cfRegex v0.3 Documentation

cfRegex is a project to provide a complete set of regex functionality for CFML.

See the file **readme.md** for project information, requirements, licensing and credits.

This documentation aims to provide everything you need to know to use cfRegex - please raise an issue if you find any bugs/errors/omissions, or have any questions.

Hyperlinks that are internal to the PDF are blue and dashed, those to external websites are red and underlined.

cfRegex Documentation

These is no enforced order to this documentation, so feel free to jump around reading pages in whatever way you find comfortable, however if you're not experienced with regex then the Regex Features section is a good starting point - particularly since it will help the examples used in the rest of the documentation make sense.

If you know regex already, you may want to simply start with reading through how to use the cfregex tag and create objects with cfRegex, before moving on to the Actions and Features sections to learn what functionality is available - however don't skip the other sections entirely, as they still contain a few things even frequent regex users do not know!

Installation

Installing cfRegex is not required - You can use Regex.cfc as a standard component in the same way as any other CFC - i.e. it can be created as `new Regex(...)` by scripts in the same directory or via a server/application mapped path.

However, if you want cfRegex to be available globally for all applications without a mapping, you need to install it by copying files to relevant locations.

Previous releases had individual packages for each CFML engine, but from v0.3 there are only two packages: one which supports Lucee, and a legacy package which works on otherwise discontinued CFML engines (OpenBD, Railo, ColdFusion 9)

NOTE: cfRegex has *not* been tested against Adobe ColdFusion 2016 or newer - it might work, or it might not. If this is important to you, get in touch to discuss options.

Lucee / Railo

Manual Setup Instructions

1. Download cfRegex v0.3 and extract to a temporary location.
2. Locate your `{lucee-server}/context/` directory (e.g. `{lucee-root}/lib/ext/lucee-server/context`). *
3. Copy `Regex.cfc` to `{lucee-server}/context/components/` directory.
4. Copy `Regex.cfc` to `{lucee-server}/context/library/tag/` directory.
5. Copy `functions/Regex*.cfm` to `{lucee-server}/context/library/function/` directory.
6. Restart the Lucee server.

You can now use the cfregex tag, create Regex objects, and use all the Regex~ functions as if they were built-in functions.

*For per-context installation (or, if you do not have a {lucee-server}/context/components directory), in steps 3..5 above use the `{lucee-web}` directory, which defaults to `{webroot}/WEB-INF/lucee`

(For Railo installation, read "railo" for every instance of "lucee" above.)

ColdFusion 9.0.1 and above

Tag and Object - Manual Setup

1. Download cfRegex v0.3 legacy and extract to a temporary location.
2. Locate your ColdFusion root directory, referred to as `{coldfusion-root}` below.
3. Copy `Regex.cfc` to `{coldfusion-root}/CustomTags/`
4. Copy `Regex.cfc` and `regex.cfm` to `{coldfusion-root}/wwwroot/WEB-INF/cftags/`

You can now use the cfregex tag and create Regex objects.

Open BlueDragon 2.0

Tag and Object - Manual Setup

1. Download cfRegex v0.3 legacy and extract to a temporary location.
2. Locate your OpenBD WEB-INF directory (e.g. `{openbd-root}/webapps/openbd/WEB-INF`).
3. Copy `Regex.cfc` and `regex.cfm` to `{openbd-root}/webapps/openbd/WEB-INF/CustomTags/`

You can now use the cf_regex tag and create Regex objects.

Functions - Manual Setup (ColdFusion and Open BlueDragon)

This process is the same for both ColdFusion and Open BlueDragon - the only difference is the location of the CustomTags directory.

For ColdFusion use `{coldfusion-root}/CustomTags` and for Open BlueDragon use `{openbd-root}/webapps/openbd/WEB-INF/CustomTags`

1. Locate the CustomTags directory create a `cfregex-functions` directory.
2. Copy `functions/*.cfm` to `{coldfusion-root}/CustomTags/cfregex-functions/`
3. Create a mapping `/cfregex-functions` to this directory.
4. At the start of every request, do:

```
<cfinclude template="/cfregex-functions/include-all.cfm" />
```

Now you can also use the functions!

Regex Object

Whenever any regex action is performed, there is a required step of processing the textual regex pattern and producing the 'machine' that actually applies the instructions provided by the pattern.

If the regex is a single use one, this is simply an unavoidable step, but when a regex is going to be used more than once, it is inefficient to repeat this identical event multiple times.

To help solve this problem, cfRegex introduces a Regex Object, allowing the regex pattern and modes to be compiled once into a variable, which can then be re-used as many times as necessary, with different actions, input text, and/or parameters.

This can allow both performance improvements and greater readability.

Creating Regex Objects

There are multiple ways to create a regex object. Typically, this can be done using the `new` keyword, like so:

```
<cfset MyRegex = new Regex( pattern , modes ) />
```

Or also with the RegexCompile function, which works similarly:

```
<cfset MyRegex = RegexCompile( pattern , modes ) />
```

When compiling a more complicated regex pattern, you can use the cfregex tag syntax, which is especially suited for splitting long regex across multiple lines:

```
<cfregex name="MyRegex" modes="modes">
    pattern
</cfregex>
```

For all three of these examples, only pattern is required - the modes are optional. For the tag syntax, the comment mode is on by default, which means whitespace is ignored (except in some situations, see the documentation on comment mode for full details).

Using Regex Objects

The use of a Regex Object is the same as with any other object, as in:

```
<cfset Result = MyRegex.action( Arguments ) />
```

Where `action` is any of the methods on the object, as listed on the Actions page.

cfregex Tag

The cfregex tag provides a convenient way to use long or complicated regex, by allowing the regex pattern to be freely spaced and interspersed with comments, which makes the expression far more understandable and maintainable.

The tag can be used either to compile a Regex Object, or for any of the actions which are available.

Creating an Object

To compile a regex object, simply use the tag with the `name` attribute to indicate the variable name to hold the created object:

```
<cfregex name="MyRx">
    ...pattern...
</cfregex>
```

You can optionally specify modes to apply:

```
<cfregex name="MyRx" modes="CASE_INSENSITIVE,DOTALL">
    ...pattern...
</cfregex>
```

Executing an Action

If you do not need a re-usable object, you can call actions directly, using the `variable` attribute to hold the result, with both of the following syntaxes being supported:

```
<cfregex action="match" variable="MatchResults" ...other parameters... >
    ...pattern...
</cfregex>
```

Or:

```
<cfregex match variable="MatchResults" ...other parameters... >
    ...pattern...
</cfregex>
```

The two different syntaxes are purely stylistic - they both work in exactly the same way.

Note also that whilst `variable` is used instead of `name` here, this is simply a matter of following existing CFML conventions; the two attributes are actually interchangable, and either one can be used in any situation.

All other parameters work in the same way as they do for functions or object-form (as described on the individual action's documentation page), and can either be supplied directly or by using `attributecollection`.

Actions

What is an Action?

In the cfRegex project, the term "action" is used to represent a trinity of available implementations for functionality: a function, an object method, and a tag action.

Although, technically, all actions have been implemented using the cffunction tag, for clarity only the Regex~ functions are referred to as "functions", any actions used via the Regex Object are referred to as "methods", or tag-form when using cfregex.

So, if you see "the X action" references, this refers to *all* implementations of X, whilst "the X function" is only applying to the function (but not the Object method nor tag-forms).

List of Actions

- Compile creates a Regex Object for when a regex is needed multiple times.
- Escape returns a literal representation of the regex pattern, allowing use within another regex.
- Find returns the character positions at which the regex is found.
- Match returns an array containing each regex match.
- Matches returns a boolean indicating if the regex exactly matches the text.
- Quote returns a quoted version of the regex, for use in another regex pattern.
- Replace returns the input text with any matches replaced by a string or function result.
- Split returns an array after treating the regex as a delimiter against the input text.

Compile

The `compile` action is used to create a Regex Object, which is worth doing when you will be using the same regex multiple times against different input text, as it avoids the need to repeatedly parse and compile the expression.

The compile action is called explicitly with the `RegexCompile` function, or when using `cfregex` tag with compile action, and of course is also used when creating a new Regex object, either with the `new` keyword, or via the `createObject` syntax.

Object (via `new` keyword)

Arguments

Name	Type	Required	Default	Notes
Pattern	RegexString	yes	n/a	The regex pattern to compile into a Regex Object.
Modes	StringList	no	none	List of regex modes to apply to the pattern.

Usage Examples

```
<cfset OneWordRx = new Regex( '\w+' ) />
```

```
<cfset LettersRx = new Regex( '[a-z]' , 'CASE_INSENSITIVE' ) />
```

```
<cfset StartlineLettersRx = new Regex( '^[a-z]' , 'CASE_INSENSITIVE,MULTILINE' ) />
```

Tag

Compile is the default `action` for the cfregex tag. This means that you do not need to explicitly specify it (but you can if you wish).

If you do not specify a variable name, the object created is called `cfregex`, but this is not recommended, especially since it will not be var-scoped in functions.

Attributes

Name	Type	Required	Default	Notes
Name	VarName	no	"cfregex"	A variable name to hold the compiled object.
Modes	StringList	no	none	List of regex modes to apply to the pattern.

Usage Examples

```
<cfregex name="OneWordRx">
    \w+
</cfregex>
```

```
<cfregex name="LettersRx" modes="case_insensitive">
    [a-z]
</cfregex>
```

```
<cfregex name="StartlineLettersRx" modes="case_insensitive,multiline">
    ^[a-z]
</cfregex>
```

Function

The `RegexCompile` function is included for consistency, but does not offer any advantages over the `new` keyword.

Arguments

Name	Type	Required	Default	Notes
Pattern	RegexString	yes	n/a	The regex pattern to compile into a Regex Object.
Modes	StringList	no	none	List of regex modes to apply to the pattern.

Usage Examples

```
<cfset OneWordRx = RegexCompile( '\w+' ) />
```

```
<cfset LettersRx = RegexCompile( '[a-z]' , 'CASE_INSENSITIVE' ) />
```

```
<cfset StartlineLettersRx = RegexCompile( '^[a-z]' , 'CASE_INSENSITIVE,MULTILINE' ) />
```

Practical Examples

Example 1

Creates a regex that can be used to ensure a directory starts with an appropriate root prefix, and is not blank nor a relative path:

```
<cfset PrefixRootRx = new Regex('^(?:$|\./)') />
<cfset RootDir = "/path/to/root" />

<cfloop item="CurDir" collection=#LotsOfDirs#>
    <cfset LotsOfDirs[CurDir] = PrefixRootRx.replace(LotsOfDirs[CurDir],RootDir) />
</cfloop>
```

Example 2

The key benefit of the cfregex tag is its ability to hold long regex patterns, without needing to worry about escaping quotes, and with the ability to neatly space out different parts and also to apply comments:

```
<cfregex name="CheckDocTypeRx" >
    ## HTML5 DocType
        (?i:<!doctype\ html\s*>)

    |

    ## XHTML
        <!DOCTYPE\s+html\s+PUBLIC\s+"-//W3C//DTD\ XHTML\ 1\.
            (?:
                ## 1.0 Strict or Traditional
                0\ (Strict|Transitional)//EN"
                \s+
                "http://www\.w3\.org/TR/xhtml1/DTD/xhtml1-(?i:\1)
            |
                ## XHTML 1.1
                1//EN"
                \s+
                "http://www\.w3\.org/TR/xhtml11/DTD/xhtml11
            )
        \.dtd"\s*>

    |

    ## HTML 4 Strict
        <!DOCTYPE\s+HTML\s+PUBLIC\s+"-//W3C//DTD\ HTML\ 4\.01//EN"
        \s+
        "http://www\.w3\.org/TR/html4/strict\.dtd"
        \s*>
</cfregex>

<cfdirectory
    name      = "HtmlFiles"
    directory = "./files-to-check"
    filter    = "*.html"
/>

<cfloop query="HtmlFiles">
    <cfset Filename = HtmlFiles.Directory * HtmlFiles.Name />

    <cfif NOT CheckDocTypeRx.matches( FileRead(Filename) , 'start' ) >
        <cfoutput>
            <li>Invalid doctype "#Filename#"
        </cfoutput>
    </cfif>
</cfloop>
```

You can use either Regex comments (as above), or CFML comments:

```
<cfregex name="FindColoursRx" >
    <!--- #888888 --->
    \#[A-F0-9]{6}(?=\s*[;"'}])
    |
    <!--- #888 --->
    \#[A-F0-9]{3}(?=\s*[;"'}])
    |
    <!--- rgb(128,128,128) and rgba(128,128,128,0.5) --->
    \brgba?\s*\([^)]+\)(?=\s*[;"'}])
    |
    <!--- hsl(128,50%,50%) and hsla(128,50%,50%,0.5) --->
    \bhsla?\s*\([^)]+\)(?=\s*[;"'}])
</cfregex>

<cfdirectory
    name      = "HtmlFiles"
    directory = "./files-to-check"
    filter    = "*.html"
/>
<cfset AllColoursFound = [] />

<cfloop query="HtmlFiles">
    <cfset Filename = HtmlFiles.Directory * HtmlFiles.Name />

    <cfset ColoursFound = FindColoursRx.match( FileRead(Filename) ) >

    <cfset AllColoursFound.addAll( ColoursFound ) />
</cfloop>
```

Escape

The `escape` action is used to convert a regex into a string that can be used to match as text within another regex pattern - that is, it escapes any regex metacharacters so as to produce a literal representation of the input text.

It is also possible to set the `ReturnType` to "class" if the text is to be used inside a character class, where different escaping rules apply. (Specifically, any hyphen characters "-" must be escaped, whilst many other characters do not need to be escaped.)

When escaping for a character class, duplicate characters are removed, and if tab or newline characters exist they are converted to `\t` and `\n` respectively.

Whilst the `escape` action exists in all three forms (object,tag,function), the RegexEscape function is the most useful of these, as it does not require a valid regex pattern as input (which both object and tag do), but can escape any arbitrary text - and it is the easy to nest within a string (unlike the tag).

See also Quote, for similar functionality.

Object

Although it is possible to call `escape` for a regex object, it is rare that you will need to do this. The RegexEscape function is usually recommended instead.

Arguments

Name	Type	Required	Default	Notes
ReturnType	Enum (regex,class)	no	regex	Determines if the regex should be escaped for use in a standard regex, or as part of a character class.

Usage Examples

```
<cfset ExampleRx = new Regex('^\w+\s{2,}-\s+\w*') />
```

```
<cfdump var=#ExampleRx.escape()# />
```

string | \^\\w\+\\s\{2,\}-\\s\+\\w*

```
<cfdump var=#ExampleRx.escape('class')# />
```

string | \^{2,}\-s+\\w*

Tag

Since the cfregex tag always treats its body content as a regex, the contents must currently be a valid regex pattern, otherwise an error is thrown - even when using the `escape` action.

