Go to the first, previous, next, last section, table of contents.

Regular Expressions

A **regular expression**, or **regexp**, is a way of describing a set of strings. Because regular expressions are such a fundamental part of `awk` programming, their format and use deserve a separate chapter.

A regular expression enclosed in slashes (`/'`) is an `awk` pattern that matches every input record whose text belongs to that set.

The simplest regular expression is a sequence of letters, numbers, or both. Such a regexp matches any string that contains that sequence. Thus, the regexp `foo` matches any string containing `foo`. Therefore, the pattern `/foo/` matches any input record containing the three characters `foo`, *anywhere* in the record. Other kinds of regexps let you specify more complicated classes of strings.

Initially, the examples will be simple. As we explain more about how regular expressions work, we will present more complicated examples.

How to Use Regular Expressions

A regular expression can be used as a pattern by enclosing it in slashes. Then the regular expression is tested against the entire text of each record. (Normally, it only needs to match some part of the text in order to succeed.) For example, this prints the second field of each record that contains the three characters `foo` anywhere in it:

```
$ awk '/foo/ { print $2 }' BBS-list
-| 555-1234
-| 555-6699
-| 555-6480
-| 555-2127
```

Regular expressions can also be used in matching expressions. These expressions allow you to specify the string to match against; it need not be the entire current input record. The two operators, `~` and `!~`, perform regular expression comparisons. Expressions using these operators can be used as patterns or in `if`, `while`, `for`, and `do` statements.

exp `~` /*regexp*/

This is true if the expression *exp* (taken as a string) is matched by *regexp*. The following example matches, or selects, all input records with the upper-case letter `J' somewhere in the first field:

```
$ awk '$1 ~ /J/' inventory-shipped
-| Jan  13  25  15 115
-| Jun  31  42  75 492
-| Jul  24  34  67 436
-| Jan  21  36  64 620
```

So does this:

```
awk '{ if ($1 ~ /J/) print }' inventory-shipped
```

exp `!~ /`*regexp*`/`
This is true if the expression *exp* (taken as a character string) is *not* matched by *regexp*. The following example matches, or selects, all input records whose first field *does not* contain the upper-case letter `J':

```
$ awk '$1 !~ /J/' inventory-shipped
-| Feb  15  32  24 226
-| Mar  15  24  34 228
-| Apr  31  52  63 420
-| May  16  34  29 208
...
```

When a regexp is written enclosed in slashes, like `/foo/`, we call it a **regexp constant**, much like `5.27` is a numeric constant, and `"foo"` is a string constant.

Escape Sequences

Some characters cannot be included literally in string constants (`"foo"`) or regexp constants (`/foo/`). You represent them instead with **escape sequences**, which are character sequences beginning with a backslash (`\').

One use of an escape sequence is to include a double-quote character in a string constant. Since a plain double-quote would end the string, you must use `\"' to represent an actual double-quote character as a part of the string. For example:

```
$ awk 'BEGIN { print "He said \"hi!\" to her." }'
-| He said "hi!" to her.
```

The backslash character itself is another character that cannot be included normally; you write `\\' to put one backslash in the string or regexp. Thus, the string whose contents are the two characters `"' and `\' must be written `"\"\\"'.

Another use of backslash is to represent unprintable characters such as tab or newline. While there is nothing to stop you from entering most unprintable characters directly in a string constant or regexp constant, they may look ugly.

Here is a table of all the escape sequences used in `awk', and what they represent. Unless noted otherwise, all of these escape sequences apply to both string constants and regexp constants.

`\\`
: A literal backslash, `\'.

`\a`
: The "alert" character, `Control-g`, ASCII code 7 (BEL).

`\b`
: Backspace, `Control-h`, ASCII code 8 (BS).

`\f`
: Formfeed, `Control-l`, ASCII code 12 (FF).

`\n`
: Newline, `Control-j`, ASCII code 10 (LF).

`\r`
: Carriage return, `Control-m`, ASCII code 13 (CR).

`\t`
: Horizontal tab, `Control-i`, ASCII code 9 (HT).

`\v`
: Vertical tab, `Control-k`, ASCII code 11 (VT).

`\`*nnn*
: The octal value *nnn*, where *nnn* are one to three digits between `0' and `7'. For example, the code for the ASCII ESC (escape) character is `\033'.

`\x`*hh*`...`
: The hexadecimal value *hh*, where *hh* are hexadecimal digits (`0' through `9' and either `A' through `F' or `a' through `f'). Like the same construct in ANSI C, the escape sequence continues until the first non-hexadecimal digit is seen. However, using more than two hexadecimal digits produces undefined results. (The `\x' escape sequence is not allowed in POSIX `awk`.)

