Note that when using a lookahead operator to perform an AND, you need to place the match expression `expr` *after* the test expression `test`:

```
(?=test)expr or (?!test)expr
```

Back to Top

Quantifiers

With the quantifiers shown below, you can specify how many instances of an element are to be matched. The basic quantifying operators are listed in the first six rows of the table.

By default, MATLAB matches as much of an expression as possible. Using the operators shown in the last two rows of the table, you can override this default behavior. Specify these options by appending a `+` or `?` immediately following one of the six basic quantifying operators.

Operator	Usage
`expr{m,n}`	Must occur at least `m` times but no more than `n` times.
`expr{m,}`	Must occur at least `m` times.
`expr{n}`	Must match exactly `n` times. Equivalent to `{n,n}`.
`expr?`	Match the preceding element 0 times or 1 time. Equivalent to `{0,1}`.
`expr*`	Match the preceding element 0 or more times. Equivalent to `{0,}`.
`expr+`	Match the preceding element 1 or more times. Equivalent to `{1,}`.
`q_expr+`	Match as much of the quantified expression as possible, but do not rescan any portions of the string if the initial match fails. The term `q_expr` represents any of the expressions shown in the top six rows of this table.
`q_expr?`	Match only as much of the quantified expression as necessary. The term `q_expr` represents any of the expressions shown in the top six rows of this table. For an example, see Lazy Quantifiers — expr*?, below.

Zero or One — expr?

Use `?` to make the HTML `<code>` and `</code>` tags optional in the string. The first string, `hstr1`, contains one occurrence of each tag. Since the expression uses `()?` around the tags, one occurrence is a match:

```
hstr1 = '<td><a name="18854"></a><code>%%</code><br></td>';
expr = '</a>(<code>)?..(</code>)?<br>';

regexp(hstr1, expr, 'match')
ans =
    '</a><code>%%</code><br>'
```

The second string, `hstr2`, does not contain the code tags at all. Just the same, the expression matches because `()?` allows for zero occurrences of the tags:

```
hstr2 = '<td><a name="18854"></a>%%<br></td>';
expr = '</a>(<code>)?..(</code>)?<br>';

regexp(hstr2, expr, 'match')
ans =
    '</a>%%<br>'
```

Zero or More — expr*

The first `regexp` command looks for at least one occurrence of `<br>` and finds it. The second command parses a different string for at least one `<br>` and fails. The third command uses `*` to parse the same line for zero or more line breaks and this time succeeds.

```
hstr1 = '<p>This string has <br><br>line breaks</p>';
regexp(hstr1, '<p>.*(<br>).*</p>', 'match')
ans =
    '<p>This string has <br><br>line breaks</p>';

hstr2 = '<p>This string has no line breaks</p>';
regexp(hstr2, '<p>.*(<br>).*</p>', 'match')
ans =
     {}

regexp(hstr2, '<p>.*(<br>)*.*</p>', 'match')
ans =
    '<p>This string has no line breaks</p>';
```

One or More — expr+

Use `+` to verify that the HTML image source is not empty. This looks for one or more characters in the `gif` filename:

```
hstr = '<a href="s12.html"><img src="b_prev.gif" border=0>';
expr = '<img src="\w+.gif';

regexp(hstr, expr, 'match')
ans =
    '<img src="b_prev.gif'
```

Exact, Minimum, and Maximum Quantities — {min,max}

Use `{m}`, `{m,}`, and `{m,n}` to verify the `href` syntax used in HTML. This statement requires the `href` to have at least one nonwhitespace character, followed by exactly one occurrence of `.html`, optionally followed by `#` and five to eight digits:

```
hstr = '<a name="18749"></a><a href="s13.html#18760">';
expr = '<a href="\w{1,}(\.html){1}(\#\d{5,8}){0,1}"';

regexp(hstr, expr, 'match')
```

```
ans =
    '<a href="s13.html#18760"'
```

Lazy Quantifiers — expr*?

This example shows the difference between the default (*greedy*) quantifier and the *lazy* quantifier (`?`). The first part of the example uses the default quantifier to match all characters from the opening `<tr` to the ending `</td`:

```
hstr = '<tr valign=top><td><a name="19184"></a><br></td>';
regexp(hstr, '</?t.*>', 'match')
ans =
    '<tr valign=top><td><a name="19184"></a><br></td>'
```

The second part uses the lazy quantifier to match the minimum number of characters between `<tr`, `<td`, or `</td` tags:

```
regexp(hstr, '</?t.*?>', 'match')
ans =
    '<tr valign=top>'    '<td>'    '</td>'
```

Back to Top

Tokens

Parentheses used in a regular expression not only group elements of that expression together, but also designate any matches found for that group as *tokens*. You can use tokens to match other parts of the same string. One advantage of using tokens is that they remember what they matched, so you can recall and reuse matched text in the process of searching or replacing.

This section covers

Operators Used with Tokens

Here are the operators you can use with tokens in MATLAB.

Operator	Usage
`(expr)`	Capture in a token all characters matched by the expression within the parentheses.

`\N`	Match the `N`th token generated by this command. That is, use `\1` to match the first token, `\2` to match the second, and so on.
`$N`	Insert the match for the `N`th token in the replacement string. Used only by the `regexprep` function. If `N` is equal to zero, then insert the entire match in the replacement string.
`(?(N)s1|s2)`	If `N`th token is found, then match `s1`, else match `s2`

Introduction to Using Tokens

You can turn any pattern being matched into a token by enclosing the pattern in parentheses within the expression. For example, to create a token for a dollar amount, you could use '`(\$\d+)`'. Each token in the expression is assigned a number, starting from 1, going from left to right. To make a reference to a token later in the expression, refer to it using a backslash followed by the token number. For example, when referencing a token generated by the third set of parentheses in the expression, use `\3`.