To convert arbitrary text that is not a regex pattern, use the RegexEscape function instead. (If necessary, in combination with cfsavecontent.)

Attributes

Name	Type	Required	Default	Notes
Variable	VarName	no	"cfregex"	The variable which the escaped regex is assigned to.
ReturnType	Enum (regex,class)	no	regex	Determines if the regex should be escaped for use in a standard regex, or as part of a character class.
Modes	StringList	no	none	If modes are provided, escaping is performed relevant to this (e.g. # is a metacharacter only when comment mode is enabled.)

Usage Examples

```
<cfregex escape variable="RegexEscaped" >
    ^\w+
    \s{2,} - \s+
    \w*
</cfregex>
<cfdump var=#RegexEscaped# />
```

string | \^\\w\+\\s\{2,\}-\\s\+\\w*

```
<cfregex escape returntype="class" variable="RegexEscaped" >
    ^\w+
    \s{2,} - \s+
    \w*
</cfregex>
<cfdump var=#RegexEscaped# />
```

string | \^{2,}\-s+\\w*

Function

The RegexEscape function accepts the argument `Text`, not `Pattern`, since it does not compile to a regex pattern.

This means it is possible to use RegexEscape against a non-regex piece of text and make it safe for use in regex. (This is not possible using the cfregex tag with action `escape`, since the tag always treats its contents as a regex pattern, so would throw a syntax error.)

Arguments

Name	Type	Required	Default	Notes
Text	String	yes	n/a	
ReturnType	Enum (regex,class)	no	regex	Determines if the regex should be escaped for use in a standard regex, or as part of a character class.
Modes	StringList	no	none	If modes are provided, escaping is performed relevant to this (e.g. # is a metacharacter only when comment mode is enabled.)

Usage Examples

```
<cfdump var=#RegexEscape( '^\w+\s{2,}-\s+\w*' )# />
```

string | \^\\w\+\\s\{2,\}-\\s\+\\w*

```
<cfdump var=#RegexEscape( '^\w+\s{2,}-\s+\w*' , 'class' )# />
```

string | \^{2,}\-s+\\w*

```
<cfdump var=#RegexEscape( '*\o/* :)' )# />
```

string | *\\o/* :\)

Practical Examples

Example 1

Escaping text which may contain regex symbols but should not be treated as such.

```
<cfset MarkdownText = "this is [markdown](http://daringfireball.net/projects/markdown/), *not* a regex." />
<cfoutput>#RegexEscape(MarkdownText)#</cfoutput>
```

Example 2

Performing a whole-word or phrase match with unknown user input:

```
<cfset Results = RegexFind( '\b#RegexEscape(Url.SearchWord)#\b' , SearchText ) />
```

Example 3

Allowing user-supplied characters to exclude as part of a larger expression, without (deliberately or accidentally) changing the behaviour of the regex:

```
<cfset ExcludedCharacters = RegexEscape(Form.CharactersToExclude,'class') />
<cfset BigRx = new Regex('...[^#ExcludedCharacters#]...') />
```

Find

The `find` action is similar to the core `refind` function of CFML, but much more flexible. You are not forced to call the function multiple times to find multiple matches - you simply use the `Limit` argument if you need to stop after a certain number of matches have been made.

If the expression is not found, the result is always an empty array. If it does find the expression, the results depend on the `returntype` attribute, but will always be an array with each element referring to each of the matches.

If you are simply checking whether a regex matches or not, and do not need to know the information about which character position it matches at, use the Matches action, which is more efficient and returns a boolean true/false.

If you only need to know the text of a match (not it's position), you can use the Match action, which can return a simple array of strings.

Return Types

There are three possible structures which the `find` action can return, which one to use depends on how much information you need.

For all three return types, if no match is found, an empty array is returned.

pos

This returns a simple array of the character positions at the start of each match. This is the default value is a return type is not set.

sub

This returns an array of structures, with each structure containing two keys, `pos` and `len`, indiciating the character position and the length of the match. Each structure is an array, the first element of which is the overall match, whilst the rest of the elements relate to the groups.

(This is akin to setting `returnsubexpressions` to true with `refind`.)

info

This returns an array of structures, containing complete information for each match. Keys `pos`, `len` and `match` return character position, length of match, and the match text respectively. Key `group` contains an array, each array element is a structure representing all groups found, with the same `pos`, `len` and `match` keys.

Object

Arguments

Name	Type	Required	Default	Notes
Text	String	yes	n/a	The text to find the regex within.
Start	Char Position	no	1	Position at which to start trying to find the regex. (1 is first character.)
Limit	Integer	no	0	Number of times to find the regex before stopping. (0 is unlimited.)
ReturnType	Enum (pos,sub,info)	no	"pos"	Determines the structure of each array element in the return variable.

Usage Examples

```
<cfset Input = "The quick fox jumps over the lazy brown dog." />
<cfset ThreeLettersRx = new Regex('\b\w(\w)(\w)\b') />
<cfset FiveLettersRx = new Regex('\b\w(\w{4})\b') />
```

```
<cfdump var=#ThreeLettersRx.find( Input )# />
```

Array
1 number 1
2 number 11
3 number 26
4 number 41

```
<cfdump var=#FiveLettersRx.find( Input )# />
```

Array
1 number 5
2 number 15
3 number 35

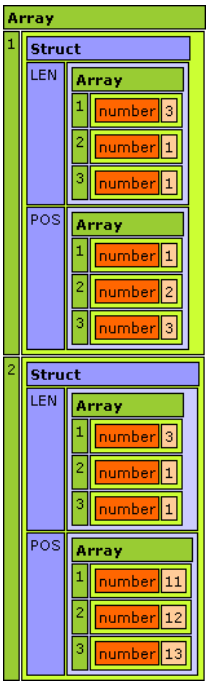
```
<cfdump var=#ThreeLettersRx.find( Input , 5 , 1 )# />
```

Array
1 number 11

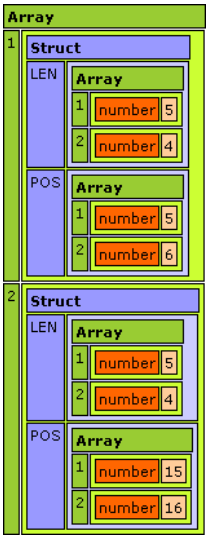
```
<cfdump var=#FiveLettersRx.find( Input , 5 , 1 )# />
```

Array
1 number 5

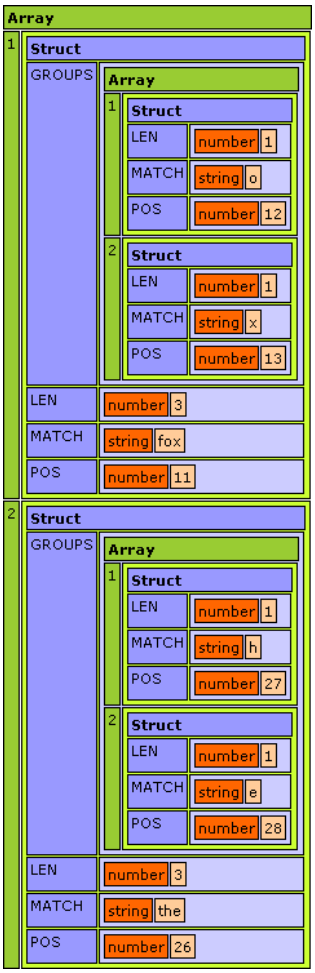
```
<cfdump var=#ThreeLettersRx.find( text=Input , limit = 2 , returntype='sub' )# />
```



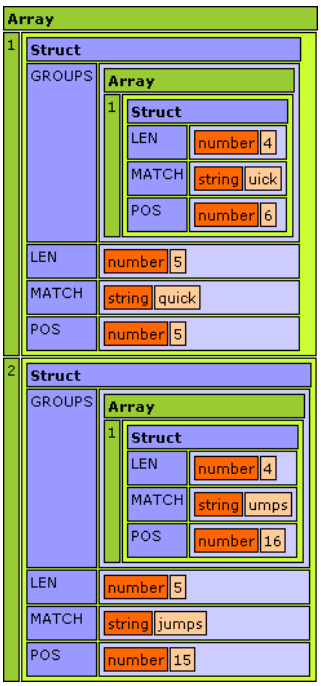

```
<cfdump var=#FiveLettersRx.find( text=Input , limit = 2 , returntype='sub' )# />
```




```
<cfdump var=#ThreeLettersRx.find( Input , 5 , 2 , 'info' )# />
```




```
<cfdump var=#FiveLettersRx.find( Input , 5 , 2 , 'info' )# />
```



Tag

Attributes

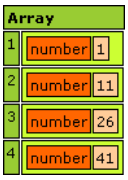
Name	Type	Required	Default	Notes
Variable	VarName	no	"cfregex"	The variable which the array is assigned to.
Text	String	yes	n/a	The text to find the regex within.
Start	Char Position	no	1	Position at which to start trying to find the regex. (1 is first character.)
Limit	Integer	no	0	Number of times to find the regex before stopping. (0 is unlimited.)
ReturnType	Enum (pos,sub,info)	no	"pos"	Determines the structure of each array element in the return variable.

Name	Type	Required	Default	Notes
Modes	StringList	no	none	List of regex modes to apply to the pattern.

Usage Examples

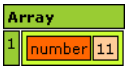
```
<cfset Input = "The quick fox jumps over the lazy brown dog." />
```

```
<cfregex find variable="WordPositions" text=#Input#>
    \b\w(\w)(\w)\b
</cfregex>
<dump var=#WordPositions#/>
```

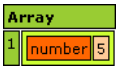



```
<cfregex find variable="WordPositions" text=#Input#>
    \b\w(\w{4})\b
</cfregex>
<dump var=#WordPositions#/>
```

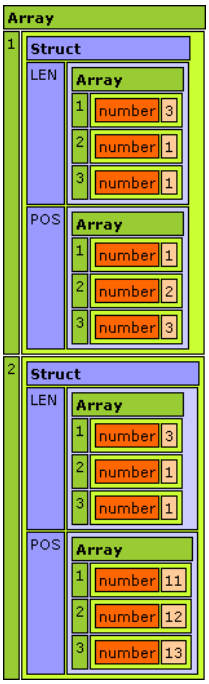

```
<cfregex find variable="WordPositions" text=#Input# start=5 limit=1 >
    \b\w(\w)(\w)\b
</cfregex>
<dump var=#WordPositions#/>
```



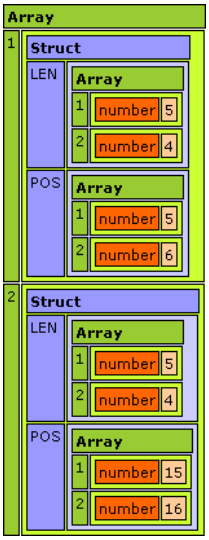

```
<cfregex find variable="WordPositions" text=#Input# start=5 limit=1 >
    \b\w(\w{4})\b
</cfregex>
<dump var=#WordPositions#/>
```



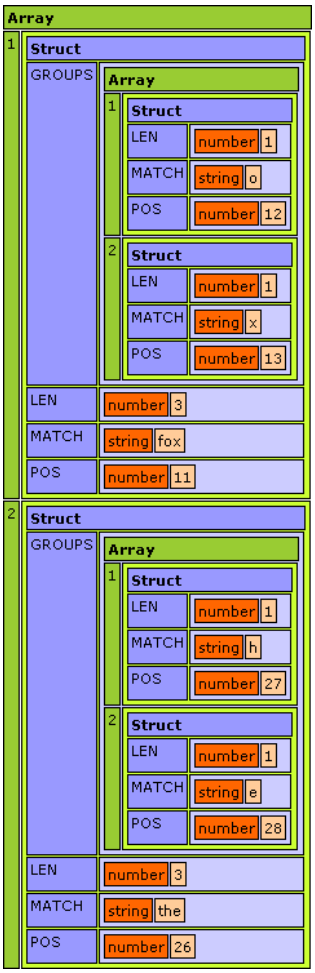

```
<cfregex find variable="WordPositions" text=#Input# limit=2 returntype="sub" >
    \b\w(\w)(\w)\b
</cfregex>
<dump var=#WordPositions#/>
```



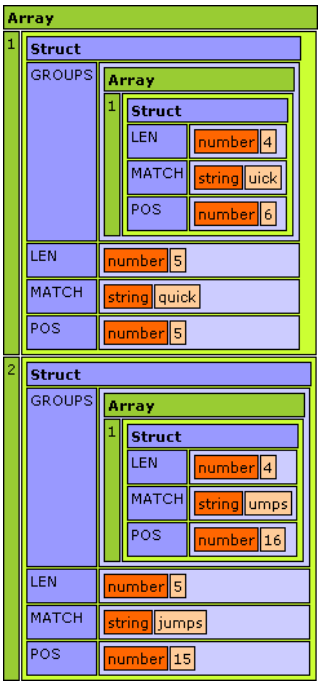

```
<cfregex find variable="WordPositions" text=#Input# limit=2 returntype="sub" >
    \b\w(\w{4})\b
</cfregex>
<dump var=#WordPositions#/>
```




```
<cfregex find variable="WordPositions" text=#Input# start=5 limit=2 returntype="info" >
    \b\w(\w)(\w)\b
</cfregex>
<dump var=#WordPositions#/>
```




```
<cfregex find variable="WordPositions" text=#Input# start=5 limit=2 returntype="info" >
    \b\w(\w{4})\b
</cfregex>
<dump var=#WordPositions#/>
```



Function

Arguments

Name	Type	Required	Default	Notes
Pattern	RegexString	yes	n/a	The regex pattern to compile into a Regex Object.
Text	String	yes	n/a	The text to find the regex within.
Start	Char Position	no	1	Position at which to start trying to find the regex. (1 is first character.)

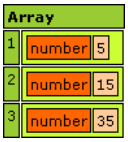
Name	Type	Required	Default	Notes
Limit	Integer	no	0	Number of times to find the regex before stopping. (0 is unlimited.)
ReturnType	Enum (pos,sub,info)	no	"pos"	Determines the structure of each array element in the return variable.
Modes	StringList	no	none	List of regex modes to apply to the pattern.

