

Regular Expressions

This appendix explains regular expressions and how to use them in Cisco IOS software configurations. It also provides details for composing regular expressions. This appendix has the following sections:

- General Concepts
- Using Regular Expressions
- Creating Regular Expressions
- Practical Examples

General Concepts

A regular expression is a pattern to match against an input string. You specify the pattern that a string must match when you compose a regular expression. Matching a string to the specified pattern is called “pattern matching.” Pattern matching either succeeds or fails.

For example, you can specify in an X.25 routing table that incoming packets with destination addresses beginning with 3107 are routed to serial interface 0. In this example, the pattern to match is the 3107 specified in the X.25 routing table. The string is the initial portion of the destination address of any incoming X.25 packet. When the destination address string matches 3107 pattern, then pattern matching succeeds and the Cisco IOS software routes the packet to serial interface 0. When the initial portion of the destination address does not match 3107, then pattern matching fails and the software does not route the packet to serial interface 0.

If a regular expression can match two different parts of an input string, it will match the earliest part first.

Using Regular Expressions

Cisco configurations use several implementations of regular expressions. Generally, you use regular expressions in the following ways:

- To specify chat scripts for asynchronous lines in the dial-on-demand routing (DDR) feature
- To specify routes in a routing table for the X.25 switching feature
- To filter packets and routing information in the DECnet and BGP protocols

Specifying Chat Scripts

On asynchronous lines, chat scripts send commands for modem dialing and logging in to remote systems. You use a regular expression in the **script dialer** command to specify the name of the chat script that the Cisco IOS software is to execute on a particular asynchronous line. You can also use regular expressions in the **dialer map** command to specify a “modem” script or “system” script to be used for a connection to one or multiple sites on an asynchronous interface.

Specifying Routes in a Routing Table

As described in the “General Concepts” section, you can use regular expressions to help specify routes in an X.25 routing table. When you create entries in an X.25 routing table, you can use regular expressions in the **x25 route** command to help specify routes for incoming calls. When a router or access server receives an incoming call that should be forwarded to its destination, the Cisco IOS software consults the X.25 routing table to determine the route. The software compares the X.121 network interface address (or destination address) field and the Call User Data (CUD) field of the incoming packet with the routing table to determine the route. When the destination address and the CUD of the incoming packet match the X.121 and CUD regular expressions you specified in the routing table, the access server or router forwards the call.

Filtering Packets and Routing Information

You can use regular expressions in access lists for both DECnet and Border Gateway Protocol (BGP). In DECnet, you can use regular expressions in the **access-list** command to filter *connect initiate* packets. With these packets, you can filter packets by DECnet object type, such as MAIL. In BGP, you use regular expressions in the **ip as-path access-list** command for path filtering by neighbor. Using regular expressions, you specify an access list filter on both incoming and outbound updates based on the BGP autonomous system paths.

Creating Regular Expressions

A regular expression can be a single-character pattern or a multiple-character pattern. That is, a regular expression can be a single character that matches the same single character in the input string or multiple characters that match the same multiple characters in the input string. This section describes creating both single-character patterns and multiple-character patterns. It also discusses creating more complex regular expressions using multipliers, alternation, anchoring, and parentheses.

Single-Character Patterns

The simplest regular expression is a single character that matches itself in the input string. For example, the single-character regular expression **3** matches a corresponding **3** in the input string. You can use any letter (A–Z, a–z) or number (0–9) as a single-character pattern. The following examples are single-character regular expression patterns:

A
k
5

You can use a keyboard character other than a letter or a number—such as an exclamation point (!) or a tilde (~)—as a single-character pattern, but certain keyboard characters have special meaning when used in regular expressions. Table 155 lists the keyboard characters with special meaning.

Table 155 Characters with Special Meaning

Character	Special Meaning	
period	.	Matches any single character, including white space.
asterisk	*	Matches 0 or more sequences of the pattern.
plus sign	+	Matches 1 or more sequences of the pattern.
question mark	?	Matches 0 or 1 occurrences of the pattern.
caret	^	Matches the beginning of the input string.
dollar sign	\$	Matches the end of the input string.
underscore	_	Matches a comma (,), left brace ({}), right brace (}), the beginning of the input string, the end of the input string, or a space.
brackets	[]	Designates a range of single-character patterns.
hyphen	-	Separates the end points of a range.
parentheses	()	(Border Gateway Protocol (BGP) specific) Designates a group of characters as the name of a confederation.