As a simple example, if you wanted to search for identical sequential letters in a string, you could capture the first letter as a token and then search for a matching character immediately afterwards. In the expression shown below, the `(\S)` phrase creates a token whenever `regexp` matches any nonwhitespace character in the string. The second part of the expression, `'\1'`, looks for a second instance of the same character immediately following the first:

```
poestr = ['While I nodded, nearly napping, ' ...
          'suddenly there came a tapping,'];

[mat tok ext] = regexp(poestr, '(\S)\1', 'match', ...
   'tokens', 'tokenExtents');
mat
mat =
    'dd'    'pp'    'dd'    'pp'
```

The tokens returned in cell array `tok` are:

```
'd', 'p', 'd', 'p'
```

Starting and ending indices for each token in the input string `poestr` are:

```
11 11,  26 26,  35 35,  57 57
```

Using Tokens — Example 1

Here is an example of how tokens are assigned values. Suppose that you are going to search the following text:

```
andy ted bob jim andrew andy ted mark
```

You choose to search the above text with the following search pattern:

```
and(y|rew)|(t)e(d)
```

This pattern has three parenthetical expressions that generate tokens. When you finally perform the search, the following tokens are generated for each match.

Match	Token 1	Token 2
andy	y	
ted	t	d
andrew	rew	
andy	y	
ted	t	d

Only the highest level parentheses are used. For example, if the search pattern `and(y|rew)` finds the text `andrew`, token 1 is assigned the value `rew`. However, if the search pattern `(and(y|rew))` is used, token 1 is assigned the value `andrew`.

Using Tokens — Example 2

Use `(expr)` and `\N` to capture pairs of matching HTML tags (e.g., `<a>` and `</a>`) and the text between them. The expression used for this example is

```
expr = '<(\w+).*?>.*?</\1>';
```

The first part of the expression, '`<(\w+)`', matches an opening bracket (`<`) followed by one or more alphabetic, numeric, or underscore characters. The enclosing parentheses capture token characters following the opening bracket.

The second part of the expression, '`.*?>.*?`', matches the remainder of this HTML tag (characters up to the `>`), and any characters that may precede the next opening bracket.

The last part, '`</\1>`', matches all characters in the ending HTML tag. This tag is composed of the sequence `</tag>`, where `tag` is whatever characters were captured as a token.

```
hstr = '<!comment><a name="752507"></a><b>Default</b><br>';
expr = '<(\w+).*?>.*?</\1>';

[mat tok] = regexp(hstr, expr, 'match', 'tokens');
mat{:}
ans =
    <a name="752507"></a>
ans =
    <b>Default</b>

tok{:}
```

```
ans =
    'a'
ans =
    'b'
```

Tokens That Are Not Matched

For those tokens specified in the regular expression that have no match in the string being evaluated, `regexp` and `regexpi` return an empty string (`''`) as the token output, and an extent that marks the position in the string where the token was expected.

The example shown here executes `regexp` on the path string `str` returned from the MATLAB `tempdir` function. The regular expression `expr` includes six token specifiers, one for each piece of the path string. The third specifier `[a-z]+` has no match in the string because this part of the path, `Profiles`, begins with an uppercase letter:

```
str = tempdir
str =
   C:\WINNT\Profiles\bpascal\LOCALS~1\Temp\

expr = ['([A-Z]:)\\(WINNT)\\([a-z]+)?.*\\' ...
        '([a-z]+)\\([A-Z]+~\d)\\(Temp)\\'];

[tok ext] = regexp(str, expr, 'tokens', 'tokenExtents');
```

When a token is not found in a string, MATLAB still returns a token string and token extent. The returned token string is an empty character string (`''`). The first number of the extent is the string index that marks where the token was expected, and the second number of the extent is equal to one less than the first.

In the case of this example, the empty token is the third specified in the expression, so the third token string returned is empty:

```
tok{:}
ans =
    'C:'    'WINNT'     ''    'bpascal'    'LOCALS~1'    'Temp'
```

The third token extent returned in the variable `ext` has the starting index set to 10, which is where the nonmatching substring, `Profiles`, begins in the string. The ending extent index is set to one less than the starting index, or 9:

```
ext{:}
ans =
     1     2
     4     8
    10     9
    19    25
    27    34
    36    39
```

Using Tokens in a Replacement String

When using tokens in a replacement string, reference them using `$1`, `$2`, etc. instead of `\1`, `\2`, etc. This example captures two tokens and reverses their order. The first, `$1`, is `'Norma Jean'` and the second, `$2`, is `'Baker'`. Note that `regexprep` returns the modified string, not a vector of starting indices.

```
regexprep('Norma Jean Baker', '(\w+\s\w+)\s(\w+)', '$2, $1')
ans =
    Baker, Norma Jean
```

Back to Top

Named Capture

If you use a lot of tokens in your expressions, it may be helpful to assign them names rather than having to keep track of which token number is assigned to which token. Use the following operator to assign a name to a token that finds a match.