Usage Examples

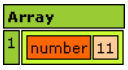
```
<cfset Input = "The quick fox jumps over the lazy brown dog." />
```

```
<cfdump var=#RegexFind( '\b\w(\w)(\w)\b' , Input )# />
```

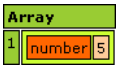

```
<cfdump var=#RegexFind( '\b\w(\w{4})\b' , Input )# />
```



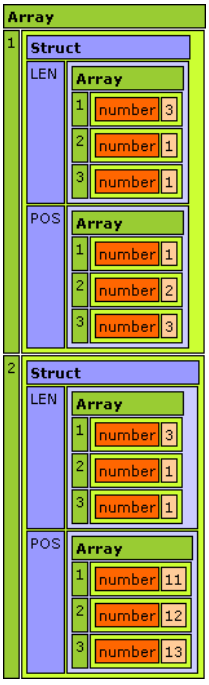

```
<cfdump var=#RegexFind( '\b\w(\w)(\w)\b' , Input , 5 , 1 )# />
```



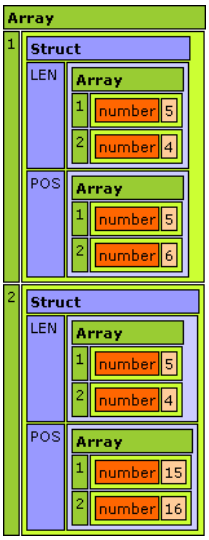

```
<cfdump var=#RegexFind( '\b\w(\w{4})\b' , Input , 5 , 1 )# />
```



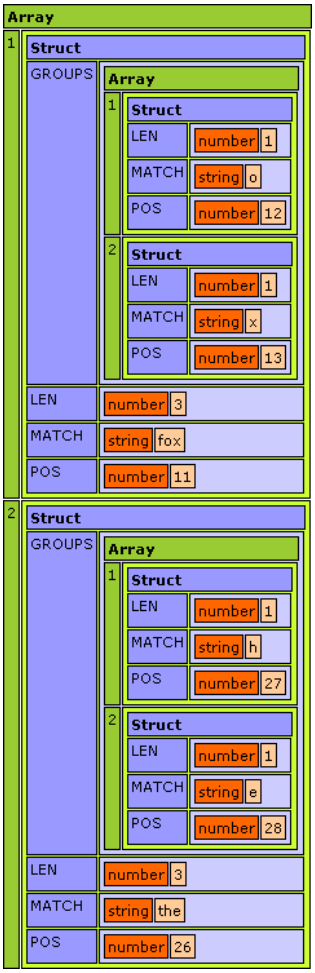

```
<cfdump var=#RegexFind( pattern='\b\w(\w)(\w)\b' , text=Input , limit = 2 , returntype='sub' )# />
```



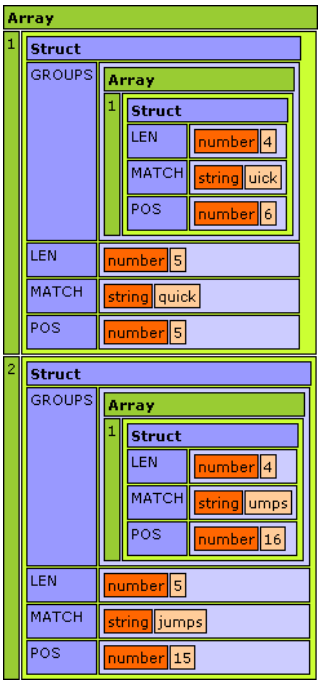

```
<cfdump var=#RegexFind( pattern='\b\w(\w{4})\b' , text=Input , limit = 2 , returntype='sub' )# />
```




```
<cfdump var=#RegexFind( '\b\w(\w)(\w)\b' , Input , 5 , 2 , 'info' )# />
```




```
<cfdump var=#RegexFind( '\b\w(\w{4})\b' , Input , 5 , 2 , 'info' )# />
```



Match

The `match` action returns an array of matches, similar to the core `rematch` function available in CF8 and compatible engines, but with far more options available.

For example, if you only need to know the first match, you can set `limit` to 1 and it will stop matching after that match (instead of having to return all matches and then discarding the rest of them).

Match also supports a callback function, which allows you to execute CFML logic against every match, and decide whether it should be included in the results. This allowing more efficient filtering - for example, combined with `limit` it might be used to find the first match which meets certain criteria, whilst only needing to keep matching until a suitable one is found.

A callback function can receive a variety of information about a match, (and also accepts an arbitrary structure passed in using `CallbackData`), for full details of how to use callbacks, see the dedicated Callback page.

When you need to do more than return the text for each match, you can set `ReturnType` to return an array of groups found, a structure of named groups, or all three of these combined. The section below goes into details on this.

Return Types

match

This returns a simple array of strings, containing the entire text matched by the regex. This is the default value is a return type is not set.

groups

This returns an array of arrays, which contain the text matched by each group captured by each match.

namedgroups

If you specify `namedgroups` for the `returntype`, you must also provide a list of names to be mapped to each group - this can either be a StringList or an Array of strings - and the result will then be an array of structs, with the numerical group matches mapped to the appropriate struct items.

If the number of groups provided exceeds the number of groups in the regex, the surplus groups are not included in the results. Similarly, if there are more groups than group names provided, only the ones named are returned.

If you need both numerical and named groups, use the `full` instead.

full

This combines the three returntypes above, to return an array of structs. Each struct has the keys `match`, `groups`, and `namedgroups` (only if optional `groupnames` is provided).

Object

Arguments

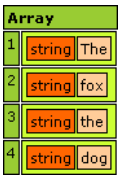
Name	Type	Required	Default	Notes
Text	String	yes	n/a	The text to match the regex against.
Start	Char Position	no	1	Position at which to start attempting to match. (1 is first character.)
Limit	Integer	no	0	Number of times to match before stopping. (0 is unlimited.)
ReturnType	Enum (match,groups, namedgroups,full)	no	"pos"	Determines the structure of each array element in the return variable.
GroupNames	StringList *or* Array	no*	none	An array of names to label groups with. *Required for ReturnType namedgroups, optional for ReturnType full, ignored for other ReturnTypes.

Name	Type	Required	Default	Notes
Callback	Function	no	none	A function called each time a match is made. If function returns false the match is excluded from results (and does not count towards limit). See Callbacks section for full details on function signature and how to use this feature.
CallbackData	Struct	no	none	A structure which is passed into the callback function.

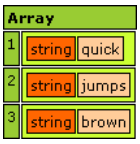
Usage Examples

```
<cfset Input = "The quick fox jumps over the lazy brown dog." />
<cfset ThreeLettersRx = new Regex('\b\w(\w)(\w)\b') />
<cfset FiveLettersRx = new Regex('\b\w(\w{4})\b') />
```

```
<cfdump var=#ThreeLettersRx.match( Input )# />
```



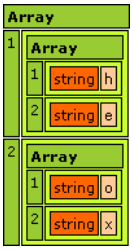

```
<cfdump var=#FiveLettersRx.match( Input )# />
```



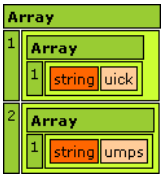

```
<cfdump var=#ThreeLettersRx.match( Input , 5 , 1 )# />
```


```
<cfdump var=#FiveLettersRx.match( Input , 5 , 1 )# />
```

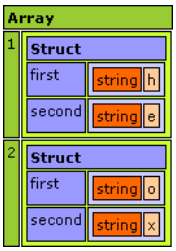

```
<cfdump var=#ThreeLettersRx.match( text=Input , limit = 2 , returntype='groups' )# />
```



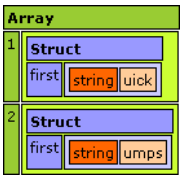

```
<cfdump var=#FiveLettersRx.match( text=Input , limit = 2 , returntype='groups' )# />
```



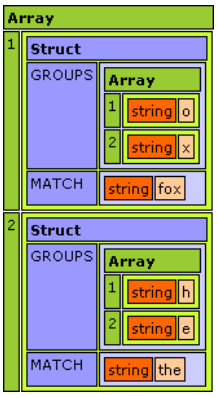

```
<cfdump var=#ThreeLettersRx.match( text=Input , limit = 2 , returntype='namedgroups' , groupnames='first,second' )# />
```



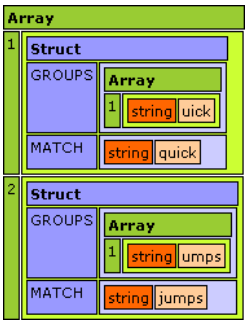

```
<cfdump var=#FiveLettersRx.match( text=Input , limit = 2 , returntype='namedgroups' , groupnames='first,second' )# />
```




```
<cfdump var=#ThreeLettersRx.match( Input , 5 , 2 , 'full' )# />
```




```
<cfdump var=#FiveLettersRx.match( Input , 5 , 2 , 'full' )# />
```



Tag

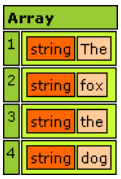
Attributes

Name	Type	Required	Default	Notes
Variable	VarName	no	"cfregex"	The variable which the result is assigned to.
Text	String	yes	n/a	The text to match the regex against.
Start	Char Position	no	1	Position at which to start attempting to match. (1 is first character.)
Limit	Integer	no	0	Number of times to match before stopping. (0 is unlimited.)
ReturnType	Enum (match,groups, namedgroups,full)	no	"pos"	Determines the structure of each array element in the return variable.
GroupNames	StringList *or* Array	no*	none	An array of names to label groups with. *Required for ReturnType namedgroups, optional for ReturnType full, ignored for other ReturnTypes.
Callback	Function	no	none	A function called each time a match is made. If function returns false the match is excluded from results (and does not count towards limit). See Callbacks section for full details on function signature and how to use this feature.
CallbackData	Struct	no	none	A structure which is passed into the callback function.
Modes	StringList	no	none	List of regex modes to apply to the pattern.

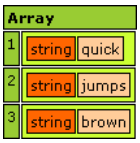
Usage Examples

```
<cfset Input = "The quick fox jumps over the lazy brown dog." />
```

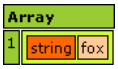
```
<cfregex match variable="ThreeLetterWords" text=#Input# >
    \b\w(\w)(\w)\b
</cfregex>
<cfdump var=#ThreeLetterWords# />
```




```
<cfregex match variable="FiveLetterWords" text=#Input# >
    \b\w(\w{4})\b
</cfregex>
<cfdump var=#FiveLetterWords# />
```

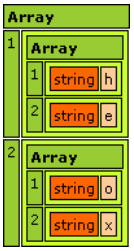



```
<cfregex match variable="FirstThreeLetterWordFrom5thChar" text=#Input# start=5 limit=1 >
    \b\w(\w)(\w)\b
</cfregex>
<cfdump var=#FirstThreeLetterWordFrom5thChar# />
```

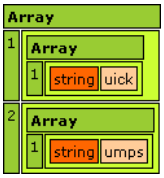



```
<cfregex match variable="FirstFiveLetterWordFrom5thChar" text=#Input# start=5 limit=1 >
    \b\w(\w{4})\b
</cfregex>
<cfdump var=#FirstFiveLetterWordFrom5thChar# />
```

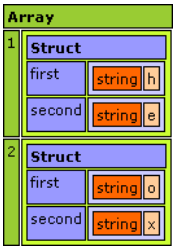

```
<cfregex match variable="ThreeLetterGroups" text=#Input# limit=2 returntype="groups" >
    \b\w(\w)(\w)\b
</cfregex>
<cfdump var=#ThreeLetterGroups# />
```



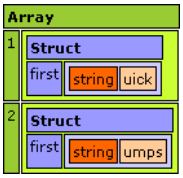

```
<cfregex match variable="FiveLetterGroups" text=#Input# limit=2 returntype="groups" >
    \b\w(\w{4})\b
</cfregex>
<cfdump var=#FiveLetterGroups# />
```



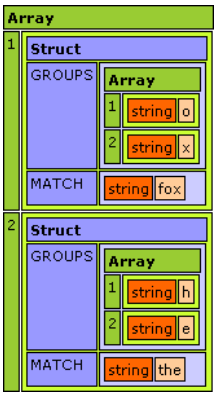
```
<cfregex match variable="ThreeLetterNamedGroups" text=#Input# limit=2 returntype="namedgroups" groupnames="first,second" >
    \b\w(\w)(\w)\b
</cfregex>
<cfdump var=#ThreeLetterNamedGroups# />
```



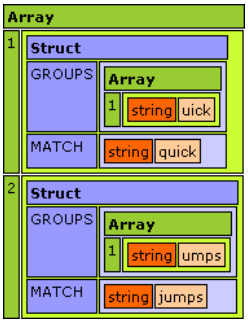

```
<cfregex match variable="FiveLetterNamedGroups" text=#Input# limit=2 returntype="namedgroups" groupnames="first,second" >
    \b\w(\w{4})\b
</cfregex>
<cfdump var=#FiveLetterNamedGroups# />
```




```
<cfregex match variable="ThreeLetterFullInfo" text=#Input# start=5 limit=2 returntype="full" >
    \b\w(\w)(\w)\b
</cfregex>
<cfdump var=#ThreeLetterFullInfo# />
```




```
<cfregex match variable="FiveLetterFullInfo" text=#Input# start=5 limit=2 returntype="full" >
    \b\w(\w{4})\b
</cfregex>
<cfdump var=#FiveLetterFullInfo# />
```



Function

Arguments

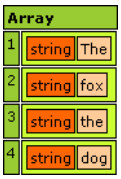
Name	Type	Required	Default	Notes
Pattern	RegexString	yes	n/a	The regex pattern to compile into a Regex Object.
Text	String	yes	n/a	The text to match the regex against.
Start	Char Position	no	1	Position at which to start attempting to match. (1 is first character.)
Limit	Integer	no	0	Number of times to match before stopping. (0 is unlimited.)

Name	Type	Required	Default	Notes
ReturnType	Enum (match,groups, namedgroups,full)	no	"pos"	Determines the structure of each array element in the return variable.
GroupNames	StringList *or* Array	no*	none	An array of names to label groups with. *Required for ReturnType namedgroups, optional for ReturnType full, ignored for other ReturnTypes.
Callback	Function	no	none	A function called each time a match is made. If function returns false the match is excluded from results (and does not count towards limit). See Callbacks section for full details on function signature and how to use this feature.
CallbackData	Struct	no	none	A structure which is passed into the callback function.
Modes	StringList	no	none	List of regex modes to apply to the pattern.