To use these special characters as single-character patterns, remove the special meaning by preceding each character with a backslash (\). The following examples are single-character patterns matching a dollar sign, an underscore, and a plus sign, respectively:

```
\$
```

```
\_
```

```
\+
```

You can specify a range of single-character patterns to match against a string. For example, you can create a regular expression that matches a string containing one of the following letters: *a*, *e*, *i*, *o*, and *u*. One and only one of these characters must exist in the string for pattern matching to succeed. To specify a range of single-character patterns, enclose the single-character patterns in square brackets ([]). The order of characters within the brackets is not important. For example, `[aeiou]` matches any one of the five vowels of the lowercase alphabet, while `[abcdABCD]` matches any one of the first four letters of the lowercase or uppercase alphabet.

You can simplify ranges by typing only the end points of the range separated by a dash (—). Simplify the previous range as follows:

```
[a-dA-D]
```

To add a hyphen as a single-character pattern in your range, include another hyphen and precede it with a backslash:

```
[a-dA-D\-]
```

You can also include a right square bracket (]) as a single-character pattern in your range. To do so, enter the following:

```
[a-dA-D\-\]]
```

The previous example matches any one of the first four letters of the lower- or uppercase alphabet, a hyphen, or a right square bracket.

You can reverse the matching of the range by including a caret (^) at the start of the range. The following example matches any letter *except* the ones listed:

`[^a-dqsv]`

The following example matches anything except a right square bracket (]) or the letter *d*:

`[^\]d]`

Multiple-Character Patterns

When creating regular expressions, you can also specify a pattern containing multiple characters. You create multiple-character regular expressions by joining letters, numbers, or keyboard characters that do not have special meaning. For example, `a4%` is a multiple-character regular expression. Precede keyboard characters that have special meaning with a backslash (\) when you want to remove their special meaning.

With multiple-character patterns, order is important. The regular expression `a4%` matches the character *a* followed by the number *4* followed by a *%* sign. If the input string does not have *a4%*, in that order, pattern matching fails. The multiple-character regular expression `a.` uses the special meaning of the period character (.) to match the letter *a* followed by any single character. With this example, the strings *ab*, *a!*, or *a2* are all valid matches for the regular expression.

You can remove the special meaning of the period character by preceding it with a backslash. In the expression `a\.` only the string *a.* matches the regular expression.

You can create a multiple-character regular expressions containing all letters, all digits, all special keyboard characters, or a combination of letters, digits, and other keyboard characters. The following examples are all valid regular expressions:

`telebit`

`3107`

`v32bis`

Multipliers

You can create more complex regular expressions that instruct the Cisco IOS software to match multiple occurrences of a specified regular expression. To do so, you use some special characters with your single- and multiple-character patterns. Table 156 lists the special characters that specify “multiples” of a regular expression.

Table 156 Special Characters Used as Multipliers

Character	Description
asterisk	* Matches 0 or more single- or multiple-character patterns.
plus sign	Matches 1 or more single- or multiple-character patterns.
question mark	? Matches 0 or 1 occurrences of the single- or multiple-character pattern.

The following example matches any number of occurrences of the letter *a*, including none:

`a*`

The following pattern requires that at least one letter *a* be present in the string to be matched:

a+

The following pattern matches the string *bb* or *bab*:

ba?b

The following string matches any number of asterisks (*):

To use multipliers with multiple-character patterns, enclose the pattern in parentheses. In the following example, the pattern matches any number of the multiple-character string *ab*:

(ab)*

As a more complex example, the following pattern matches one or more instances of alphanumeric pairs (but not none; that is, an *empty string* is not a match):

([A-Za-z][0-9])+

The order for matches using multipliers (*, +, or ?) is longest construct first. Nested constructs are matched from outside to inside. Concatenated constructs are matched beginning at the left side of the construct. Thus, the regular expression matches *A9b3*, but not *9Ab3* because the letter appears first in the construct.

Alternation

Alternation allows you to specify alternative patterns to match against a string. You separate the alternative patterns with a vertical bar (|). Exactly one of the alternatives can match the input string. For example, the regular expression **codex|telet** matches the string *codex* or the string *telet*, but not both *codex* and *telet*.

Anchoring

You can instruct the Cisco IOS software to match a regular expression pattern against the beginning or the end of the input string. That is, you can specify that the beginning or end of an input string contain a specific pattern. You “anchor” these regular expressions to a portion of the input string using the special characters shown in Table 157.

Table 157 Special Characters Used for Anchoring

Character	Description
^	Matches the beginning of the input string.
\$	Matches the end of the input string.