Operator	Usage
`(?<name>expr)`	Capture in a token all characters matched by the expression within the parentheses. Assign a `name` to the token.
`\k<name>`	Match the token referred to by `name`.
`$<name>`	Insert the match for named token in a replacement string. Used only with the `regexprep` function.
`(?(name)s1|s2)`	If named token is found, then match `s1`; otherwise, match `s2`

When referencing a named token within the expression, use the syntax `\k<name>` instead of the numeric `\1`, `\2`, etc.:

```
poestr = ['While I nodded, nearly napping, ' ...
          'suddenly there came a tapping,'];

regexp(poestr, '(?<anychar>.)\k<anychar>', 'match')
ans =
    'dd'    'pp'    'dd'    'pp'
```

Labeling Your Output

Named tokens can also be useful in labeling the output from the MATLAB regular expression functions. This is especially true when you are processing numerous strings.

This example parses different pieces of street addresses from several strings. A short name is assigned to each token in the expression string:

```
str1 = '134 Main Street, Boulder, CO, 14923';
str2 = '26 Walnut Road, Topeka, KA, 25384';
```

```
str3 = '847 Industrial Drive, Elizabeth, NJ, 73548';

p1 = '(?<adrs>\d+\s\S+\s(Road|Street|Avenue|Drive))';
p2 = '(?<city>[A-Z][a-z]+)';
p3 = '(?<state>[A-Z]{2})';
p4 = '(?<zip>\d{5})';

expr = [p1 ', ' p2 ', ' p3 ', ' p4];
```

As the following results demonstrate, you can make your output easier to work with by using named tokens:

```
loc1 = regexp(str1, expr, 'names')
loc1 =
     adrs: '134 Main Street'
     city: 'Boulder'
    state: 'CO'
      zip: '14923'

loc2 = regexp(str2, expr, 'names')
loc2 =
     adrs: '26 Walnut Road'
     city: 'Topeka'
    state: 'KA'
      zip: '25384'

loc3 = regexp(str3, expr, 'names')
loc3 =
     adrs: '847 Industrial Drive'
     city: 'Elizabeth'
    state: 'NJ'
      zip: '73548'
```

Back to Top

Conditional Expressions

With conditional expressions, you can tell MATLAB to match an expression only if a certain condition is true. A conditional expression is similar to an `if-then` or an `if-then-else` clause in programming. MATLAB first tests the state of a given condition, and the outcome of this tests determines what, if anything, is to be matched next. The following table shows the two conditional syntaxes you can use with MATLAB.

Operator	Usage
`(?(cond)expr)`	If condition `cond` is true, then match expression `expr`

`(?(cond)expr`$_1$`|expr`$_2$`)`	If condition `cond` is true, then match expression `expr`$_1$. Otherwise match expression `expr`$_2$

The first entry in this table is the same as an `if-then` statement. MATLAB tests the state of condition `cond` and then matches expression `expr` only if the condition was found to be true. In the form of an `if-then` statement, it would look like this:

```
if cond then expr
```

The second entry in the table is the same as an `if-then-else` statement. If the condition is true, MATLAB matches `expr`$_1$; if false, it matches `expr`$_2$ instead. This syntax is equivalent to the following programming statement:

```
if cond then expr1 else expr2
```

The condition `cond` in either of these syntaxes can be any one of the following:

- A specific token, identified by either number or name, is located in the input string. See Conditions Based on Tokens, below.
- A lookaround operation results in a match. See Conditions Based on a Lookaround Match, below.
- A dynamic expression of the form `(?@cmd)` returns a nonzero numeric value. See Conditions Based on Return Values, below.

Conditions Based on Tokens

In a conditional expression, MATLAB matches the expression only if the condition associated with it is met. If the condition is based on a token, then the condition is met if MATLAB matches more than one character for the token in the input string.

To specify a token in a condition, use either the token number or, for tokens that you have assigned a name to, its name. Token numbers are determined by the order in which they appear in an expression. For example, if you specify three tokens in an expression (that is, if you enclose three parts of the expression in parentheses), then you would refer to these tokens in a condition statement as `1`, `2`, and `3`.

The following example uses the conditional statement `(?(1)her|his)` to match the string regardless of the gender used. You could translate this into the phrase, "**if** token `1` is found (i.e., `Mr` is followed by the letter `s`), **then** match `her`, **else** match `his`:

```
expr = 'Mr(s?)\..*?(?(1)her|his) son';

[mat tok] = regexp('Mr. Clark went to see his son', ...
   expr, 'match', 'tokens')
mat =
    'Mr. Clark went to see his son'
tok =
    {1x2 cell}

tok{:}
```

```
ans =
     ''    'his'
```

In the second part of the example, the token `s` is found and MATLAB matches the word `her`:

```
[mat tok] = regexp('Mrs. Clark went to see her son', ...
expr, 'match', 'tokens')
mat =
    'Mrs. Clark went to see her son'
tok =
    {1x2 cell}

tok{:}
ans =
    's'    'her'
```

> **Note** When referring to a token within a condition, use just the number of the token. For example, refer to token `2` by using the number `2` alone, and not `\2` or `$2`.

Conditions Based on a Lookaround Match

Lookaround statements look for text that either precedes or follows an expression. If this lookaround text is located, then MATLAB proceeds to match the expression. You can also use lookarounds in conditional statements. In this case, if the lookaround text is located, then MATLAB considers the condition to be met and matches the associated expression. If the condition is not met, then MATLAB matches the `else` part of the expression.

Conditions Based on Return Values

MATLAB supports different types of dynamic expressions. One type of dynamic expression, having the form `(?@cmd)`, enables you to execute a MATLAB command (shown here as `cmd`) while matching an expression. You can use this type of dynamic expression in a conditional statement if the command in the expression returns a numeric value. The condition is considered to be met if the return value is nonzero.

