Next: Common Commands, Previous: Addresses, Up: sed Programs

3.3 Overview of Regular Expression Syntax

To know how to use `sed`, people should understand regular expressions (*regexp* for short). A regular expression is a pattern that is matched against a subject string from left to right. Most characters are *ordinary*: they stand for themselves in a pattern, and match the corresponding characters in the subject. As a trivial example, the pattern

```
The quick brown fox
```

matches a portion of a subject string that is identical to itself. The power of regular expressions comes from the ability to include alternatives and repetitions in the pattern. These are encoded in the pattern by the use of *special characters*, which do not stand for themselves but instead are interpreted in some special way. Here is a brief description of regular expression syntax as used in `sed`.

char
: A single ordinary character matches itself.

*
: Matches a sequence of zero or more instances of matches for the preceding regular expression, which must be an ordinary character, a special character preceded by `\`, a `.`, a grouped regexp (see below), or a bracket expression. As a GNU extension, a postfixed regular expression can also be followed by `*`; for example, `a**` is equivalent to `a*`. POSIX 1003.1-2001 says that `*` stands for itself when it appears at the start of a regular expression or subexpression, but many nonGNU implementations do not support this and portable scripts should instead use `\*` in these contexts.

\+
: As `*`, but matches one or more. It is a GNU extension.

\?
: As `*`, but only matches zero or one. It is a GNU extension.

\{*i*\}
: As `*`, but matches exactly *i* sequences (*i* is a decimal integer; for portability, keep it between 0 and 255 inclusive).

\{*i*,*j*\}
: Matches between *i* and *j*, inclusive, sequences.

\{*i*,\}
: Matches more than or equal to *i* sequences.

\(*regexp*\)
: Groups the inner *regexp* as a whole, this is used to:

  - Apply postfix operators, like `\(abcd\)*`: this will search for zero or more whole sequences of '`abcd`', while `abcd*` would search for '`abc`' followed by zero or more occurrences of '`d`'. Note that support for `\(abcd\)*` is required by POSIX 1003.1-2001, but many non-GNU implementations do not support it and hence it is not universally portable.
  - Use back references (see below).

.
: Matches any character, including newline.

^
: Matches the null string at beginning of the pattern space, i.e. what appears after the circumflex must appear at the beginning of the pattern space.

  In most scripts, pattern space is initialized to the content of each line (see How `sed` works). So, it is a useful simplification to think of `^#include` as matching only lines where '`#include`' is the first thing on line—if there are spaces before, for example, the match fails. This simplification is valid as long as the original content of pattern space is not modified, for example with an `s` command.

  `^` acts as a special character only at the beginning of the regular expression or subexpression (that is, after `\(` or `\|`). Portable scripts should avoid `^` at the beginning of a subexpression, though, as POSIX allows implementations that treat `^` as an ordinary character in that context.

$
: It is the same as `^`, but refers to end of pattern space. `$` also acts as a special character only at the end of the regular expression or subexpression (that is, before `\)` or `\|`), and its use at the end of a subexpression is not portable.

[*list*]
[^*list*]
: Matches any single character in *list*: for example, `[aeiou]` matches all vowels. A list may include sequences like *char1*-*char2*, which matches any character between (inclusive) *char1* and *char2*.

A leading `^` reverses the meaning of *list*, so that it matches any single character *not* in *list*. To include `]` in the list, make it the first character (after the `^` if needed), to include `-` in the list, make it the first or last; to include `^` put it after the first character.

The characters `$`, `*`, `.`, `[`, and `\` are normally not special within *list*. For example, `[\*]` matches either '`\`' or '`*`', because the `\` is not special here. However, strings like `[.ch.]`, `[=a=]`, and `[:space:]` are special within *list* and represent collating symbols, equivalence classes, and character classes, respectively, and `[` is therefore special within *list* when it is followed by `.`, `=`, or `:`. Also, when not in `POSIXLY_CORRECT` mode, special escapes like `\n` and `\t` are recognized within *list*. See Escapes.

regexp1`\|`*regexp2*
: Matches either *regexp1* or *regexp2*. Use parentheses to use complex alternative regular expressions. The matching process tries each alternative in turn, from left to right, and the first one that succeeds is used. It is a GNU extension.

regexp1regexp2
: Matches the concatenation of *regexp1* and *regexp2*. Concatenation binds more tightly than `\|`, `^`, and `$`, but less tightly than the other regular expression operators.

`\`*digit*
: Matches the *digit*-th `\(...\)` parenthesized subexpression in the regular expression. This is called a *back reference*. Subexpressions are implicity numbered by counting occurrences of `\(` left-to-right.

`\n`
: Matches the newline character.

`\`*char*
: Matches *char*, where *char* is one of `$`, `*`, `.`, `[`, `\`, or `^`. Note that the only C-like backslash sequences that you can portably assume to be interpreted are `\n` and `\\`; in particular `\t` is not portable, and matches a '`t`' under most implementations of `sed`, rather than a tab character.

Note that the regular expression matcher is greedy, i.e., matches are attempted from left to right and, if two or more matches are possible starting at the same character, it selects the longest.

Examples:

'`abcdef`'
: Matches '`abcdef`'.

'`a*b`'
: Matches zero or more '`a`'s followed by a single '`b`'. For example, '`b`' or '`aaaaab`'.

'`a\?b`'
: Matches '`b`' or '`ab`'.

'`a\+b\+`'
: Matches one or more '`a`'s followed by one or more '`b`'s: '`ab`' is the shortest possible match, but other examples are '`aaaab`' or '`abbbbb`' or '`aaaaaabbbbbb`'.

'`.*`'
'`.\+`'
: These two both match all the characters in a string; however, the first matches every string (including the empty string), while the second matches only strings containing at least one character.

'`^main.*(.*)`'
: This matches a string starting with '`main`', followed by an opening and closing parenthesis. The '`n`', '`(`' and '`)`' need not be adjacent.

'`^#`'
: This matches a string beginning with '`#`'.

'`\\$`'
: This matches a string ending with a single backslash. The regexp contains two backslashes for escaping.

'`\$`'
: Instead, this matches a string consisting of a single dollar sign, because it is escaped.

'`[a-zA-Z0-9]`'
: In the C locale, this matches any ASCII letters or digits.

'`[^ tab]\+`'
: (Here `tab` stands for a single tab character.) This matches a string of one or more characters, none of which is a space or a tab. Usually this means a word.

'`^\(.*\)\n\1$`'
: This matches a string consisting of two equal substrings separated by a newline.

'`.\{9\}A$`'
: This matches nine characters followed by an '`A`'.

'`^.\{15\}A`'
: This matches the start of a string that contains 16 characters, the last of which is an '`A`'.