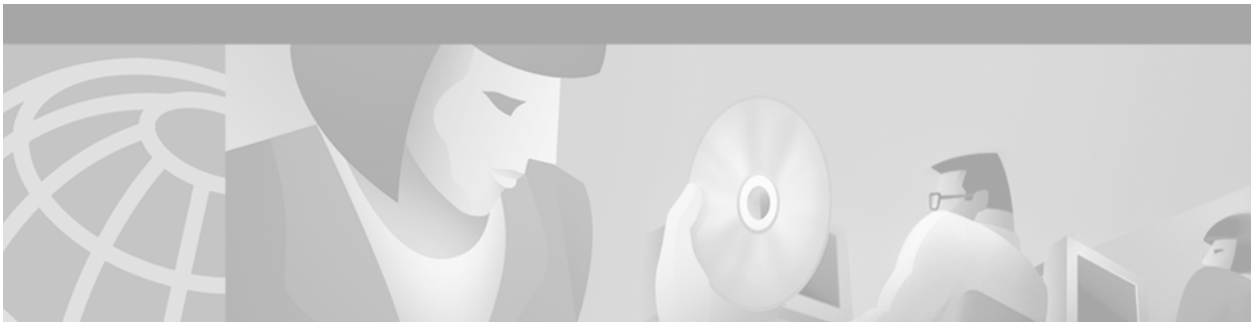

Regular Expressions

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

General Concepts About Regular Expressions

A regular expression is entered as part of a command and is a pattern made up of symbols, letters, and numbers that represent an input string for matching (or sometimes not matching). Matching the string to the specified pattern is called *pattern matching*.

Pattern matching either succeeds or fails. If a regular expression can match two different parts of an input string, it will match the earliest part first.

Cisco configurations uses regular expression pattern matching in several implementations. The following is a list of some of these implementations:

- BGP IP AS-path and X.29 access lists
- Modem (or chat) and system scripts
- X.25 route substitute destination feature
- Protocol translation ruleset scripts

Cisco Regular Expression Pattern Matching Characters

Table 25 summarizes the basic Cisco IOS regular expression characters and their functions.

Table 25 Cisco Regular Expression Characters

Regular Expression Character	Function	Examples
.	Matches any single character.	0.0 matches 0x0 and 020 t..t matches strings such as test, text, and tart
\	Matches the character following the backslash. Also matches (escapes) special characters.	172\.1\.. matches 172.1.10.10 but not 172.12.0.0 \. allows a period to be matched as a period
[ ]	Matches the characters or a range of characters separated by a hyphen, within left and right square brackets.	[02468a-z] matches 0, 4, and w, but not 1, 9, or K
^	Matches the character or null string at the beginning of an input string.	^123 matches 1234, but not 01234
?	Matches zero or one occurrence of the pattern. (Precede the question mark with Ctrl-V sequence to prevent it from being interpreted as a help command.)	ba?b matches bb and bab
$	Matches the character or null string at the end of an input string.	123$ matches 0123, but not 1234
*	Matches zero or more sequences of the character preceding the asterisk. Also acts as a wildcard for matching any number of characters.	5* matches any occurrence of the number 5 including none 18\..* matches the characters 18. and any characters that follow 18.
+	Matches one or more sequences of the character preceding the plus sign.	8+ requires there to be at least one number 8 in the string to be matched
() []	Nest characters for matching. Separate endpoints of a range with a dash (-).	(17)* matches any number of the two-character string 17 ([A-Za-z][0-9])+ matches one or more instances of letter-digit pairs: b8 and W4, as examples

Table 25 Cisco Regular Expression Characters (continued)

Regular Expression Character	Function	Examples
|	Concatenates constructs. Matches one of the characters or character patterns on either side of the vertical bar.	A(B|C)D matches ABD and ACD, but not AD, ABCD, ABBD, or ACCD
_	Replaces a long regular expression list by matching a comma (,), left brace ({), right brace (}), the beginning of the input string, the end of the input string, or a space.	The characters _1300_ can match any of the following strings: ^1300$ ^1300space space1300 {1300, ,1300, {1300} ,1300,

The order for matching using the * or + character is longest construct first. Nested constructs are matched from the outside in. Concatenated constructs are matched beginning at the left side. If a regular expression can match two different parts of an input string, it will match the earliest part first.

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 to Z, a to z) or number (0 to 9) as a single-character pattern. You can use also 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 not the characters listed in Table 25 that have special meaning when used in regular expressions.

To use the characters listed in Table 25 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 endpoints of the range separated by a hyphen (-). 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 lowercase or uppercase alphabet, a hyphen, or a right square bracket.

You can reverse the matching of the range by including a caret (^) sign 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!, and a2 are all valid matches for 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.

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.

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 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])+
```

Alternation

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

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.

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.

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 Cisco 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 asT, 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.

Regular Expression Examples

This section provides the following practical examples of regular expression use:

- Regular Expression Pattern Matching in Access List Example
- Regular Expression Pattern Matching in Scripts Example
- Regular Expression Pattern Matching in X.25 Routing Entries Example
- Regular Expression Pattern Matching in a Protocol Translation Ruleset Example

Regular Expression Pattern Matching in Access List Example

Both the BGP IP autonomous system path feature and X.29 access list configuration statements can use regular expression patterns to match addresses for allowing or denying access.

The following BGP example contains the regular expression ^123.*. The example specifies that the BGP neighbor with IP address 172.23.1.1 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 172.18.0.0
neighbor 172.19.6.6 remote-as 123
neighbor 172.23.1.1 remote-as 47
neighbor 10.125.1.1 filter-list 1 out
```

The following example uses the regular expression string ^4$ to configure the router to receive the routes originated from only autonomous system 4:

```
ip as-path access-list 1 permit ^4$

router bgp 1
 neighbor 4.4.4.4 remote-as 4
 neighbor 4.4.4.4 route-map foo in

route-map foo permit 10
 match as-path 1
```

An X.29 access list can contain any number of access list items. The list items are processed in the order in which they are entered, with the first regular expression pattern match causing the permit or deny condition. The following example permits connections to hosts with addresses beginning with the string 31370:

```
x29 access-list 2 permit ^31370
```

Regular Expression Pattern Matching in Scripts Example

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.