Back to Top

Dynamic Regular Expressions

In a dynamic expression, you can make the pattern that you want `regexp` to match dependent on the content of the input string. In this way, you can more closely match varying input patterns in the string being parsed. You can also use dynamic expressions in replacement strings for use with the `regexprep` function. This gives you the ability to adapt the replacement text to the parsed input.

You can include any number of dynamic expressions in the `match_expr` or `replace_expr` arguments of these commands:

```
regexp(string, match_expr)
regexpi(string, match_expr)
```

```
regexprep(string, match_expr, replace_expr)
```

MATLAB supports three types of dynamic operators for use in a match expression. See Dynamic Operators for the Match Expression for more information.

Operator	Usage
`(??expr)`	Parse `expr` as a separate regular expression, and include the resulting string in the match expression. This gives you the same results as if you called `regexprep` inside of a `regexp` match expression.
`(?@cmd)`	Execute the MATLAB command `cmd`, discarding any output that may be returned. This is often used for diagnosing a regular expression.
`(??@cmd)`	Execute the MATLAB command `cmd`, and include the string returned by `cmd` in the match expression. This is a combination of the two dynamic syntaxes shown above: `(??expr)` and `(?@cmd)`.

MATLAB supports one type of dynamic expression for use in the replacement expression of a `regexprep` command. See Dynamic Operators for the Replacement Expression for more information.

Operator	Usage
`${cmd}`	Execute the MATLAB command `cmd`, and include the string returned by `cmd` in the replacement expression.

Example of a Dynamic Expression

As an example of a dynamic expression, the following `regexprep` command correctly replaces the term `internationalization` with its abbreviated form, `i18n`. However, to use it on a different term such as `globalization`, you have to use a different replacement expression:

```
match_expr = '(^\w)(\w*)(\w$)';

replace_expr1 = '$118$3';
regexprep('internationalization', match_expr, replace_expr1)
ans =
    i18n

replace_expr2 = '$111$3';
regexprep('globalization', match_expr, replace_expr2)
ans =
    g11n
```

Using a dynamic expression `${num2str(length($2))}` enables you to base the

replacement expression on the input string so that you do not have to change the expression each time. This example uses the dynamic syntax `${cmd}` from the second table shown above:

```
match_expr = '(^\w)(\w*)(\w$)';
replace_expr = '$1${num2str(length($2))}$3';

regexprep('internationalization', match_expr, replace_expr)
ans =
    i18n

regexprep('globalization', match_expr, replace_expr)
ans =
    g11n
```

Dynamic Operators for the Match Expression

There are three types of dynamic expressions you can use when composing a match expression:

- Dynamic Expressions That Modify the Match Expression — (??expr)
- Dynamic Commands That Modify the Match Expression — (??@cmd)
- Dynamic Commands That Serve a Functional Purpose — (?@cmd)

The first two of these actually modify the match expression itself so that it can be made specific to changes in the contents of the input string. When MATLAB evaluates one of these dynamic statements, the results of that evaluation are included in the same location within the overall match expression.

The third operator listed here does not modify the overall expression, but instead enables you to run MATLAB commands during the parsing of a regular expression. This functionality can be useful in diagnosing your regular expressions.

Dynamic Expressions That Modify the Match Expression — (??expr). The `(??expr)` operator parses expression `expr`, and inserts the results back into the match expression. MATLAB then evaluates the modified match expression.

Here is an example of the type of expression that you can use with this operator:

```
str = {'5XXXXX', '8XXXXXXXX', '1X'};
regexp(str, '^(\d+)(??X{$1})$', 'match', 'once')
```

The purpose of this particular command is to locate a series of `X` characters in each of the strings stored in the input cell array. Note however that the number of `X`s varies in each string. If the count did not vary, you could use the expression `X{n}` to indicate that you want to match `n` of these characters. But, a constant value of `n` does not work in this case.

The solution used here is to capture the leading count number (e.g., the `5` in the first string of the cell array) in a token, and then to use that count in a dynamic expression. The dynamic expression in this example is `(??X{$1})`, where `$1` is the value captured by the token `\d+`. The operator `{$1}` makes a quantifier of that token value. Because the expression is dynamic, the same pattern works on all three of the input strings in the cell

array. With the first input string, `regexp` looks for five `X` characters; with the second, it looks for eight, and with the third, it looks for just one:

```
regexp(str, '^(\d+)(??X{$1})$', 'match', 'once')
ans =
    '5XXXXX'    '8XXXXXXXX'    '1X'
```

Dynamic Commands That Modify the Match Expression — (??@cmd). MATLAB uses the `(??@function)` operator to include the results of a MATLAB command in the match expression. This command must return a string that can be used within the match expression.