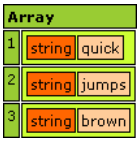
Usage Examples

```
<cfset Input = "The quick fox jumps over the lazy brown dog." />
```

```
<cfdump var=#RegexMatch( '\b\w(\w)(\w)\b' , Input )# />
```



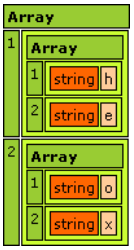

```
<cfdump var=#RegexMatch( '\b\w(\w{4})\b' , Input )# />
```



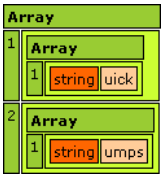

```
<cfdump var=#RegexMatch( '\b\w(\w)(\w)\b' , Input , 5 , 1 )# />
```


```
<cfdump var=#RegexMatch( '\b\w(\w{4})\b' , Input , 5 , 1 )# />
```

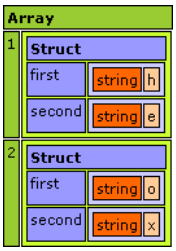

```
<cfdump var=#RegexMatch( pattern='\b\w(\w)(\w)\b' , text=Input , limit = 2 , returntype='groups' )# />
```



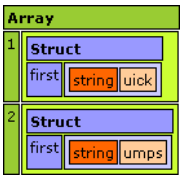

```
<cfdump var=#RegexMatch( '\b\w(\w{4})\b' , text=Input , limit = 2 , returntype='groups' )# />
```



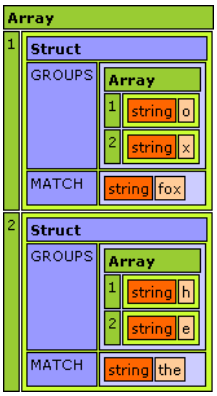

```
<cfdump var=#RegexMatch( pattern='\b\w(\w)(\w)\b' , text=Input , limit = 2 , returntype='namedgroups' , groupnames='first,second' )# />
```



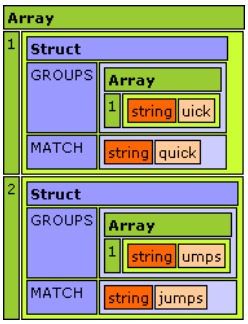

```
<cfdump var=#RegexMatch( pattern='\b\w(\w{4})\b' , text=Input , limit = 2 , returntype='namedgroups' , groupnames='first,second' )# />
```




```
<cfdump var=#RegexMatch( '\b\w(\w)(\w)\b' , Input , 5 , 2 , 'full' )# />
```




```
<cfdump var=#RegexMatch( '\b\w(\w{4})\b' , Input , 5 , 2 , 'full' )# />
```



Practical Examples

Whilst the Usage Examples above give examples of *how* the different options of Match can be used, this section gives practical situations to show *why* Match might be used.

Example 1

Obtain all IP addresses found in the input text:

```
<cfset Ip4Addresses = RegexMatch( '\b[12]?\d\d(?:\.[12]?\d\d){3}\b' ) />
```

Example 2

Locate "TODO" tasks in CFML files:

```
<cfset TodoRx = new Regex('<!---\s*TODO:[^-]++(?s:(?!--->).)*--->|//\s*TODO:[^\n]+') />
<cfset Todos = StructNew() />

<cfdirectory name="ProjFiles" recursive directory="/project" />
<cfloop query="ProjFiles">
    <cfif ProjFiles.Type NEQ 'File'><cfcontinue/></cfif>
    <cfset CurFilename = ProjFiles.Directory & ProjFiles.Name />

    <cfset Todos[CurFilename] = TodoRx.match( FileRead(CurFilename) ) />
</cfloop>
```

Example 3

Locate all CSS colour codes:

```
<cfregex match
    variable = "colours"
    input    = #CssCode#
    modes    = "case_insensitive"
    >
    <!--- #888888 --->
    \#[A-F0-9]{6}(?=\s*[;"'}])
    |
    <!--- #888 --->
    \#[A-F0-9]{3}(?=\s*[;"'}])
    |
    <!--- rgb(128,128,128) and rgba(128,128,128,0.5) --->
    \brgba?\s*\([^)]+\)(?=\s*[;"'}])
    |
    <!--- hsl(128,50%,50%) and hsla(128,50%,50%,0.5) --->
    \bhsla?\s*\([^)]+\)(?=\s*[;"'}])
</cfregex>
```

Example 4

Using `match` to extract function and argument code from a CFC. The entire match text is not required, only the text captured in the four groups, so returntype `groups` is used.

```
<cfregex
    action     = "match"
    variable   = "Funcs"
    text       = "#Arguments.ComponentCode#"
    returntype = "groups"
    >

    ## 1 :  space to hide tag from cfml compiler (unescaped whitespace is ignored)
    (< cffunction\ name=")

    ## 2 : match any name, excluding compile (handled separately)
    ((?!compile)[^"]++)

    ## 3 : remaining attribute text
    ("[^>]+)

    ## take unnecessary attributes out of group
    access="public"\ action>

    ## 4 : grab all the arguments (again, space to hide tag from compiler)
    ((?:
        \n\t+< cfargument.*?/>
    )*+)
</cfregex>
```

Matches

The `matches` action returns a boolean which indicates if the supplied regex matches the entire input text.

It also supports an optional `returntype` which allows different checks, as detailed below.

Return Types

exact

Perform an exact match and returns true only if the regex exactly matches the entire text.

partial

Return true is the regex matches anywhere within the text.

start

Return true only if the regex match starts at the first character of the text.

end

Return true only if the regex match ends with the last character of the text.

count

Returns the number of times a partial match is found anywhere within the text.

Object

Arguments

Name	Type	Required	Default	Notes
Text	String	yes	n/a	The text to check if the regex matches in.
ReturnType	Enum (exact,partial,start,end,count)	no	"exact"	See Return Types section for details.

Usage Examples

```
<cfset Input = "The quick fox jumps over the lazy brown dog." />
<cfset NineWordsRx = new Regex( '(?:\w+\W){9}' ) />
<cfset ThreeWordsRx = new Regex( '(?:\w+\W){3}' ) />
<cfset TtheRx = new Regex( '[Tt]he' ) />
```

```
<cfdump var=#NineWordsRx.matches( Input )# />
```

boolean true

```
<cfdump var=#ThreeWordsRx.matches( Input )# />
```

boolean false

```
<cfdump var=#ThreeWordsRx.matches( Input , 'partial' )# />
```

boolean true

```
<cfdump var=#ThreeWordsRx.matches( Input , 'count' )# />
```

number 3

```
<cfdump var=#TtheRx.matches( Input , 'partial' )# />
```

boolean true

```
<cfdump var=#TtheRx.matches( Input , 'count' )# />
```

number 2

```
<cfdump var=#TtheRx.matches( Input , 'start' )# />
```

boolean true

```
<cfdump var=#TtheRx.matches( Input , 'end' )# />
```

boolean false

```
<cfset DogRx = new Regex('dog\.') />
<cfdump var=#DogRx.matches( Input , 'end' )# />
```

boolean true

```
<cfset TheRx = new Regex('the') />
<cfdump var=#TheRx.matches( Input , 'start' )# />
```

boolean false

```
<cfset TheRx = new Regex('the','case_insensitive') />
<cfdump var=#TheRx.matches( Input , 'start' )# />
```

boolean true

Tag

Attributes

Name	Type	Required	Default	Notes
Variable	VarName	no	"cfregex"	The variable which the result is assigned to.
Text	String	yes	n/a	The text to check if the regex matches in.
ReturnType	Enum (exact,partial,start,end,count)	no	"exact"	See Return Types section for details.
Modes	StringList	no	none	List of regex modes to apply to the pattern.

Usage Examples

```
<cfset Input = "The quick fox jumps over the lazy brown dog." />
```

```
<cfregex matches variable="isExactMatch" text=#Input# >
    (?:\w+\W){9}
</cfregex>
<dump var=#isExactMatch#/>
```

boolean true

```
<cfregex matches variable="isExactMatch" text=#Input# >
    (?:\w+\W){3}
</cfregex>
<dump var=#isExactMatch#/>
```

boolean false

```
<cfregex matches variable="isPartialMatch" text=#Input# returntype="partial" >
    (?:\w+\W){3}
</cfregex>
<dump var=#isPartialMatch#/>
```

boolean true

```
<cfregex matches variable="MatchCount" text=#Input# returntype="count" >
    (?:\w+\W){3}
</cfregex>
<dump var=#MatchCount#/>
```

number 3

```
<cfregex matches variable="isPartialMatch" text=#Input# returntype="partial" >
    [Tt]he
</cfregex>
<dump var=#isPartialMatch#/>
```

boolean true

```
<cfregex matches variable="MatchCount" text=#Input# returntype="count" >
    [Tt]he
</cfregex>
<dump var=#MatchCount#/>
```

number 2

```
<cfregex matches variable="isStartMatch" text=#Input# returntype="start" >
    [Tt]he
</cfregex>
<dump var=#isStartMatch#/>
```

boolean true

```
<cfregex matches variable="isEndMatch" text=#Input# returntype="start" >
    [Tt]he
</cfregex>
<dump var=#isEndMatch#/>
```

boolean false

```
<cfregex matches variable="isEndMatch" text=#Input# returntype="start" >
    dog\.
</cfregex>
<dump var=#isEndMatch#/>
```

boolean true

```
<cfregex matches variable="isStartMatch" text=#Input# returntype="start" >
    the
</cfregex>
<dump var=#isStartMatch#/>
```

boolean false

```
<cfregex matches variable="isStartMatch" text=#Input# returntype="start" flags="case_insensitive" >
    the
</cfregex>
<dump var=#isStartMatch#/>
```

boolean true

Function

Arguments

Name	Type	Required	Default	Notes
Pattern	RegexString	yes	n/a	The regex pattern to compile into a Regex Object.
Text	String	yes	n/a	The text to check if the regex matches in.
ReturnType	Enum (exact,partial,start,end,count)	no	"exact"	See Return Types section for details.
Modes	StringList	no	none	List of regex modes to apply to the pattern.

Usage Examples

```
<cfset Input = "The quick fox jumps over the lazy brown dog." />
```

```
<cfdump var=#RegexMatches( '(?:\w+\W){9}' , Input )# />
```

boolean true

```
<cfdump var=#RegexMatches( '(?:\w+\W){3}' , Input )# />
```

boolean false

```
<cfdump var=#RegexMatches( '(?:\w+\W){3}' , Input , 'partial' )# />
```

boolean true

```
<cfdump var=#RegexMatches( '(?:\w+\W){3}' , Input , 'count' )# />
```

number 3

```
<cfdump var=#RegexMatches( '[Tt]he' , Input , 'partial' )# />
```

boolean true

```
<cfdump var=#RegexMatches( '[Tt]he' , Input , 'count' )# />
```

number 2

```
<cfdump var=#RegexMatches( '[Tt]he' , Input , 'start' )# />
```

boolean true

```
<cfdump var=#RegexMatches( '[Tt]he' , Input , 'end' )# />
```

boolean false

```
<cfdump var=#RegexMatches( 'dog\.' , Input , 'end' )# />
```

boolean true

```
<cfdump var=#RegexMatches( 'the' , Input , 'start' )# />
```

boolean false

```
<cfdump var=#RegexMatches( 'the' , Input , 'start' , 'case_insensitive' )# />
```

boolean true

Practical Examples

Example 1

Checking for a UK postcode:

```
<cfif NOT RegexMatches
    ( '[A-Z]{1,2}[0-9][A-Z]?[0-9]{1,2}[A-Z]{2}'
    , RegexReplace(Form.Postcode,'\s|-','')
    )>
    <cfset Error = "Unrecognised postcode." />
</cfif>
```

(This is a simplified regex, actual UK postcode validation is more complicated.)

Example 2

Validating that a password has enough non-alpha characters:

```
<cfif RegexMatches('\W',Form.Password,'count') LT 3>
    <cfset Error = "Passwords must contain at least three non-alphanumeric characters" />
</cfif>
```

Example 3

If you wanted to identify files which contained var-scoped variables, so that they can be converted to using the local scope, you might use:

```
<cfset FindVarRx = new Regex
    ( '<cfset\s+var\s+'
      & '|'
      & '<cfscript>[^<v]+(?:(?!</cfscript>).)*?(?<=\n\t{0,5})var\s+'
    , 'CASE_INSENSITIVE,DOTALL'
    ) />

<cfset VarFiles = StructNew() />
<cfset Total = 0 />
<cfdirectory name="Files" recurse directory="/codebase" filter="\*.cfm|\*.cfc" />

<cfloop query="Files">
    <cfif Files.Type NEQ 'FILE'><cfcontinue/></cfif>

    <cfset Found = FindVarRx.matches
        ( Text       : FileRead(Files.Directory & Files.Name)
        , ReturnType : 'count'
        ) />

    <cfif Found >
        <cfset VarFiles[Files.Directory & Files.Name] = Found />
        <cfset Total += Found />
    </cfif>
</cfloop>

<cfif Total >
    <cfoutput>Found #Total# Vars in #StructCount(VarFiles)# of #Files.RecordCount# files.</cfoutput>
    <cfdump var=#VarFiles# />
<cfelse>
    <cfoutput>No vars found in #Files.RecordCount# files.</cfoutput>
</cfif>
```

(note: this is to demonstrate how the method might be used; the regex itself may return false positives (e.g. vars inside comments or strings)

Example 4

The following example verifies that the start of a string matches one of five doctype definitions.

```
<cfregex
    action     = "matches"
    returntype = "start"
    variable   = "isSupportedDoctype"
    text       = #HtmlContent#
    >
    ## HTML5 DocType
        (?i:<!doctype\ html\s*>)

    |

    ## XHTML
        <!DOCTYPE\s+html\s+PUBLIC\s+"-//W3C//DTD\ XHTML\ 1\.
            (?:
                ## 1.0 Strict or Transitional
                0\ (Strict|Transitional)//EN"
                \s+
                "http://www\.w3\.org/TR/xhtml1/DTD/xhtml1-(?i:\1)
            |
                ## XHTML 1.1
                1//EN"
                \s+
                "http://www\.w3\.org/TR/xhtml11/DTD/xhtml11
            )
        \.dtd"\s*>

    |

    ## HTML 4 Strict
        <!DOCTYPE\s+HTML\s+PUBLIC\s+"-//W3C//DTD\ HTML\ 4\.01//EN"
        \s+
        "http://www\.w3\.org/TR/html4/strict\.dtd"
        \s*>
</cfregex>
```

Quote

The `quote` action will wrap a string in `\Q` and `\E` meaning it can be used within a regex without needing to escape individual metacharacters.

This can be an easier to read solution than the Escape action, but is less flexible (since you cannot quote `\E` itself). It is also not possible to use `quote` within character classes, whereas Escape can be used here also.

Object

Although it is possible to call `quote` for a regex object, it is rare that you will need to do this. The RegexQuote function is usually recommended instead.

Arguments

No arguments.

Usage Examples

```
<cfset ExampleRx = new Regex('\w+\s{2,}') />

<cfdump var=#ExampleRx.quote()# />
```

string `\Q\w+\s{2,}\E`

Tag

Since the cfregex tag always treats its body content as a regex, the contents must currently be a valid regex pattern, otherwise an error is thrown - even when using the `quote` action.