The following example uses regular expressions telebit.* and usr.* to identify chat scripts for Telebit and US Robotics modems. 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.*
```

If you adhere to a chat script naming convention of the form [modem-script *modulation-type] in the **dialer map** command, .*-v32bis for example, this allows you to specify the modulation type that is best for the system you are calling, and allows the modem type for the line to be specified by the modem **chat-script** command.

The following example shows the use of chat scripts implemented with the *system-script* and *modem-script* options of the **dialer map** command. If there is traffic for IP address 10.2.3.4, the router will dial the 91800 number using the usrobotics-v32 script, matching the regular expression in the modem chat script. Then the router will run the unix-slip chat script as the system script to log in. If there is traffic for 10.3.2.1, the router will dial 8899 using usrobotics-v32, matching both the modem script and modem chat script regular expressions. The router will then log in using the cisco-compressed script.

```
! Script for dialing a usr v.32 modem:
chat-script usrobotics-v32 ABORT ERROR "" "AT Z" OK "ATDT \T" TIMEOUT 60 CONNECT \c
!
! Script for logging into a UNIX system and starting up SLIP:
chat-script unix-slip ABORT invalid TIMEOUT 60 name: billw word: wewpass ">" "slip
default"
```

```
!
! Script for logging into a Cisco access server and starting up TCP header compression:
chat-script cisco-compressed...
!
line 15
 script dialer usrobotics-*
!
interface async 15
 dialer map ip 10.2.3.4 system-script *-v32 system-script cisco-compressed 91800
 dialer map ip 10.3.2.1 modem-script *-v32 modem-script cisco-compressed 91800
```

Regular Expression Pattern Matching in X.25 Routing Entries Example

The **x25 route** command is used to create an entry in the X.25 routing table that the router consults to learn where to forward incoming calls and place outgoing packet assembler/disassembler (PAD) or protocol translation calls. Regular expressions are used with the **x25 route** command to allow pattern-matching operations on the addresses and user data. A common operation is to use prefix matching on the X.121 Data Network Identification Code (DNIC) field and route accordingly. The caret sign anchors the match to the beginning of the pattern.

In the following example, the **x25 route** command causes all X.25 calls to addresses whose first four 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
```

The following example routes any incoming calls that begin with 2222 to the specified data-link connection identifier (DLCI) link.

```
x25 route ^2222 interface serial 1 dlci 20
```

The following example uses the regular expression ^ (carat) character to prevent (clear) X.25 routing for calls that do not specify a source address.

```
x25 route source ^$ clear
```

Regular Expression Pattern Matching in a Protocol Translation Ruleset Example

Note Protocol translation rulesets are supported only in Cisco IOS Release 12.3(8)T and later software.

Regular expressions for the Protocol Translation Ruleset feature have two uses: They match a text string against a defined pattern, and they can use information from a defined regular expression match operation to create a different string using substitution. These operations are performed by combining the characters described in Table 25 with commands from the translate ruleset configuration mode.

To understand regular expression pattern matching, begin by using Table 25 to interpret the following regular expression statement to match a string starting with the characters 172.18.:

^172\.18\..*

The following regular expression statement matches a five-digit number starting with 10 or 11:

```
^1[0-1]...$
```

Consider the following set of actions in a ruleset named B. This ruleset listens for incoming Telnet connections from a particular IP address and port number but ignores (skips) others, decides which PAD destination address the matched incoming connections should be connected to, then finally sets the PAD connection's X.25 VC idle timer from the first digit of the port number.

```
translate ruleset B from telnet to pad
match dest-addr ^10.2.2.(..)$ dest-port ^20..$
skip dest-addr ^10.2.2.11$
set pad dest-addr 4444
substitute telnet dest-port ^200(.)$ into pad idle \1
```

The caret sign anchors a match to the beginning of a string, in this example, 10.2.2 for the destination address and 20 for the destination port.

The parentheses are a powerful tool for the regular expression match operation because they identify groups of characters needed for a substitution. Combined with the substitute...into statement, the parentheses can dynamically create a broad range of string patterns and connection configurations.

In the example, the periods in the parentheses pair can be thought of as placeholders for the characters to be substituted. The dollar sign anchors the substitution match to the end of a string. The backslash preceding the number makes it a literal setting, so no substitution will be done to the idle timer setting.

The **test translate ruleset** command tests the script, and for the previous example would provide a report like the following:

```
Translate From: Telnet 10.2.2.10 Port 2000
        To:   PAD 4444
        Ruleset B
        0/1 users active
```

Consider the following, more complex expression:

```
^172\.18\.(10)\.(.*)$.
```

This expression matches any string beginning with 172.18. and identifies two groups, one that matches 10 and the other that matches a wildcard character.

Let us say that the regular expression ^172\.18\.(10)\.(.*)$ matched the characters 172.18.10.255 from an incoming connection. Once the match is made, the software places the character groups 10 and 255 into buffers and writes the matched groups using a substitution expression.

Regular expression substitution into the expression 0001172018\1\2 would generate the string 000117201810255.

The regular expression \0 would write the entire matched string, and substitution into the expression 0001\0 would generate the string 0001172.18.10.255.