The `regexp` command below uses the dynamic expression `(??@flilplr($1))` to locate a palindrome string, "Never Odd or Even", that has been embedded into a larger string:

```
regexp(pstr, '(.{3,}).?(??@fliplr($1))', 'match')
```

The dynamic expression reverses the order of the letters that make up the string, and then attempts to match as much of the reversed-order string as possible. This requires a dynamic expression because the value for `$1` relies on the value of the token `(.{3,})`:

```
% Put the string in lowercase.
str = lower(...
  'Find the palindrome Never Odd or Even in this string');

% Remove all nonword characters.
str = regexprep(str, '\W*', '')
str =
   findthepalindromeneveroddoreveninthisstring

% Now locate the palindrome within the string.
palstr = regexp(str, '(.{3,}).?(??@fliplr($1))', 'match')
str =
   'neveroddoreven'
```

Dynamic expressions in MATLAB have access to the currently active workspace. This means that you can change any of the functions or variables used in a dynamic expression just by changing variables in the workspace. Repeat the last command of the example above, but this time define the function to be called within the expression using a function handle stored in the base workspace:

```
fun = @fliplr;

palstr = regexp(str, '(.{3,}).?(??@fun($1))', 'match')
palstr =
   'neveroddoreven'
```

Dynamic Commands That Serve a Functional Purpose — (?@cmd). The `(?@cmd)` operator specifies a MATLAB command that `regexp` or `regexprep` is to run while parsing the overall match expression. Unlike the other dynamic expressions in MATLAB, this operator does not alter the contents of the expression it is used in. Instead, you can use

this functionality to get MATLAB to report just what steps it is taking as it parses the contents of one of your regular expressions.

The following example parses a word for zero or more characters followed by two identical characters followed again by zero or more characters:

```
regexp('mississippi', '\w*(\w)\1\w*', 'match')
ans =
    'mississippi'
```

To track the exact steps that MATLAB takes in determining the match, the example inserts a short script `(?@disp($1))` in the expression to display the characters that finally constitute the match. Because the example uses greedy quantifiers, MATLAB attempts to match as much of the string as possible. So, even though MATLAB finds a match toward the beginning of the string, it continues to look for more matches until it arrives at the very end of the string. From there, it backs up through the letters `i` then `p` and the next `p`, stopping at that point because the match is finally satisfied:

```
regexp('mississippi', '\w*(\w)(?@disp($1))\1\w*');
i
p
p
```

Now try the same example again, this time making the first quantifier lazy (`*?`). Again, MATLAB makes the same match:

```
regexp('mississippi', '\w*?(\w)\1\w*', 'match')
ans =
    'mississippi'
```

But by inserting a dynamic script, you can see that this time, MATLAB has matched the string quite differently. In this case, MATLAB uses the very first match it can find, and does not even consider the rest of the string:

```
regexp('mississippi', '\w*?(\w)(?@disp($1))\1\w*';)
m
i
s
```

To demonstrate how versatile this type of dynamic expression can be, consider the next example that progressively assembles a cell array as MATLAB iteratively parses the input string. The `(?!)` operator found at the end of the expression is actually an empty lookahead operator, and forces a failure at each iteration. This forced failure is necessary if you want to trace the steps that MATLAB is taking to resolve the expression.

MATLAB makes a number of passes through the input string, each time trying another combination of letters to see if a fit better than last match can be found. On any passes in which no matches are found, the test results in an empty string. The dynamic script `(?@if(~isempty($&)))` serves to omit these strings from the `matches` cell array:

```
matches = {};
```

```
expr = ['(Euler\s)?(Cauchy\s)?(Boole)?(?@if(~isempty($&)),' ...
   'matches{end+1}=$&;end)(?!)'];

regexp('Euler Cauchy Boole', expr);

matches
matches =
    'Euler Cauchy Boole'    'Euler Cauchy '    'Euler '
'Cauchy Boole'    'Cauchy '    'Boole'
```

The operators `$&` (or the equivalent `$0`), `$``, and `$'` refer to that part of the input string that is currently a match, all characters that precede the current match, and all characters to follow the current match, respectively. These operators are sometimes useful when working with dynamic expressions, particularly those that employ the `(?@cmd)` operator.

This example parses the input string looking for the letter `g`. At each iteration through the string, `regexp` compares the current character with `g`, and not finding it, advances to the next character. The example tracks the progress of scan through the string by marking the current location being parsed with a `^` character.

(The `$`` and `$´` operators capture that part of the string that precedes and follows the current parsing location. You need two single-quotation marks (`$''`) to express the sequence `$´` when it appears within a string.)

```
str = 'abcdefghij';
expr = '(?@disp(sprintf(''starting match: [%s^%s]'',$`,$'')))g';

regexp(str, expr, 'once');
starting match: [^abcdefghij]
starting match: [a^bcdefghij]
starting match: [ab^cdefghij]
starting match: [abc^defghij]
starting match: [abcd^efghij]
starting match: [abcde^fghij]
starting match: [abcdef^ghij]
```

Dynamic Operators for the Replacement Expression

The three types of dynamic expressions discussed above can be used only in the match expression (second input) argument of the regular expression functions. MATLAB provides one more type of dynamic expression; this one is for use in a replacement string (third input) argument of the `regexprep` function.

Dynamic Commands That Modify the Replacement Expression — ${cmd}. The `${cmd}` operator modifies the contents of a regular expression replacement string, making this string adaptable to parameters in the input string that might vary from one use to the next. As with the other dynamic expressions used in MATLAB, you can include any number of these expressions within the overall replacement expression.

In the `regexprep` call shown here, the replacement string is `'${convert($1,$2)}'`. In this case, the entire replacement string is a dynamic expression:

```
regexprep('This highway is 125 miles long', ...
          '(\d+\.?\d*)\W(\w+)', '${convert($1,$2)}')
```

The dynamic expression tells MATLAB to execute a function named `convert` using the two tokens `(\d+\.?\d*)` and `(\w+)`, derived from the string being matched, as input arguments in the call to `convert`. The replacement string requires a dynamic expression because the values of `$1` and `$2` are generated at runtime.