To convert arbitrary text that is not a regex pattern, use the RegexQuote function instead. (If necessary, in combination with cfsavecontent.)

Attributes

Name	Type	Required	Default	Notes
Variable	VarName	no	"cfregex"	The variable which the escaped regex is assigned to.

Usage Examples

```
<cfregex quote variable="RegexQuoted" >
    \w+
    \s{2,}
</cfregex>
<cfdump var=#RegexQuoted# />
```

string `\Q\w+\s{2,}\E`

Function

The RegexQuote function accepts the argument `Text`, not `Pattern`, since it does not compile to a regex pattern.

This means it is possible to use RegexQuote against a non-regex piece of text and make it safe for use in regex. (This is not possible using the cfregex tag with

action `quote`, since the tag always compiles its contents to a regex.)

Arguments

Name	Type	Required	Default	Notes
Text	String	yes	n/a	

Usage Examples

```
<cfdump var=#RegexQuote('\w+\s{2,}')# />
```

string \Q\w+\s{2,}\E

```
<cfdump var=#RegexQuote('*\o/* :)')# />
```

string \Q*\o/* :)\E

Replace

The `replace` action replaces occurances of the matched regex in the input text with the specified replacement. Like other cfRegex actions you can specify a starting position, and can apply any limit to how many times replacement is made (not just "one" or "all", as `rereplace` has).

The value to replace each match with can be specified in one of three ways: as a replacement string, a replacement array, or a callback function that returns a string.

Replacement Types

string

If you specify a simple string variable the match found in the input text will be replaced with the contents of this variable. If you include `$0` in the replacement string, this refers to the entire matched text, and use `$1`..`$n` to refer to numbered groups. *(Where `n` indicates is the number of captured groups, not the literal letter n.)*

If you have ten (or more) groups, you can use `$10`, however if you have less than ten groups this will be interpreted as `$1` (i.e group 1), then a literal `0` character, similarly with $11, $12, $20, etc.

If you need a `$` in your replacement string, you need to use `\$`, and if you need `\$` you need to use `\\$`, and so on.

function

If you provide a function for the replacement variable, that function is called each time a match is made, and the result of the function is used as the text to replace the match with.

The function must be a UDF or an object method (i.e. built-in functions cannot be used directly).

These callback functions receive a selection of arguments containing information relating to the match - the text matched, the groups found, and so on. Full details of what arguments a function should receive are detailed on the page dedicated to callbacks.

The string result of the callback is subject to the same processing rules as a replacement string, so `$0`..`$n` are replaced by group text and `\$` is needed to represent a literal `$` character.

array

By specifying an array of replacements, you can use a different value for each match found. If the number of matches found is greater than the length of the array supplied, the replacements start again from the beginning of the array.

Each element in the array can be *either* a replacement string *or* a callback function - you can use a combination of both in the same array.

String replacement values use the same rules as the replacement string (i.e. use `$0`..`$n`), and callback functions work in the same way as they would when passed singularly.

NOTE: Due to a bug in Adobe ColdFusion, only string replacements can be used in arrays with ACF9. You can still use callbacks outside of arrays for ACF9. Neither Railo not OpenBD are affected by this bug, so this feature works fine for them.

Object

Arguments

Name	Type	Required	Default	Notes
Text	String	yes	n/a	The text in which the regex is to be replaced
Replacement	String *OR* Callback *OR* Array	yes	n/a	See section "Replacement Types" above.
Start	Char Position	no	1	Position at which to start replacing (1 is first character.)
Limit	Integer	no	0	Number of times to replace before stopping. (0 is unlimited.)
CallbackData	Struct	no	none	A structure which is passed into the callback function.

Usage Examples

```
<cfset Input = "The quick fox jumps over the lazy brown dog." />
<cfset OneWordRx = new Regex( '\w+' ) />
```

```
<cfdump var=#OneWordRx.replace( Input , "[word]" )# />
```

string [word] [word] [word] [word] [word] [word] [word] [word] [word].

```
<cfdump var=#OneWordRx.replace( Input , "[word]" , 5 )# />
```

string The q[word] [word] [word] [word] [word] [word] [word] [word].

```
<cfdump var=#OneWordRx.replace( Input , "[word]" , 5 , 2 )# />
```

string The q[word] [word] jumps over the lazy brown dog.

```
<cfdump var=#OneWordRx.replace( Input , "[$0]" )# />
```

string [The] [quick] [fox] [jumps] [over] [the] [lazy] [brown] [dog].

```
<cfdump var=#OneWordRx.replace( Input , ["[word]","$0"] )# />
```

string [word] quick [word] jumps [word] the [word] brown [word].

```
<cfdump var=#OneWordRx.replace( text=Input , replacement=["[word]","[$0]"] , limit = 4 )# />
```

string [word] [quick] [word] [jumps] over the lazy brown dog.

```
<cfdump var=#OneWordRx.replace( Input , UcaseFunc , 1 , 3 )# />

<cffunction name="UcaseFunc" returntype="String" output="false">
    <cfargument name="Match" type="String" required />
    <cfreturn UCase(Argments.Match) />
</cffunction>
```

string THE QUICK FOX jumps over the lazy brown dog.

```
<cfdump var=#OneWordRx.replace( Input , [ UcaseFunc , "[word]" ] , 1 , 3 )# />

<cffunction name="UcaseFunc" returntype="String" output="false">
    <cfargument name="Match" type="String" required />
    <cfreturn UCase(Argments.Match) />
</cffunction>
```

string THE [word] FOX jumps over the lazy brown dog.

Tag

Attributes

Name	Type	Required	Default	Notes
Variable	VarName	no	"cfregex"	The variable which the result is assigned to.
Text	String	yes	n/a	The text in which the regex is to be replaced
Replacement	String *OR* Callback *OR* Array	yes	n/a	See section "Replacement Types" above.
Start	Char Position	no	1	Position at which to start replacing (1 is first character.)
Limit	Integer	no	0	Number of times to replace before stopping. (0 is unlimited.)
CallbackData	Struct	no	none	A structure which is passed into the callback function.
Modes	StringList	no	none	List of regex modes to apply to the pattern.

Usage Examples

```
<cfset Input = "The quick fox jumps over the lazy brown dog." />
<cfset OneWordRx = new Regex( '\w+' ) />
```

```
<cfregex replace variable="Output" text=#Input# replacement="[word]" >
    \w+
</cfregex>
<dump var=#Output#/>
```

string [word] [word] [word] [word] [word] [word] [word] [word] [word].

```
<cfregex replace variable="Output" text=#Input# replacement="[word]" , 5 >
    \w+
</cfregex>
<dump var=#Output#/>
```

string | The q[word] [word] [word] [word] [word] [word] [word] [word].

```
<cfregex replace variable="Output" text=#Input# replacement="[word]" start=5 limit=2 >
    \w+
</cfregex>
<dump var=#Output#/>
```

string | The q[word] [word] jumps over the lazy brown dog.

```
<cfregex replace variable="Output" text=#Input# replacement="[$0]" >
    \w+
</cfregex>
<dump var=#Output#/>
```

string | [The] [quick] [fox] [jumps] [over] [the] [lazy] [brown] [dog].

```
<cfregex replace variable="Output" text=#Input# replacement=#["[word]","$0"]# >
    \w+
</cfregex>
<dump var=#Output#/>
```

string | [word] quick [word] jumps [word] the [word] brown [word].

```
<cfregex replace variable="Output" text=#Input# replacement=#["[word]","[$0]"]# limit=4 >
    \w+
</cfregex>
<dump var=#Output#/>
```

string | [word] [quick] [word] [jumps] over the lazy brown dog.

```
<cfregex replace variable="Output" text=#Input# replacement=#UcaseFunc# limit=3 >
    \w+
</cfregex>
<dump var=#Output#/>
```

```
<cffunction name="UcaseFunc" returntype="String" output="false">
    <cfargument name="Match" type="String" required />
    <cfreturn UCase(Argments.Match) />
</cffunction>
```

string | THE QUICK FOX jumps over the lazy brown dog.

```
<cfregex replace variable="Output" text=#Input# replacement=#[ UcaseFunc , "[word]" ]# limit=3 >
    \w+
</cfregex>
<dump var=#Output#/>
```

```
<cffunction name="UcaseFunc" returntype="String" output="false">
    <cfargument name="Match" type="String" required />
    <cfreturn UCase(Argments.Match) />
</cffunction>
```

string | THE [word] FOX jumps over the lazy brown dog.

Function

Arguments

Name	Type	Required	Default	Notes
Pattern	RegexString	yes	n/a	The regex pattern to compile into a Regex Object.
Text	String	yes	n/a	The text in which the regex is to be replaced
Replacement	String *OR* Callback *OR* Array	yes	n/a	See section "Replacement Types" above.
Start	Char Position	no	1	Position at which to start replacing (1 is first character.)
Limit	Integer	no	0	Number of times to replace before stopping. (0 is unlimited.)
CallbackData	Struct	no	none	A structure which is passed into the callback function.
Modes	StringList	no	none	List of regex modes to apply to the pattern.

Usage Examples

```
<cfset Input = "The quick fox jumps over the lazy brown dog." />
```

```
<cfdump var=#RegexReplace( '\w+' , Input , "[word]" )# />
```

string [word] [word] [word] [word] [word] [word] [word] [word] [word].

```
<cfdump var=#RegexReplace( '\w+' , Input , "[word]" , 5 )# />
```

string The q[word] [word] [word] [word] [word] [word] [word] [word].

```
<cfdump var=#RegexReplace( '\w+' , Input , "[word]" , 5 , 2 )# />
```

string The q[word] [word] jumps over the lazy brown dog.

```
<cfdump var=#RegexReplace( '\w+' , Input , "[$0]" )# />
```

string [The] [quick] [fox] [jumps] [over] [the] [lazy] [brown] [dog].

```
<cfdump var=#RegexReplace( '\w+' , Input , ["[word]","$0"] )# />
```

string [word] quick [word] jumps [word] the [word] brown [word].

```
<cfdump var=#RegexReplace( pattern='\w+' , text=Input , replacement=["[word]","[$0]"] , limit = 4 )# />
```

string [word] [quick] [word] [jumps] over the lazy brown dog.

```
<cfdump var=#RegexReplace( '\w+' , Input , UcaseFunc , 1 , 3 )# />

<cffunction name="UcaseFunc" returntype="String" output="false">
    <cfargument name="Match" type="String" required />
    <cfreturn UCase(Arguments.Match) />
</cffunction>
```

string THE QUICK FOX jumps over the lazy brown dog.

```
<cfdump var=#RegexReplace( '\w+' , Input , [ UcaseFunc , "[word]" ] , 1 , 3 )# />

<cffunction name="UcaseFunc" returntype="String" output="false">
    <cfargument name="Match" type="String" required />
    <cfreturn UCase(Arguments.Match) />
</cffunction>
```

string THE [word] FOX jumps over the lazy brown dog.

Practical Examples

Example 1

A function which appends cfthrow after all TODO tasks in some CFML code:

```
<cffunction name="convertTodoToThrow" returntype="String" output="false" access="public">
    <cfargument name="Content" type="String" required />
    <cfset var Result = Arguments.Content />

    <cfif NOT StructKeyExists(Variables,'TodoTaxRx')>
        <cfset Variables.TodoTagRx    = new Regex('<!---\s*TODO:\s*([^-]++(?s:(?!--->).)*)--->') />
        <cfset Variables.TodoScriptRx = new Regex('//\s*TODO:\s*([^\n]+)') />
    </cfif>

    <!--- $0 is the entire matched tech, $1 is the first captured group (i.e. text of the TODO) --->
    <cfset Result = Variables.TodoTagRx.replace( Result , '$0#chr(10)#<cfthrow type="TODO" message="$1" />' ) />
    <cfset Result = Variables.TodoScriptRx.replace( Result , '$0#chr(10)#throw( type="TODO" message="$1" );' )/>

    <cfreturn Result />
</cffunction>
```

Example 2

This example uses a replacement callback to converts all IPv4 addresses to IPv6 format, excluding those specified by the callbackdata.

```
<cfset DocumentText = RegexReplace
    ( Pattern      = '([12]\d\d)\.([12]?\d\d)\.([12]?\d\d)\.([12]?\d\d)'
    , Text         = DocumentText
    , Replacement  = convertIP4toIP6
    , CallbackData = {ExcludedIPs:['192.168.0.1','127.0.0.1']}
    ) />

<cffunction name="convertIP4toIP6" returntype="String" output="false">
    <cfargument name="Match"  required />
    <cfargument name="Groups" required />
    <cfargument name="Data"   required />

    <cfif StructKeyExists(Arguments.Data,'ExcludedIps')
        AND ArrayFind(Arguments.Data.ExcludedIps,Arguments.Match)
        >
        <cfreturn Arguments.Match />
    </cfif>

    <cfreturn "::ffff:"
        & toHex( Arguments.Groups[1].Match )
        & toHex( Arguments.Groups[2].Match )
        & ":"
        & toHex( Arguments.Groups[3].Match )
        & toHex( Arguments.Groups[4].Match )
        />
</cffunction>


<cffunction name="toHex" returntype="String" output="false">
    <cfreturn Right( '0' & FormatBaseN( Arguments[1] , 16 ) , 2) />
</cffunction>
```

Split

The `split` action allows you to convert a string into an array, treating regex matches as delimiters. In effect, this is the opposite of Match, which creates an array of the matches; whilst Split populates the array with the text that does *not* match the regex.

You can use `start` to specify a character position before which no splitting takes place, and use `limit` to specify the maximum number of times the text should be split.

If you have complicated splitting conditions, you can pass in a callback function which will be called each time a match is found, and the split will only occur if the function returns true.

Object

Arguments

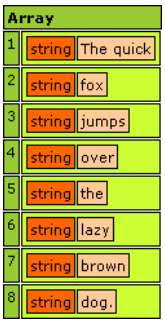
Name	Type	Required	Default	Notes
Text	String	yes	n/a	The text which is to be split by the regex.
Start	Char Position	no	1	Position at which to start splitting (1 is first character.)
Limit	Integer	no	0	Number of times to split before stopping. (0 is unlimited.)
Callback	Function	no	none	A function called each time a match is made. If function returns false the split does not occur (and does not count towards limit). See Callbacks section for full details on function signature and how to use this feature.
CallbackData	Struct	no	none	A structure which is passed into the callback function.