Note another use for the ^ symbol. As an example, the following regular expression matches an input string only if the string starts with *abcd*:

^abcd

Whereas the following expression is a range that matches any single letter, as long as it is not the letters *a*, *b*, *c*, or *d*:

[^abcd]

With the following example, the regular expression matches an input string that ends with *.12*:

\$.12

Contrast these anchoring characters with the special character underscore (`_`). Underscore matches the beginning of a string (`^`), the end of a string (`$`), space (), braces (`{ }`), comma (`,`), or underscore (`_`). With the underscore character, you can specify that a pattern exist anywhere in the input string. For example, `_1300_` matches any string that has `1300` somewhere in the string. The string's `1300` can be preceded by or end with a space, brace, comma, or underscore. So, while `{1300_}` matches the regular expression, `21300` and `13000` do not.

Using the underscore character, you can replace long regular expression lists. For example, you can replace the following list of regular expressions with simply `_1300_`:

```
^1300$  
^1300(space)  
(space)1300  
{1300,  
,1300,  
{1300}  
,1300,
```

Parentheses for Recall

As shown in the “Multipliers” section, you use parentheses with multiple-character regular expressions to multiply the occurrence of a pattern. You can also use parentheses around a single- or multiple-character pattern to instruct the IOS software to remember a pattern for use elsewhere in the regular expression.

To create a regular expression that recalls a previous pattern, you use parentheses to instruct memory of a specific pattern and a backslash (`\`) followed by an integer to reuse the remembered pattern. The integer specifies the occurrence of a parentheses in the regular expression pattern. If you have more than one remembered pattern in your regular expression, then `\1` uses the first remembered pattern and `\2` uses the second remembered pattern, and so on.

The following regular expression uses parentheses for recall:

```
a(.)bc(.)\1\2
```

This regular expression matches the letter `a` followed by any character (call it character #1) followed by `bc`, followed by any character (character #2), followed by character #1 again, followed by character #2 again. In this way, the regular expression can match `aZbcTZT`. The software identifies character #1 as `Z` and character #2 as `T`, and then uses `Z` and `T` again later in the regular expression.

The parentheses do not change the pattern; they only instruct the software to recall that part of the matched string. The regular expression `(a)b` still matches the input string `ab`, and `^3107` still matches a string beginning with `3107`, but now the Cisco IOS software can recall the `a` of the `ab` string and the starting `3107` of another string for use later.

Practical Examples

This section shows you practical examples of regular expressions. The examples correspond with the various ways you can use regular expressions in your configurations.

Specifying Chat Scripts Example

The following example uses regular expressions in the **chat-script** command to specify chat scripts for lines connected to Telebit and U.S. Robotics modems. The regular expressions are **telebit.*** and **usr.***. When the chat script name (the string) matches the regular expression (the pattern specified in the command), then the Cisco IOS software uses that chat script for the specified lines. For lines 1 and 6, the Cisco IOS software uses the chat script named *telebit* followed by any number of occurrences (*) of any character (.). For lines 7 and 12, the software uses the chat script named *usr* followed by any number of occurrences (*) of any character (.).

```
! Some lines have Telebit modems
line 1 6
chat-script telebit.*
! Some lines have US Robotics modems
line 7 12
chat-script usr.*
```

X.25 Switching Feature Example

In the following X.25 switching feature example, the **x25 route** command causes all X.25 calls to addresses whose first four Data Network Identification Code (DNIC) digits are 1111 to be routed to serial interface 3. Note that the first four digits (^1111) are followed by a regular expression pattern that the Cisco IOS software is to remember for use later. The \1 in the rewrite pattern recalls the portion of the original address matched by the digits following the 1111, but changes the first four digits (1111) to 2222.

```
x25 route ^1111(.*) substitute-dest 2222\1 interface serial 3
```

DECnet Access List Example

In the following DECnet example, the regular expression is **^SYSTEM\$**. The access list permits access to all connect initiate packets that match the access identification of SYSTEM.

```
access-list 300 permit 0.0 63.1023 eq id ^SYSTEM$
```

BGP IP Access Example

The following BGP example contains the regular expression **^123.***. The example specifies that BGP neighbor with IP address 128.125.1.1 is not sent advertisements about any path through or from the adjacent autonomous system 123.

```
ip as-path access-list 1 deny ^123 .*

router bgp 109
network 131.108.0.0
neighbor 129.140.6.6 remote-as 123
neighbor 128.125.1.1 remote-as 47
neighbor 128.125.1.1 filter-list 1 out
```

