

Regular Expressions

This appendix describes regular expressions. You can use regular expressions in addresses and fields for the X.25 switching feature, in Chat scripts for the dial-on-demand routing (DDR) feature, and in access lists for BGP and DECnet.

Regular expressions allow pattern-matching operations on strings contained in commands; for example, wide ranges of X.121 addresses and Call User Data fields. If you are familiar with regular expressions from UNIX programs such as *regexp*, you are already familiar with much of our regular expression implementation.

A regular expression is a formula for generating a set of strings. If a particular string can be generated by a given regular expression, then that string and regular expression *match*. In many ways, a regular expression is a program, and the regular expression matches the strings it generates.

A regular expression is a branch or any number of branches separated by a vertical bar (`|`). A string is said to match the regular expression if it is generated by the “program” specified in any of the branches. Of course, a string can be generated by more than one branch. For example, `abc` is generated by all branches in the following regular expression.

```
abc|a*(bc)+|(ab)?c.*
```

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

The regular expression support and the technical information for this portion of the documentation is based on Henry Spencer’s public domain *rgrep(3)* library package.

A regular expression is built up of different components, each of which is used to build the regular expression string-generating program. The possible components of a regular expression are as follows:

- Range
- Atom
- Pieces
- Branch

Ranges

A range is a sequence of characters contained within left and right square brackets ([]). A character matches a range if that character is contained within the range; for example, the following syntax forms the range consisting of the characters a, q, c, s, b, v, and d. The order of characters is usually not important; however, there are exceptions and these will be noted.

```
[aqcsbvd]
```

You can specify an ASCII sequence of characters by specifying the first and last characters in that sequence, and separating them with a hyphen (-).

```
[a-dqsv]
```

The above example could also be written so as to specify right square brackets (]) as a character in a range. To do so, enter the bracket as the *first* character after the initial left square bracket that starts the range.

The following example matches a right bracket and the letter d:

```
[ ]d]
```

To include a hyphen (-), enter it as either the first or the last character of the range.

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. When using the caret with the special rules for including a bracket or hyphen, make the caret the very first character.

```
[^a-dqsv]
```

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

```
[^]d]
```

Atoms

Atoms are the most primitive usable part of regular expressions. An atom can be as simple as a single character. The letter a is an atom, for example. It is also a very simple regular expression, that is, a program that generates only one string, which is the single-letter string made up of the letter a. While this may seem trivial, it is important to understand the set of strings that your regular expression program generates. As will be seen in upcoming explanations and examples, much larger sets of strings can be generated from more complex regular expressions.

Certain characters have a special meaning when used as atoms; refer to Table C-1.

Table C-1 Special Symbols Used as Atoms

Character	Description
.	Matches any single character.
^	Matches the beginning of the input string.
\$	Matches the end of the input string.
\c character	Matches <i>character</i> .
_	Matches a comma(,), left brace ({}), right brace (}), the beginning of the input string, the end of the input string, or a space.

Note another use for the ^ symbol.

As an example, the regular expression matches `abcd` only if `abcd` starts the full string to be matched:

`^abcd`

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

`[^abcd]`

It was previously stated that a single character string such as the letter a is an atom. A character by itself, such as \$, means “match the end of the input string.”

\$

Whereas this atom matches a dollar sign (\$). Preceding a character with a backslash (\) removes the special meaning of that character:

`\$`

Any character can be preceded with the backslash character with no adverse effect:

`\a`

Atoms are also full regular expressions surrounded by parentheses. For example, both `a` and `(a)` are atoms matching the letter a. This will be important later, as we see patterns to manipulate entire regular expressions.

Pieces

A piece is an atom optionally followed by one of the symbols listed in Table C-2:

Table C-2 Special Symbols Used with Pieces

Character	Description
*	Matches 0 or more sequences of the atom.
+	Matches 1 or more sequences of the atom.
?	Matches the atom or the null string.

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

a*

The following string requires there to be at least one letter a in the string to be matched:

a+

The following string means that the letter a can be there *once*, but it does not have to be:

a?

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

The following is an example using parentheses. This string matches any number of the two-atom string ab:

(ab)*

As a more complex example, the following string 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 the optional *, +, or ? symbols is longest construct first. Nested constructs are matched from outside to inside. Concatenated constructs are matched beginning at the left side of the construct.

Branch

A branch is simply a set of zero or more concatenated pieces. The previous alphanumeric example was an example of a branch as concatenated pieces. Branches are matched in the order normally read—from left to right. For example, in the previous example, the regular expression matches A9b3, but not 9Ab3 because the alphabet is given first in the two-atom branch of [A-Za-z][0-9].