Usage Examples

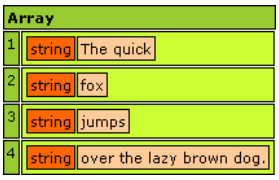
```
<cfset Input = "The quick fox jumps over the lazy brown dog." />
<cfset WhitespaceRx = new Regex( '\s+' ) />
<cfset WordRx = new Regex( '\w+' ) />
```

```
<cfdump var=#WhitespaceRx.split( Input )# />
```


```
<cfdump var=#WhitespaceRx.split( Input , 5 )# />
```

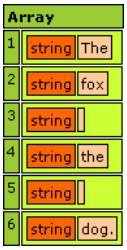



```
<cfdump var=#WhitespaceRx.split( Input , 5 , 3 )# />
```



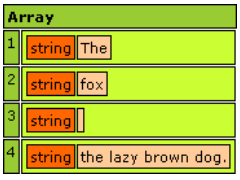

```
<cfdump var=#WordRx.split( text=Input , callback=checkWord )# />

<cffunction name="checkWord" returntype="Boolean" output="false">
    <cfargument name="Match" type="String" />
    <cfreturn Len(Arguments.Match) GTE 4 />
</cffunction>
```




```
<cfdump var=#WordRx.split( text=Input , callback=checkWord , limit=3 )# />

<cffunction name="checkWord" returntype="Boolean" output="false">
    <cfargument name="Match" type="String" />
    <cfreturn Len(Arguments.Match) GTE 4 />
</cffunction>
```



Tag

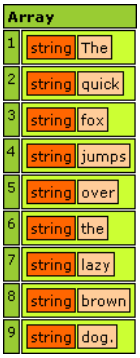
Attributes

Name	Type	Required	Default	Notes
Variable	VarName	no	"cfregex"	The variable which the result is assigned to.
Text	String	yes	n/a	The text which is to be split by the regex.
Start	Char Position	no	1	Position at which to start splitting (1 is first character.)
Limit	Integer	no	0	Number of times to split before stopping. (0 is unlimited.)
Callback	Function	no	none	A function called each time a match is made. If function returns false the split does not occur (and does not count towards limit). See Callbacks section for full details on function signature and how to use this feature.
CallbackData	Struct	no	none	A structure which is passed into the callback function.
Modes	StringList	no	none	List of regex modes to apply to the pattern.

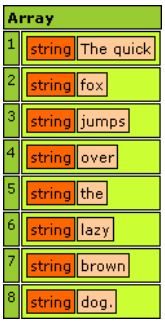
Usage Examples

```
<cfset Input = "The quick fox jumps over the lazy brown dog." />
```

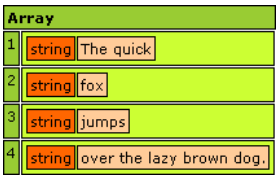
```
<cfregex split variable="Output" text=#Input# >
    \s+
</cfregex>
<dump var=#Output#/>
```




```
<cfregex split variable="Output" text=#Input# start=5 >
    \s+
</cfregex>
<dump var=#Output#/>
```

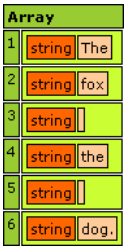



```
<cfregex split variable="Output" text=#Input# start=5 limit=3 >
    \s+
</cfregex>
<dump var=#Output#/>
```




```
<cfregex split variable="Output" text=#Input# callback=#checkWord# >
    \w+
</cfregex>
<dump var=#Output#/>

<cffunction name="checkWord" returntype="Boolean" output="false">
    <cfargument name="Match" type="String" />
    <cfreturn Len(Arguments.Match) GTE 4 />
</cffunction>
```



```
<cfregex split variable="Output" text=#Input# callback=#checkWord# limit=3 >
    \w+
</cfregex>
<dump var=#Output#/>

<cffunction name="checkWord" returntype="Boolean" output="false">
    <cfargument name="Match" type="String" />
    <cfreturn Len(Arguments.Match) GTE 4 />
</cffunction>
```

Function

Arguments

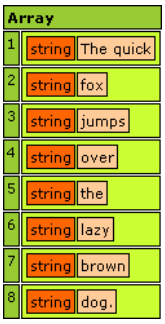
Name	Type	Required	Default	Notes
Pattern	RegexString	yes	n/a	The regex pattern to compile into a Regex Object.
Text	String	yes	n/a	The text which is to be split by the regex.
Start	Char Position	no	1	Position at which to start splitting (1 is first character.)
Limit	Integer	no	0	Number of times to split before stopping. (0 is unlimited.)
Callback	Function	no	none	A function called each time a match is made. If function returns false the split does not occur (and does not count towards limit). See Callbacks section for full details on function signature and how to use this feature.
CallbackData	Struct	no	none	A structure which is passed into the callback function.
Modes	StringList	no	none	List of regex modes to apply to the pattern.

Usage Examples

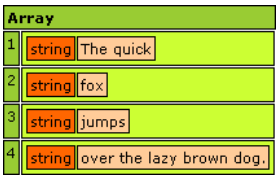
```
<cfset Input = "The quick fox jumps over the lazy brown dog." />
```

```
<cfdump var=#RegexSplit( '\s+' , Input )# />
```


```
<cfdump var=#RegexSplit( '\s+' , Input , 5 )# />
```

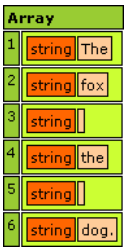



```
<cfdump var=#RegexSplit( '\s+' , Input , 5 , 3 )# />
```



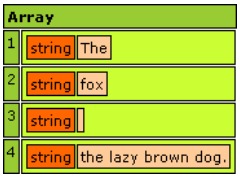

```
<cfdump var=#RegexSplit( pattern='\w+' , text=Input , callback=checkWord )# />

<cffunction name="checkWord" returntype="Boolean" output="false">
    <cfargument name="Match" type="String" />
    <cfreturn Len(Arguments.Match) GTE 4 />
</cffunction>
```




```
<cfdump var=#RegexSplit( pattern='\w+' , text=Input , callback=checkWord , limit=3 )# />

<cffunction name="checkWord" returntype="Boolean" output="false">
    <cfargument name="Match" type="String" />
    <cfreturn Len(Arguments.Match) GTE 4 />
</cffunction>
```



Practical Examples

Example 1

Split on a comma that has not been escaped with a backslash:

```
<cfregex
    action   = "split"
    variable = "OutputArray"
    text     = #InputString#
    >
    ## lookbehind for either...
    (?<=
        ## not a backslash
        [^\\]
    |
        ## or, an EVEN number of backslashes
        ## (an odd number would be escaped)
        (?<!\\)(?:\\\\){1,10}
    )
    ## followed by a comma
    ,
</cfregex>
```

Features

This section describes the features which cfRegex provides. Unlike Actions which are called directly and *do something*, a feature is something which provides additional functionality across multiple actions, to change how an action works.

Another way of looking at it, features are things that work in addition to the regex engine functionality, and are applicable to more than one action.

Regex Functionality

The cfRegex project provides a number of features over the core CFML RE functions by means of simply using a newer regex engine. Since these are enhancements at the regex-level, they are documented in the general regex features section. They include:

- Lookbehinds: `(?<=...)` and `(?<!...)`
- Atomic Groups: `(?>...)`
- Possessive Quantifiers: `.++` and `.*+` and `.{1,4}+`
- Unicode Character Classes: `\p{...}` and `\P{...}`

(The remainder of the features discussed on this page are ones which cfRegex provides distinct from the regex engine used.)

Callbacks

A callback function lets you supply CFML logic that is executed each time a regex match is made, which provides a great deal of power and flexibility.

Callbacks can be used for the Match, Replace, and Split actions.

For full details on how callbacks work, see the dedicated Callbacks page.

Limit

In the core CFML RE functions, `refind` only matches once, `rematch` always matches everything, and `rereplace` can only replace once or everything, which results in far less efficient code when doing anything other than these things.

In cfRegex, all actions that have the ability to match multiple times will match as many times as possible, by default, and they all accept the `Limit` parameter which can be used to control how many times the action occurs.

Limit can be used with Find, Match, Replace, and Split actions.

See the Limit page for more information on how it works.

Start

In the core CFML RE functions, `refind` lets you specify a starting character position which allows you to skip the first part of the input text. This is essential to allow for looping through multiple times (since `refind` can only match once per call).

With cfRegex, this `start` parameter has been extended so that all actions that have a limit parameter also accept the `start` parameter, to provide a convenient way to skip text that you do not want the regex pattern to see.

See the Start page for more information on how it works.

Callbacks

Callbacks are a powerful feature that allows you to provide a CFML function to be executed every time a regex match is made, which enables you to run your own logic that can override the default behaviour.

There are two types of callbacks, Boolean Callbacks return either true or false to say "perform default behaviour" or "don't perform it", and are used with Match and Split actions, whilst String Callbacks are used with Replace and return the value to use for the replacement text. Both types receive identical arguments - the only difference is how their returned value is used.

A callback function must be a UDF or an object method (i.e. it cannot be a built-in function), and needs to be a variable (do not surround it with quotes, and in tag-form use hashes).

Arguments

A callback function will receive the following named arguments:

Name	Type	Passed	Notes
Pos	Char Position	always	The position which the current match starts at.
Len	Integer	always	The length of the current match (can be zero).
Match	String	always	The text of the current match.
Groups	Array	always	An array containing numbered group information.
NamedGroups	Struct	sometimes	A struct containing named group information, *if* `GroupNames` passed to calling function.
Data	Struct	sometimes	A struct containing passed-in data, *if* `CallbackData` passed to calling function.

The callback function is called using named arguments via argumentcollection, so *order* is not important, but the *names* must match exactly.

The `Groups` and `NamedGroups` arguments both contain the same data, but the former uses backreference positions, whilst the latter uses the names supplied to the original calling function for the keys of the structure.

Each group element within both these arguments contains a structure with keys `Pos`, `Len`, and `Match` with the values for that particular group.

Example Functions

Below are a couple of example functions which you can use as a base to create your own callback functions.

A sample boolean callback function which doesn't change behaviour (i.e. always returns true):

```
<cffunction name="BooleanCallback" returntype="Boolean" output="false">
    <cfargument name="Pos"         type="Numeric" required  hint="The start position of the match."  />
    <cfargument name="Len"         type="Numeric" required  hint="The length of the match."          />
    <cfargument name="Match"       type="String"  required  hint="The text of the match."            />
    <cfargument name="Groups"      type="Array"   required  hint="Array of group information."       />
    <cfargument name="NamedGroups" type="Struct"  optional  hint="Struct of named group information." />
    <cfargument name="Data"        type="Struct"  optional  hint="Struct containing passed-in data." />

    <cfreturn true />
</cffunction>
```

A sample replace callback function which doesn't change the result (i.e. always returns same text as was matched):

```
<cffunction name="ReplaceCallback" returntype="String" output="false">
    <cfargument name="Pos"         type="Numeric" required  hint="The start position of the match."  />
    <cfargument name="Len"         type="Numeric" required  hint="The length of the match."          />
    <cfargument name="Match"       type="String"  required  hint="The text of the match."            />
    <cfargument name="Groups"      type="Array"   required  hint="Array of group information."       />
    <cfargument name="NamedGroups" type="Struct"  optional  hint="Struct of named group information." />
    <cfargument name="Data"        type="Struct"  optional  hint="Struct containing passed-in data." />

    <cfreturn Arguments.Match />
</cffunction>
```

Note that you do not need to specify all arguments with cfargument tags - the function will receive them, but if you prefer, you can specify only the ones you are using. (Since they are provided as named arguments, you don't *strictly* need the cfargument tags at all.)

Usage Examples

Callback functions can be used any time you want to apply CFML logic to a match, and avoid having to use complicated and messy hacks like replacement tokens.

Example 1

For example, if you had a document containing codes which needed to be updated to use newer codes from a database, you might do this:

```
 <cfset DocText = RegexReplace
    ( '\b[A-Z]{2}-\d{4}\b'
    , DocText
    , CodeReplaceFunc
    ) />

 <cffunction name="CodeReplaceFunc" returntype="String" output="false">
    <cfargument name="Pos"         type="Numeric" required  hint="The start position of the match."  />
    <cfargument name="Match"       type="String"  required  hint="The text of the match."            />

    <cfif Application.Codes.isOld(Arguments.Match)>

        <cfset var NewCode = Application.Codes.lookupNewCode(Arguments.Match) />

        <cflog
            file = "DocCodeReplace"
            text = "Converted '#Arguments.Match#' at #Arguments.Pos# to '#NewCode#'"
        />

        <cfreturn NewCode />
    <cfelse>

        <cflog
            file = "DocCodeReplace"
            text = "Skipped '#Arguments.Match#' at #Arguments.Pos#"
        />

        <cfreturn Arguments.Match />
    </cfif>
 </cffunction>
```

Example 2

This example shows how you might extract a list of country codes from a document. The regex filters down possible candidates, before using an existing function to confirm which ones are valid.

The end result is the `Countries` array only contains valid countries.

```
<cfregex match
    variable = "Countries"
    text     = #DocText#
    callback = #CountryCheckFunc#
    >
    ## Lookbehind to ensure whitespace, start of string, or other valid prefix.
    (?<=\s|\A|["'])

    ## Negative lookahead to exclude unused codes (AAA-AAZ,QMA-QZZ,XAA-XZZ,ZZA-ZZZ)
    (?!AA|ZZ|X|Q[M-Z])

    ## Any two uppercase letters (except as excluded above).
    [A-Z]{3}

    ## Lookahead to ensure whitespace, end of string, or other valid suffix
    (?=\s|\z|\b[^/:])
</cfregex>

<cffunction name="CountryCheckFunc" returntype="Boolean" output="false">
    <cfargument name="Match" type="String" required />

    <cfreturn Application.Countries.isValid(ISO=Arguments.Match) />
</cffunction>
```

Limit

In the core CFML RE functions, `refind` only matches once, `rematch` always matches everything, and `rereplace` can only replace once or everything, which results in far less efficient code when doing anything other than these things.

In cfRegex, all Actions that have the ability to match multiple times will match as many times as possible, by default, and they all accept the `Limit` parameter which can be used to control how many times the action occurs.

Using Match as an example:

```
<cfset FirstNumber = RegexMatch( Pattern = '\d+' , Text = LargeInput , Limit = 1 ) />
```

will always be more efficient than:

```
<cfset FirstNumber = ReMatch( reg_expression = '\d+' , string = LargeInput )[1] />
```

This is because `ReMatch` has to return all matches before the first one can be extracted, whilst `RegexMatch` stops as soon as the limit is hit.

Limit accepts a positive integer; a limit of zero means "no limit" or unlimited.

It is important to understand, limit indicates **the number of times the action occurs** - for Split this means the resulting array will be one larger than the limit number, which is different to java.util.regex's native split limit, but consistent with the way limit works with the rest of cfRegex.