The following example defines the file named `convert` that converts measurements from imperial units to metric. To convert values from the string being parsed, `regexprep` calls the `convert` function, passing in values for the quantity to be converted and name of the imperial unit:

```
function valout  = convert(valin, units)
switch(units)
    case 'inches'
        fun = @(in)in .* 2.54;    uout = 'centimeters';
    case 'miles'
        fun = @(mi)mi .* 1.6093;  uout = 'kilometers';
    case 'pounds'
        fun = @(lb)lb .* 0.4536;  uout = 'kilograms';
    case 'pints'
        fun = @(pt)pt .* 0.4731;  uout = 'litres';
    case 'ounces'
        fun = @(oz)oz .* 28.35;   uout = 'grams';
end
val = fun(str2num(valin));
valout = [num2str(val) ' ' uout];


regexprep('This highway is 125 miles long', ...
          '(\d+\.?\d*)\W(\w+)', '${convert($1,$2)}')
ans =
   This highway is 201.1625 kilometers long


regexprep('This pitcher holds 2.5 pints of water', ...
          '(\d+\.?\d*)\W(\w+)', '${convert($1,$2)}')
ans =
   This pitcher holds 1.1828 litres of water


regexprep('This stone weighs about 10 pounds', ...
          '(\d+\.?\d*)\W(\w+)', '${convert($1,$2)}')
ans =
   This stone weighs about 4.536 kilograms
```

As with the `(??@ )` operator discussed in an earlier section, the `${ }` operator has access to variables in the currently active workspace. The following `regexprep` command uses the

array `A` defined in the base workspace:

```
A = magic(3)
A =
     8     1     6
     3     5     7
     4     9     2

regexprep('The columns of matrix _nam are _val', ...
          {'_nam', '_val'}, ...
          {'A', '${sprintf(''%d%d%d '', A)}'})
ans =
The columns of matrix A are 834 159 672
```

Back to Top

String Replacement

The `regexprep` function enables you to replace a string that is identified by a regular expression with another string. The following syntax replaces all occurrences of the regular expression `expr` in string `str` with the string `repstr`. The new string is returned in `s`. If no matches are found, return string `s` is the same as input string `str`.

```
s = regexprep('str', 'expr', 'repstr')
```

The replacement string can include any ordinary characters and also any of the operators shown in the following table:

Operator	Usage
Operators from the Character Representation table	The character represented by the operator sequence
`$``	That part of the input string that precedes the current match
`$&` or `$0`	That part of the input string that is currently a match
`$'`	That part of the input string that follows the current match. In MATLAB, use `$''` to represent the character sequence `$'`
`$N`	The string represented by the token identified by the number `N`
`$<name>`	The string represented by the token identified by `name`
`${cmd}`	The string returned when MATLAB executes the command `cmd`

You can capture parts of the input string as tokens and then reuse them in the replacement string. Specify the parts of the string to capture using the token capture operator `(...)`. Specify the tokens to use in the replacement string using the operators `$1`, `$2`, `$N` to reference the first, second, and `N`th tokens captured. (See the section on Tokens and the example Using Tokens in a Replacement String in this documentation for information on using tokens.)

> **Note** When referring to a token within a replacement string, use the number of the token preceded by a dollar sign. For example, refer to token `2` by using `$2`, and not `2` or `\2`.

The following example uses both the `${cmd}` and `$N` operators in the replacement strings of nested `regexprep` commands to capitalize the first letter of each sentence. The inner `regexprep` looks for the start of the entire string and capitalizes the single instance; the outer `regexprep` looks for the first letter following a period and capitalizes the two instances:

```
s1 = 'here are a few sentences.';
s2 = 'none are capitalized.';
s3 = 'let''s change that.';
str = [s1 ' ' s2 ' ' s3]

regexprep(regexprep(str, '(^.)', '${upper($1)}'), ...
   '(?<=\.\s*)([a-z])','${upper($1)}')

ans =
Here are a few sentences. None are capitalized. Let's change that
```

Make `regexprep` more specific to your needs by specifying any of a number of options with the command. See the `regexprep` reference page for more information on these options.

Back to Top

Handling Multiple Strings

You can use any of the MATLAB regular expression functions with cell arrays of strings as well as with single strings. Any or all of the input parameters (the string, expression, or replacement string) can be a cell array of strings. The `regexp` function requires that the string and expression arrays have the same number of elements. The `regexprep` function requires that the expression and replacement arrays have the same number of elements. (The cell arrays do not have to have the same shape.)

Whenever either input argument in a call to `regexp`, or the first input argument in a call to `regexprep` function is a cell array, all output values are cell arrays of the same size.

This section covers the following topics:

- Finding a Single Pattern in Multiple Strings

- Finding Multiple Patterns in Multiple Strings
- Replacing Multiple Strings

Finding a Single Pattern in Multiple Strings

The example shown here uses the `regexp` function on a cell array of strings `cstr`. It searches each string of the cell array for consecutive matching letters (e.g., `'oo'`). The function returns a cell array of the same size as the input array. Each row of the return array contains the indices for which there was a match against the input cell array.

Here is the input cell array:

```
cstr = {                                  ...
'Whose woods these are I think I know.' ; ...
'His house is in the village though;'   ; ...
'He will not see me stopping here'      ; ...
'To watch his woods fill up with snow.'};
```

Find consecutive matching letters by capturing a letter as a token `(.)` and then repeating that letter as a token reference, `\1`:

```
idx = regexp(cstr, '(.)\1');

whos idx
  Name      Size                    Bytes  Class

  idx       4x1                       296  cell array

idx{:}
ans =                 % 'Whose woods these are I think I know.'
    8                 %         |8

ans =                 % 'His house is in the village though;'
   23                 %                         |23

ans =                 % 'He will not see me stopping here'
    6    14    23     %       |6      |14      |23

ans =                 % 'To watch his woods fill up with snow.'
   15    22           %                |15    |22
```

To return substrings instead of indices, use the `'match'` parameter:

```
mat = regexp(cstr, '(.)\1', 'match');
mat{3}
ans =
   'll'    'ee'    'pp'
```

Finding Multiple Patterns in Multiple Strings

This example uses a cell array of strings in both the input string and the expression. The

two cell arrays are of different shapes: `cstr` is 4-by-1 while `expr` is 1-by-4. The command is valid as long as they both have the same number of cells.

Find uppercase or lowercase `'i'` followed by a white-space character in `str{1}`, the sequence `'hou'` in `str{2}`, two consecutive matching letters in `str{3}`, and words beginning with `'w'` followed by a vowel in `str{4}`.

```
expr = {'i\s', 'hou', '(.)\1', '\<w[aeiou]'};
idx = regexpi(cstr, expr);

idx{:}
ans =                  % 'Whose woods these are I think I know.'
   23    31            %                         |23     |31

ans =                  % 'His house is in the village though;'
    5    30            %      |5                       |30

ans =                  % 'He will not see me stopping here'
    6    14    23      %       |6      |14      |23

ans =                  % 'To watch his woods fill up with snow.'
    4    14    28      %     |4        |14           |28
```

Note that the returned cell array has the dimensions of the input string, `cstr`. The dimensions of the return value are always derived from the input string, whenever the input string is a cell array. If the input string is not a cell array, then it is the dimensions of the expression that determine the shape of the return array.

Replacing Multiple Strings

When replacing multiple strings with `regexprep`, use a single replacement string if the expression consists of a single string. This example uses a common replacement value (`'--'`) for all matches found in the multiple string input `cstr`. The function returns a cell array of strings having the same dimensions as the input cell array:

```
s = regexprep(cstr, '(.)\1', '--', 'ignorecase')
s =
    'Whose w--ds these are I think I know.'
    'His house is in the vi--age though;'
    'He wi-- not s-- me sto--ing here'
    'To watch his w--ds fi-- up with snow.'
```

You can use multiple replacement strings if the expression consists of multiple strings. In this example, the input string and replacement string are both 4-by-1 cell arrays, and the expression is a 1-by-4 cell array. As long as the expression and replacement arrays contain the same number of elements, the statement is valid. The dimensions of the return value match the dimensions of the input string:

```
expr = {'i\s', 'hou', '(.)\1', '\<w[aeiou]'};
repl = {'-1-'; '-2-'; '-3-'; '-4-'};
```

```
s = regexprep(cstr, expr, repl, 'ignorecase')
s =
    'Whose w-3-ds these are -1-think -1-know.'
    'His -2-se is in the vi-3-age t-2-gh;'
    'He -4--3- not s-3- me sto-3-ing here'
    'To -4-tch his w-3-ds fi-3- up -4-th snow.'
```

Back to Top

Function, Mode Options, Operator, Return Value Summaries

Function Summary

MATLAB Regular Expression Functions

Function	Description
regexp	Match regular expression.
regexpi	Match regular expression, ignoring case.
regexprep	Replace string using regular expression.
regexptranslate	Translate string into regular expression.

Mode Options Summary

Mode Keyword	Flag	Description
'ignorecase'	(?i)	Do not consider letter case when matching patterns to a string. (The default for regexpi)
'matchcase'	(?-i)	Letter case must match when matching patterns to a string. (The default for regexp)
'dotall'	(?s)	Match dot ('.') in the pattern string with any character. (The default)
'dotexceptnewline'	(?-s)	Match dot in the pattern with any character that is not a newline.
'lineanchors'	(?m)	Match the ^ and $ metacharacters at the beginning and end of a line.

'stringanchors'	(?-m)	Match the ^ and $ metacharacters at the beginning and end of a string (the default).
'freespacing'	(?x)	Ignore spaces and comments when parsing the string. (You must use '\ ' and '\#' to match space and # characters.)
'literalspacing'	(?-x)	Parse space characters and comments (the # character and any text to the right of it) in the same way as any other characters in the string. (The default)

Operator Summary

Character Types

Operator	Usage
.	Any single character, including white space
[$c_1c_2c_3$]	Any character contained within the brackets: c_1 or c_2 or c_3
[^$c_1c_2c_3$]	Any character not contained within the brackets: anything but c_1 or c_2 or c_3
[c_1-c_2]	Any character in the range of c_1 through c_2
\s	Any white-space character; equivalent to [\f\n\r\t\v]
\S	Any nonwhitespace character; equivalent to [^ \f\n\r\t\v]
\w	Any alphabetic, numeric, or underscore character (For English character sets, this is equivalent to [a-zA-Z_0-9].)
\W	Any character that is not alphabetic, numeric, or underscore (For English character sets, this is equivalent to [^a-zA-Z_0-9].)
\d	Any numeric digit; equivalent to [0-9]
\D	Any nondigit character; equivalent to [^0-9]
\oN or \o{N}	Character of octal value N
\xN or \x{N}	Character of hexadecimal value N

Character Representation

Operator	Usage
`\\`	Backslash
`\a`	Alarm (beep)
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\char`	If a character has special meaning in a regular expression, precede it with backslash (`\`) to match it literally.