When a boolean callback function is used, if it returns `false` the action does not occur, so the match does not count towards the limit - only returning `true` from the callback will count towards the limit.

Limit can be used with Find, Match, Replace, and Split actions.

Start

In the core CFML RE functions, `refind` lets you specify a starting character position which allows you to skip the first part of the input text. This is essential to allow for looping through multiple times (since `refind` can only match once per call).

With cfRegex, this `start` parameter has been extended to all actions that have a limit parameter, to provide a convenient way to skip text that you do not want the regex pattern to see.

For example, to exclude the first line from a search you might do:

```
<cfset Something = RegexMatch
    ( pattern = '...'
    , text    = InputText
    , start   = find( Chr(10) , InputText )
    ) />
```

Start accepts a positive integer to start *before* - a value of 1 means the start of the string, a value of `len(string)` is the position before the last character of the string.

The text skipped with the `start` parameter is *not* available to lookbehinds, (though this *might* change in future). (For now, if you need to skip X characters but also need a lookbehind, you can begin your expression with `(?s:.){X}` - where `X` is a positive integer.)

Regex Concepts

When dealing with regex, what people generally bring to mind is a seemingly arcane string of characters and, whilst this regex pattern is a key part of regex, it is just a *part* of it.

The pattern is a series of instructions to the regex engine on how to match whatever it has been created to match. However, the regex engine is also subject to a series of modes which alter the way certain regex instructions are interpreted, and it is important to understand what these modes do, and know when you need to enable or disable them.

The pattern and modes are combined to build the regex object, which can then be executed against a particular input text to identify positions where the pattern matches, but they only dictate what is matched, but not how this occurs.

The mechanics of how patterns are applied can get complicated, and developing a working knowledge of all the factors involved is not a trivial task, however it is important to understand at least the basics, in order to craft expressions correctly, and to avoid mistakes which can lead to bad performance. Fortunately there are a handful of rules which are pretty simple to learn, and it is well worth doing so.

By understanding these concepts of what a regex engine is being instructed to do and how it executes these instructions you will significantly improve your regex skills.

Regex Patterns

A regex pattern is the string of characters which provides instructions to the regex engine on what is to be matched.

A pattern describes these instructions using assorted punctuation and other symbols known as metacharacters, which can be used alongside normal characters.

Patterns can be used to match specific positions, one or more characters, or a combination of characters and positions.

A pattern can also be called an expression, and within an expression you can have multiple sub-expressions, also known as groups, which can capture a particular segment of text, and/or be used to treating a collection of characters as a single unit.

For full details on the syntax of regex patterns, see the Regex Features section.

Regex Modes

In regex there are a number of different modes which can be applied to control aspects of how patterns are matched, generally relating to how newlines are detected, and whether uppercase and lowercase characters are considered the same.

Using Modes

To apply a mode to a regex, you supply the `Modes` parameter when compiling the regex - either via the dedicated Compile action, or when using any of the functions or tags with a single-use pattern.

The parameter accepts a comma-delimited list of codes, to determine which modes to turn on. (The codes for each mode can be found detailed below.)

Flags

Modes can also be defined as part of a regex pattern itself, in the form of flags which enable or disable that mode. (The flags for each mode can be found detailed below.)

A flag can be turned on for the rest of the expression, by using **`(?`***`flag`***`)`**, turned off by using **`(?-`***`flag`***`)`** and applied for only part of an expression by using **`(?`***`flag`***`:`***`expression`***`)`**

For example:

```
(?i)this is case insensitive
```

```
(?i)this is case insensitive (?-i) but this is not
```

```
(?i:this is case insensitive) but this is not
```

```
(?i:this is (?-i:except this part) case insensitive)
```

You can enable or disable multiple flags at once, like so:

```
(?is)case insensitive and single-line enabled
```

```
(?m-d:multi-line enabled, but unix lines disabled)
```

In cfRegex, all modes are disabled by default, with the exception of Comment mode which is enabled when using the cfregex tag (but disabled for functions).

Available Modes

Unix Lines

code: **UNIX_LINES**
flag: **d**

Tells the regex engine that only the newline character (`\n`) should be treated as part of a newline.

Carriage returns (`\r`) should not be considered a part of a newline.

This is significant when combined with Multiline mode and the `^` and `$` markers are used to match start/end of lines.

Case Insensitivity
code: **CASE_INSENSITIVE**
flag: **i**

Means that uppercase and lowercase letters are not differentiated - i.e. "abc" and "ABC" are considered the same.

Comment
code: **COMMENTS**
flag: **x**

Enables commenting and free-spacing mode.

All whitespace is ignored unless preceeded by a backslash.

When a hash (`#`) is encountered, all content until the end of the line is ignored.

Multi-line
code: **MULTILINE**
flag: **m**

When enabled, the `^` and `$` markers will match start and end of lines, as opposed to just start and end of input (which can be matched using `\A` and `\z` instead.)

Dot-All or Single-line
code: **DOTALL**
flag: **s**

By default, the `.` character matches everything except newlines.

The dot-all mode means `.` matches everything including newlines.

Unicode Case-insensitivity
code: **UNICODE_CASE**
flag: **u**

This mode is similar to Case-insensitivity, but whilst that applies only to standard characters, this will apply to unicode characters also.

Canonical Equivalence
code: **CANON_EQ**
flag: **c**

When this flag is enabled, two characters will be considered to match if their full canonical decompositions match.

Matching Mechanics

Knowing how a regex engine interprets the rules of a regex pattern is useful, as it helps you to write more efficient expressions and avoid certain mistakes.

The core regex engine used by cfRegex is the java.util.regex library, which uses an algorithm known as "backtracking NFA".

Whilst you don't need to know the full details of the underlying theory of how this algorithm works and how it compares to other possible implementations, it is useful to understand a number of practical aspects to it, which are described in the sections below.

Regex works Left-to-Right

This is something common to all existing regex engines - matching is done from left-to-right, in all situations. Whilst a regex matcher might skip back to a previous position, it never actually scans from right-to-left

If you need to deal only with something at the end of a string (such as a file extension) it is always best to isolate the end of the string using simpler methods before trying to match a regex against it. (e.g. use `Right(string,num)` or `ListLast(string,delimiter)` as these both work by starting from the right and moving towards the left.)

To demonstrate this behaviour, consider the text "the quick fox jumps over the lazy brown dog", and the expression "`\b.+?dog`". You might think the "`.+?`" will match at least once but as few characters as possible, and so the closest the "`\b`" can match to "dog" is after the "n" of "brown", before the space - however this is falling into the trap of applying your own direction to the interpretation of the instructions.

The regex engine starts with the "`\b`", which matches immediately, before the first "t", then the "`.+?`" matches the "t", tries to pass control to the next instruction (because it is a lazy quantifier), but this keeps failing and passes back to the lazy quantifier, which keeps expanding it's match one character at a time, until it has eventually found the "dog" at the end and ended up matching the *entire* text.

So remember: Regex always starts with the first instruction and at the start (left-most position) of a string of characters.

Backtracking

One of the key benefits of regex is its ability to try multiple alternatives, however it is not always obvious what the cost of this may be.

When matching, a regex engine applies the regex instructions one at a time. If the instruction succeeds, it moves forward and tries the next instruction. When an instruction fails, it looks to see if there are alternatives, if there are none it skips back to the last point it had alternatives and tries a different option, and it repeats this whole process until either the regex has matched, or the regex engine has ran out of alternatives to try.

A regex matcher will not skip forward to future parts of the regex and try those first. It will not even check what the next instruction is and keep track of when that might match - it deals with the current instruction completely before even considering what the following instruction might be.

For example, the pattern `^.+(?=://)` might be used in order to match the protocol from a URL - basically saying, "start at the beginning, match any character as

many times as possible, then look for (but don't match) ://".

However, what happens here is that the `+` quantifer matches the *entire* input string until the end, then it tries (and fails) to match :// and it backtracks one character at a time, attempting to match, failing, and backtracking again, until eventually it reaches almost the start of the string, and eventually finds the :// at position four (assuming a string starting "http://...").

The most common solution people use for this is to use the "lazy" `+?` quantifier instead, however whilst this is significantly faster it will still involve unnecessary backtracking. Lazy quantifiers will match one unit at a time, then immediately say "ok, what's next", and (when the next instruction fails) control is passed back, an additional unit is matched, and then it passes control forward again. For this example that's not going to matter much, but for longer strings this can be just as much as issue as the `.+` originally used.

The improved solution is not to use `.` but to be more specific with what is expected to be found. Using either `[^:]+` or `\w+` will both prevent the regex going all the way along the string, whilst also avoiding the continuous backtracking that a lazy quantifier can cause.

See the following diagram for a more visible example of what is going on:

Sometimes backtracking is unavoidable - without it regex would not be as useful as it is - but the key is to try to keep it to a minimum, and if it can be avoided altogether that's even better.

Alternation uses the earliest match.

When using pipe "`|`" to allow a number of alternatives, it is important to keep in mind that regex does not even consider later options if an earlier one has matched successfully.

The expression `(car|cart|carpet)(\w*)` matched against the text "carpets" will place "car" in group one and "pets" in group two.

In *other* regex implemenations (ones without backtracking), all three alternatives are matched in parallel and the longest match is used (which would result in "carpet" and "s").

If we changed the expression to `(car|cart|carpet)([^p]\w*)` and still matched against the text "carpets", the first group would still match "car", however (when attempting to match the second group) the `[^p]` would prevent a succesful match, so the regex engine would backtrack, attempt "cart" which fails, then attempt "carpet" which succeeds, then move forward to the second group again, which would match the "s".

For this reason, you should order alternatives so the most likely to match is first and the least likely of the options is last.

Regex Features

Introduction

This section covers the basic features which are used in a regex pattern to instruct the regex engine what it is looking for.

Whilst it is intended to give someone new to regex a knowledge of all the syntax elements, the docs also cover things which may be new even to experienced regex users, so is worth reading through whatever your level.

If there is anything within these pages that you are unsure about or that could be explained better, please send a message to the mailing list - this documentation is still a work in progress, and feedback is welcome!

Features

Metacharacters are characters in a regex pattern with special meaning which make up the instructions which the regex engine uses when attempting to match.

These can be used to indicate characters which should match, but also to verify that the matches occur at particular Positions within the text.

Sometimes it is necessary to use Encoded Characters in regex, if the literal character would be unclear or is not able to be represented directly.

Character Classes are used to represent any single character out of a set of possible matches, or to represent a negated set of characters which should not match.

Quantifiers allow the specification of minimum and/or maximum number of times the preceeding item is allowed to match.

Groups allow you to treat several characters as a single item to be acted on, as well as being able to capture the contents of the group for use in backreferences.

Alternation is a way to specify multiple possible paths down which a regex can match.

Metacharacters

A metacharacter is one or more characters that possess special meaning within a regex pattern, and can be used to build up the instructions which are used when applying a regex to a string.

Within a regex, everything is either a metacharacter or a literal character - the latter simply being a character that does not possess special meaning, but that matches the same character it represents.

The meaning of some metacharacters varies depending on if any regex modes are enabled - for example, the caret "^" metacharacter always matches the start of a string, but if Multiline mode is enabled, it *also* matches the start of a line.

In addition to different modes, it is important to be aware that there are two sets of metacharacters: those that only apply when in a normal expression, and a *different* (smaller) set which only apply when inside a character class. (For example, inside a character class the caret means "not", if it is the first character.)

Perhaps the most important metacharacter is the backslash "\" which has the ability to escape *or* create metacharacters by preceeding another character. That is, by using "\^", you are escaping the metacharacter and indicating the literal caret character should be matched at this point, whilst doing "\b" will instead *create* a metacharacter, which in this case tells the regex engine to match a word boundary at that point.

Different regex implementations can have different metacharacters - and some use the same metacharacters but may have subtly different meanings - so it is important to ensure you know what metacharacters to use for the particular regex engine which you are using.

The specific meanings of metacharacters is defined in the pages describing their features, or you can refer to the metacharacter reference chart for an overview.

Positions

When working with regex it is useful to remember that matching can deal with positions as well as (or instead of) matching actual characters. A position is the 'gap' between characters, and is sometimes referred to as a zero-length match.

There are two main ways to match positions - by using pre-defined "boundary" metacharacters, or by using ad-hoc "lookaround" expressions.

Boundaries

Start of input

To match the start of the input text - the position before the first character - you can use either "\A" or "^". The former will only ever match this position, but when multiline mode is enabled, "^" will additionally match start of line position.

End of input

To match the end of the input text - the position after the last character - you can use either "\z" or "$". The former will only ever match the end of input, whilst the latter can also match the end of lines, if multiline mode is enabled.

There is also a "\Z" which almost matches the end of input but will match at the position before a trailing newline, if there is one.

Start of line

When in multiline mode, you can use caret "^" to match the start of the line, which is defined as the position after a newline.

What is considered a newline can be altered with Unix Lines mode, which allows you to include or exclude carriage returns, (however this will only affect the start of line position if there are individual carriage returns, since carriage returns paired with a line-break come at the end of the line, not the start).

With multiline mode disabled, there is no explicit start of line character, though a positive lookbehind can be used, i.e. `(?<=\n)`

End of line

When in Multiline mode you can use dollar "$" to match the end of the line, which is defined as the position before any newline.

Whether carriage returns are considered part of newline can be controlled with the Unix Lines mode which (when enabled) means that only newline character is considered a newline, and the position matched will be *after* any carriage returns that might otherwise be paired with a newline.

With multiline mode disabled, there is no explicit end of line character, though a positive lookahead can be used, i.e. `(?=\n)`

Word Boundary

This is slightly different to what you might expect. It does *not* match whitespace between words (remember, whitespace is characters, and we're dealing in positions), but the "`\b`" word boundary metacharacter is used to match a change between a word character and a non-word character.

There is a word boundary position between the two characters "a-" and also between "-b", but there is not a word boundary between "ab" nor is there one between "--".

Whilst some regex implementations have distinct "start word" and "end word" boundaries, the engine used by cfRegex does not differentiate them. You can workaround this by using lookarounds to immitate start of word `(?<!\w)(?=\w)` and end of word `(?<=\w)(?!\w)`

You can match the opposite of a word boundary using "`\B`", which will match between "ab" *and* between "--" but not between "a-" nor "-b".

Lookarounds

When you need to match an adhoc-position, you can use lookarounds. A lookaround lets you use a sub-expression to indicate the position that can match. For lookaheads you have the full regex syntax available to you. For lookbehinds you can only use limited-width quantifiers (that is, the standard "`*`", "`+`", and variants are unavailable, since they do not have a maximum width).

As you might guess from the name, lookarounds do not actually match anything - they simply *look* at what is ahead or behind and determine if their sub-expression will match or not, and either succeed (and let matching continue) or fail (and the match fails).

Lookarounds can be either positive (their sub-expression must match to succeed), or negative (their sub-expression must not match to success), which - combined with lookahead and lookbehind - gives four different lookarounds in total:

- Positive lookahead: `(?=...)`
- Negative lookahead: `(?!...)`
- Positive lookbehind: `(?<=...)`
- Negative lookbehind: `(?<!...)`

It is useful to remember that - since lookarounds do not consume characters - they can be "stacked" to allow for a combination of conditions (which might be less maintainable if expressed all together), for example you can use "`(?=\w)(?!x)`" to match the position before any word character except the letter "x". As a single lookahead this would need to be "`(?=[A-Za-wyz0-9_])`" which is obviously more long-winded.

Encoded Characters

There are some characters which can be inconvenient, or even impossible, to represent natively in regex. To allow regex to still easily match text involving these characters, there are metacharacters which represent these characters in encoded form.

The most common of these are newline, carriage return, and tab characters. Whilst all of these *can* usually be included as literal characters, it is generally more readable to use their encoded form:

- "`\n`" for **n**ewline
- "`\r`" for carriage **r**eturn
- "`\t`" for **t**ab.

For any other character, there are three ways to represent them, ASCII Hexadecimal, Unicode Hexadecimal, or Octal.

(Although it is available, Octal encoding is not commonly used or known about, and offers no advantages, so it is recommended not to use this.)

ASCII

To encode a character using ASCII values, simply prefix the two-digit hex character code with backslash-x "`\x`".

For example, the character "•" is ASCII decimal 149, which is 95 in hexadecimal, so to use in a regex you would do "`\x95`".

Unicode

To encose a character using Unicode values, use backslash-u "`\u`" followed by the four-digit hex character code.

For example, the unicode character "☺" has the code 263A, so encoded in a regex pattern it would look like "`\u263A`".

Octal

To encode a character with Octal, you use backslash-zero "`\0`" as the prefix, followed by a value between 0 and 377 (255 in decimal).

For example, the character "•" is 225 in octal, so is encoded as "`\0225`" in a regex.

Character Classes

Character classes allow you to specify a set of characters from which a single character will be matched (unless a quantifier is used), and to do this they provide a notation that is significantly shorter than having to alternate between all the possible characters you want to allow.

The character classes notation use an almost completely different set of metacharacters than normal regex syntax, with only five characters that need to be escaped within a class to prevent them being treated as metacharacters.

The only standard regex constructs which apply inside a character class, other than nested classes, is encoded characters. These work in exactly the same way as a literal character inside a class, and so can be used in ranges.

A character class is formed with two brackets `[...]` within which a set of characters is provided, and the following rules apply.

Ranges

A class of "`[abcdef0123456789]`" will match a single character if it is a letter from "a" to "f" or a digit from "0" to "9", but to avoid listing all the letters, ranges can be used.

The above example can be condensed to "`[a-f0-9]`" and will still match the same thing.

This works for *all* characters (not just letters and numbers), based on their assigned numerical value in a character chart, so it is important to be aware of this order. (You can use `charmap` to see character order.)

You can also use encoded characters in ranges, so the ASCII hex encoded "`[\x61-\x7A\x30-\x39]`" is equivalent to the "`[a-f0-9]`" version.

It is especially important to remember that doing "`[A-z]`" will include six characters which exist between "Z" and "a", which is probably not desired, and is can easily be overlooked - so doing it is not recommended even if you do actually want those six characters.

To match a literal hyphen "-" you should prefix it with a backslash "`\-`", though also be aware that if the first or last character in the class is hyphen it is treated as a literal too (since it cannot be a range).

Negative Classes

If the first character in a class is a caret "`^`" it transforms the class into a negative character class.

A negative class works in exactly the same way as a normal non-negated class, except the class represents all the characters that should *not* be matched.

So "`[^0-9]`" will match any character that is not a digit - *any* character, including whitespace, control characters, and so on).

To match a literal caret, either escape with "`\^`" or do not place it as the first character.

Nesting

Character classes can be nested. That is, you can do "`[[a-f][0-9]]`" and it will work (although this example doesn't have any benefit over "`[a-f0-9]`").

By default, classes are combined by union (adding the results together), thus "`[[a-f][^a-c]]`" is *not* equivalent to "`[def]`" but actually means "abcdef OR anything not abc", which results in *any* character being matched.

To combine nested classes with intersection (only the characters common to both classes are used), you can use the special metacharacter "`&&`" between the classes, so "`[[a-f]&&[^a-c]]`" is equivalent to "`[def]`".

Note that you do not need to escape a single ampersand "`&`" because the metacharacter only exists as a double-ampersand (which should not otherwise appear in a class, however you can use "`\&&`" if you do somehow have a double-ampersand to be escaped).

Escaping

To include a "`\`", "`[`" or "`]`" inside a character class, they always need to be escaped as "`\\`", "`\[`" and "`\]`" respectively.

When a hypen "`-`" is not first or last in a class, it must be escaped as "`\-`", and it is recommended to always manually escape for greater maintainability.

If a caret "`^`" is the first character in a class, it creates a negative class, unless it is escaped with "`\^`". A caret that is not at the start of a class does not need to be escaped.

In certain situations, "`&&`" is a metacharacter, but a single ampersand does not need escaping. Similarly, "`{`" and "`}`" can occur as *part* of a metacharacter but do not themselves need escaping.

No error is returned from over-escaping inside a class, it simply reduces readability and may confuse people new to regex.

In summary, only the five following characters must be escaped to match their literal values inside a class: `[ ^ - \ ]`

Shorthand Classes

As you might imagine, there are a number of classes which would be used more frequently than others, and so these classes have shorthand notation to simplify patterns that use them.

- Instead of "`[0-9]`" you can use "`\d`" for a **d**igit".
- Instead of "`[A-Za-z0-9_]`" you can use "`\w`" for a **w**ord character.
- Instead of "`[\r\n\t ]`" you can use "`\s`" for a white**s**pace character.

These three shortcuts all have negated character class variants too:

- Instead of "`[^0-9]`" you can use "`\D`" for a non-digit.
- Instead of "`[^A-Za-z0-9_]`" you can use "`\W`" for a non-word character.
- Instead of "`[^\r\n\t ]`" you can use "`\S`" for a non-whitespace character.

Since character classes can be nested, you can also nest these shorthand classes, so to match hexidecimal digits you can do: "`[\dA-F]`" which is equivalent to doing "`[[0-9][A-F]]`".

(The `\s` class technically includes two other characters ASCII 11 (vertical tab) and ASCII 12 (form feed) which are also considered whitespace, but generally are not used any more, so are not listed above to avoid unnecessary complexity.)

Other Predefined Classes

In addition to the basic shorthand classes listed above, there are a couple of sets of other convenience classes defined, POSIX-compatible classes and Unicode category classes.

Both of these can be referenced using "`\p{Code}`" for the normal class, and "`\P{Code}`" for the negated variant.

For full details of what codes are available and the characters they represent, see the individual POSIX-compatible classes and Unicode category classes pages.

POSIX-compatible classes

These take the form of "`\p{Keyword}`" and "`\P{Keyword}`" for normal and negated variants.

Code	Description	Represents	Notes
Alpha	any letter	`[a-zA-Z]`	
Upper	uppercase letters	`[A-Z]`	
Lower	lowercase letters	`[a-z]`	
Digit	decimal digit	`[0-9]`	
Alnum	alphanumeric	`[a-zA-Z0-9]`	this is *not* the same as `\w` since it excludes "_"
XDigit	hexadecimal digit	`[a-fA-F0-9]`	
Punct	punctuation	`[!"#$%&'()*+,-./:;<=>?@\[\\\]\^_`|{}~]`	
Graph	graphic character	`[\p{Alnum}\p{Punct}]`	
Print	printable character	`[\x20-\x7E]`	Graph and space (excludes newlines and tab).
Blank	tab or space	`[ \t]`	
Space	any whitespace	`\s`	
Cntrl	any control character	`[\x00-\x1F\x7F]`	

These keywords are case-sensitive - i.e. "`\p{alpha}`" will cause an error.

Note also that the "`[:alpha:]`" syntax used by other regex implementations does *not* work.

Unicode Category Classes

Unicode defines a number of "categories", which can be referenced with "`\p{Code}`" and "`\P{Code}`", using either one or two letter codes to represent which category of characters they belong to.

For details of which characters are matched, consult the documentation for java.lang.Character or the Unicode Category details.

Code	Description
C	all control chars
Cc	cntrl
Cf	format
Cn	unassigned
Co	private use
Cs	surrogate
L	all letters
L1	Latin-1
LD	letter or digit
Ll	lowercase letter
Lm	modifier letter
Lo	other letter
Lt	titlecase letter
Lu	uppercase letter
M	all mark
Mc	combining spacing mark
Me	enclosing mark
Mn	non spacing mark
N	all numbers
Nd	decimal digit number
Nl	letter number
No	other number
P	all punctuation
Pc	connector punctuation
Pd	dash punctuation
Pe	end punctuation
Po	other punctuation
Ps	start punctuation
S	all symbols
Sc	currency symbol
Sk	modifier symbol
Sm	math symbol
So	other symbol
Z	all separators
Zl	line separator
Zp	paragraph separator
Zs	space separator

Quantifiers

A quantifier is something that tells the regex engine to repeat the previous item a number (quantity) of times, according to one of three behaviours (greedy, lazy, or possessive).

Numeric Quantifiers

A numeric quantifier is delimited by `{` and `}` metacharacters, and can repeat an exact number of times, a minimum number, or a range of repetitions.

- To match an exact number, use "`{a}`" where "a" is any positive integer.
- To match a minimum (with no upper limit), use "`{a,}`"
- To match a range, use "`{a,b}`" where "b" is a positive integer greater than "a".

Shorthand Quantifiers

In regex, there are a number of quantifiers used commonly enough to justify having their own shorthand notation.

- To match 0 or 1 times, use "`?`" instead of "`{0,1}`"
- To match 0 or more times, use "`*`" instead of "`{0,}`"
- To match 1 or more times, use "`+`" instead of "`{1,}`"

Whilst these shorthands are useful, it is important not to get carried away and over-use them. If there is a known range of repetition it may well result in greater readability and/or performance to explicitly specify the range.

Behaviours

Greedy

The default behaviour for a quantifier is to start by matching as much as possible, and only to relinquish characters if required to do so by backtracking (when the following instruction is unable to match).

Since greedy is the default behaviour, all quantifiers are greedy unless they are converted to lazy or possessive.

Whilst the traditional way of explaining quantifiers, for example "`+`", is to say "match one or more times", it better represents the behaviour of greedy quantifiers to say "match as many as possible, at least once".

Lazy

If a quantifier is suffixed with `?` then it becomes a lazy quantifier which will match as little as required, and only add more characters to its match if required to do so (again, because the following instruction does not yet match).

Lazy quantifiers are "`??`", "`*?`", "`+?`", "`{a}?`", "`{a,}?`", "`{a,b}?`".

Whilst lazy quantifiers can be useful, a lot of times it is better to use a greedy (or possessive) quantifier combined with a negative character class, e.g. "`[^x]+x`" instead of "`.+?x`", but this will depend on exactly what is being matched.

Possessive

A quantifier can be suffixed with `+` to make it posessive. This is identical to a greedy quantifier, except it will not backtrack within itself - even if to do so would allow the overall expression to match, a possessive quantifier is all or nothing.

Possessivenes only make sense for quantifiers that match a variable number of characters. If matching an exact number (e.g. `{4}`), or if matching one or zero (i.e. `?`) there is no difference between possessive and greedy.

Possessive quantifiers are "`*+`", "`++`", "`{a,}+`", "`{a,b}+`".

Whilst incorrect use of possessive quantifiers can cause an expression not to match (and it might not be immediately obvious why), they are also an important feature which can help to improve performance by preventing unwanted backtracking.

Grouping

Groups are useful when you have a sub-expression that should be treated as a single unit, so that it can either be captured or repeated.

There are three different types of groups: capturing group, non-capturing group, and atomic group.

Capturing Groups

A capturing group is where the contents of a group are stored, and can be used as a backreference within the expression, or returned to be acted upon outside of the regex (such as in a replacement string or function).

To create a capturing group, simply enclose the sub-expression with parentheses:

```
(captured group)
```

Capturing groups can be nested - their capture number is counted based on the position of their opening parenthesis, and captured content includes that of any enclosed groups. That is, "`(a(b)(c))((d)e)`" results in the five captured values of "abc","b","c","de","d".

Backreferences

When you want to refer to the value of a captured group, you use what is known as a backreference, which is the group number preceeded by a backslash. So for group 1 you do `\1`, for group 2 you do `\2` and so on. It is possible to have over a hundred groups, but it is *not* recommended to actually use this many - if you have a regex with more than a dozen captured groups then you should consider if there is a better way to do whatever you are doing.

It is important to remember that a backreference is equivalent to the literal text which was captured by the group, not the instructions within in. (For example, "`([abc])\1`" will match "aa" or "bb" or "cc", but not "ab" or anything else.)

Non-Capturing Groups

When you do not need the value of a group, but simply want to act upon it as a single item, you should use a non-capturing group.

```
(?:non-capturing group)
```

You can also combine a non-capturing group with a mode flag, to apply a particular regex mode only to the expression within the group.

For example, if there is a place you need dot to match newline, but not for the whole expression, then "`(?s:.)`" could be used.

Alternatively, you might have an expression which is case-insensitive, expect for one small part, "`(?-i:CASE IMPORTANT)`" is a way to do that.

Non-capturing groups with flags can still also be used for repetition.

Atomic Groups

Atomic groups are also non-capturing but they go a step further than simply treating a sub-expression as a single item - they prevent the regex engine from backtracking inside the group (whilst a normal non-atomic group allows backtracking to re-evaluate its contents).

This is an advanced feature that can help improve performance, but you should fully understand what backtracking is - when you want it and when you don't - before attempting to use atomic groups.

```
(?>atomic group)
```

Alternation

When your regex has multiple alternative paths that can be matched, you can use alternation to enable each of the paths to be tried as required. This is done by providing a pipe "`|`" delimited list of options, such as "`this|that|other`".

Since generally you only want part of a regex to have different paths, alternation is usually done inside either groups or lookarounds, both of which limit the scope of the alternatives to within their paretheses.

It is important to remember that the regex engine will not try each alternative if it does not need to - it will try the first, and see if the rest of the pattern matches. If it does not, the regex engine will backtrack to the start of the sub-expression and try the next alternative.

A common mistake is to try and use the "`|`" inside a character class, this is not necessary, and will simply allow the pipe character to be matched.