Grouping Operators

Operator	Usage
`(expr)`	Group regular expressions and capture tokens.
`(?:expr)`	Group regular expressions, but do not capture tokens.
`(?>expr)`	Group atomically.
$expr_1$`|`$expr_2$	Match expression $expr_1$ or expression $expr_2$.

Nonmatching Operators

Operator	Usage
`(?#comment)`	Insert a comment into the expression. Comments are ignored in matching.

Positional Operators

Operator	Usage
`^expr`	Match `expr` if it occurs at the beginning of the input string.

`expr$`	Match `expr` if it occurs at the end of the input string.
`\<expr`	Match `expr` when it occurs at the beginning of a word.
`expr\>`	Match `expr` when it occurs at the end of a word.
`\<expr\>`	Match `expr` when it represents the entire word.

Lookaround Operators

Operator	Usage
`(?=expr)`	Look ahead from current position and test if `expr` is found.
`(?!expr)`	Look ahead from current position and test if `expr` is not found
`(?<=expr)`	Look behind from current position and test if `expr` is found.
`(?<!expr)`	Look behind from current position and test if `expr` is not found.

Quantifiers

Operator	Usage
`expr{m,n}`	Match `expr` when it occurs at least `m` times but no more than `n` times consecutively.
`expr{m,}`	Match `expr` when it occurs at least `m` times consecutively.
`expr{n}`	Match `expr` when it occurs exactly `n` times consecutively. Equivalent to `{n,n}`.
`expr?`	Match `expr` when it occurs 0 times or 1 time. Equivalent to `{0,1}`.
`expr*`	Match `expr` when it occurs 0 or more times consecutively. Equivalent to `{0,}`.
`expr+`	Match `expr` when it occurs 1 or more times consecutively. Equivalent to `{1,}`.

`q_expr`	Match as much of the quantified expression as possible, where `q_expr` represents any of the expressions shown in the first six rows of this table.
`q_expr+`	Match as much of the quantified expression as possible, but do not rescan any portions of the string if the initial match fails.
`q_expr?`	Match only as much of the quantified expression as necessary.

Ordinal Token Operators

Operator	Usage
`(expr)`	Capture in a token all characters matched by the expression within the parentheses.
`\N`	Match the `N`th token generated by this command. That is, use `\1` to match the first token, `\2` to match the second, and so on.
`$N`	Insert the match for the `N`th token in the replacement string. If `N` is equal to zero, then insert the entire match in the replacement string. (Used only by the `regexprep` function.)
`(?(N)s1|s2)`	If `N`th token is found, then match `s1`; otherwise, match `s2`.

Named Token Operators

Operator	Usage
`(?<name>expr)`	Capture in a token all characters matched by the expression within the parentheses. Assign a `name` value to the token.
`\k<name>`	Match the token referred to by `name`.
`$<name>`	Insert the match for named token in a replacement string. (Used only with the `regexprep` function.)
`(?(name)s1|s2)`	If named token is found, then match `s1`; otherwise, match `s2`.

Conditional Expression Operators

Operator	Usage
`(?(cond)expr)`	If condition `cond` is `true`, then match expression `expr`.
`(?(cond)expr`$_1$`|expr`$_2$`)`	If condition `cond` is `true`, then match expression `expr`$_1$. Otherwise match expression `expr`$_2$.

Dynamic Expression Operators

Operator	Usage
`(??expr)`	Parse `expr` as a separate regular expression, and include the resulting string in the match expression. This gives you the same results as if you called `regexprep` inside of a `regexp` match expression.
`(??@cmd)`	Execute the MATLAB command represented by `cmd`, and include the string returned by the command in the match expression. This is a combination of the two dynamic syntaxes shown previously: `(??expr)` and `(?@cmd)`.
`(?@cmd)`	Execute the MATLAB command represented by `cmd` and discard any output the command returns. (Helpful for diagnosing regular expressions).
`${cmd}`	Execute the MATLAB command represented by `cmd`, and include the string returned by the command in the replacement expression.

Replacement String Operators

Operator	Usage
Operators from Character Representation table	The character represented by the operator sequence
`$'`	That part of the input string that precedes the current match
`$&` or `$0`	That part of the input string that is currently a match
`$'`	That part of the input string that follows the current match (In MATLAB, use `$''` to represent the character sequence `$'`.)
`$N`	The string represented by the token identified by `name`
`$<name>`	The string represented by the token identified by `name`
`${cmd}`	The string returned when MATLAB executes the command `cmd`

Return Value Summary

Qualifier	Description	Default Order
start	Row vector containing the starting index of each substring of `str` that matches `expr`	1
end	Row vector containing the ending index of each substring of `str` that matches `expr`	2
tokenExtents	Cell array containing the starting and ending indices of each substring of `str` that matches a token in `expr` (This is a `double` array when used with `'once'`.)	3
match	Cell array containing the text of each substring of `str` that matches `expr` (This is a string when used with `'once'`.)	4
tokens	Cell array of cell arrays of strings containing the text of each token captured by `regexp` (This is a cell array of strings when used with `'once'`.)	5
names	Structure array containing the name and text of each *named* token captured by `regexp` (If there are no named tokens in `expr`, `regexp` returns a structure array with no fields.) Field names of the returned structure are set to the token names, and field values are the text of those tokens. Named tokens are generated by the expression `(?<tokenname>)`.	6
split	Cell array containing those parts of the input string that are delimited by substrings returned when using the `regexp 'match'` option	7

---

[1] The term "word characters" in this text refers to characters that are alphabetic, numeric, or underscore.

Back to Top