`\/`
: A literal slash (necessary for regexp constants only). You use this when you wish to write a regexp constant that contains a slash. Since the regexp is delimited by slashes, you need to escape the slash that is part of the pattern, in order to tell `awk` to keep processing the rest of the regexp.

`\"`
: A literal double-quote (necessary for string constants only). You use this when you wish to write a string constant that contains a double-quote. Since the string is delimited by double-quotes, you need to escape the quote that is part of the string, in order to tell `awk` to keep processing the rest of the string.

In `gawk`, there are additional two character sequences that begin with backslash that have special meaning in regexps. See section Additional Regexp Operators Only in `gawk`.

In a string constant, what happens if you place a backslash before something that is not one of the characters listed above? POSIX `awk` purposely leaves this case undefined. There are two choices.

- Strip the backslash out. This is what Unix `awk` and `gawk` both do. For example, `"a\qc"` is the same as `"aqc"`.
- Leave the backslash alone. Some other `awk` implementations do this. In such implementations, `"a\qc"` is the same as if you had typed `"a\\qc"`.

In a regexp, a backslash before any character that is not in the above table, and not listed in section Additional Regexp Operators Only in `gawk`, means that the next character should be taken literally, even if it would normally be a regexp operator. E.g., `/a\+b/` matches the three characters `a+b'.

For complete portability, do not use a backslash before any character not listed in the table above.

Another interesting question arises. Suppose you use an octal or hexadecimal escape to represent a regexp metacharacter (see section Regular Expression Operators). Does `awk` treat the character as literal character, or as a regexp operator?

It turns out that historically, such characters were taken literally (d.c.). However, the POSIX standard indicates that they should be treated as real metacharacters, and this is what `gawk` does. However, in compatibility mode (see section Command Line Options), `gawk` treats the characters represented by octal and hexadecimal escape sequences literally when used in regexp constants. Thus, `/a\52b/` is equivalent to `/a\*b/`.

To summarize:

1. The escape sequences in the table above are always processed first, for both string constants and regexp constants. This happens very early, as soon as `awk` reads your program.
2. `gawk` processes both regexp constants and dynamic regexps (see section Using Dynamic Regexps), for the special operators listed in section Additional Regexp Operators Only in `gawk`.
3. A backslash before any other character means to treat that character literally.

Regular Expression Operators

You can combine regular expressions with the following characters, called **regular expression operators**, or **metacharacters**, to increase the power and versatility of regular expressions.

The escape sequences described above in section Escape Sequences, are valid inside a regexp. They are introduced by a `` `\' ``. They are recognized and converted into the corresponding real characters as the very first step in processing regexps.

Here is a table of metacharacters. All characters that are not escape sequences and that are not listed in the table stand for themselves.

`\`
: This is used to suppress the special meaning of a character when matching. For example:

  ```
  \$
  ```

  matches the character `` `$' ``.

`^`
: This matches the beginning of a string. For example:

  ```
  ^@chapter
  ```

  matches the `` `@chapter' `` at the beginning of a string, and can be used to identify chapter beginnings in Texinfo source files. The `` `^' `` is known as an **anchor**, since it anchors the pattern to matching only at the beginning of the string. It is important to realize that `` `^' `` does not match the beginning of a line embedded in a string. In this example the condition is not true:

  ```
  if ("line1\nLINE 2" ~ /^L/) ...
  ```

`$`

This is similar to `^', but it matches only at the end of a string. For example:

```
p$
```

matches a record that ends with a `p'. The `$' is also an anchor, and also does not match the end of a line embedded in a string. In this example the condition is not true:

```
if ("line1\nLINE 2" ~ /1$/) ...
```

`.`

The period, or dot, matches any single character, *including* the newline character. For example:

```
.P
```

matches any single character followed by a `P' in a string. Using concatenation we can make a regular expression like `U.A', which matches any three-character sequence that begins with `U' and ends with `A'. In strict POSIX mode (see section Command Line Options), `.' does not match the NUL character, which is a character with all bits equal to zero. Otherwise, NUL is just another character. Other versions of `awk` may not be able to match the NUL character.

`[...]`

This is called a **character list**. It matches any *one* of the characters that are enclosed in the square brackets. For example:

```
[MVX]
```

matches any one of the characters `M', `V', or `X' in a string. Ranges of characters are indicated by using a hyphen between the beginning and ending characters, and enclosing the whole thing in brackets. For example:

```
[0-9]
```

matches any digit. Multiple ranges are allowed. E.g., the list `[A-Za-z0-9]` is a common way to express the idea of "all alphanumeric characters." To include one of the characters `\', `]', `-' or `^' in a character list, put a `\' in front of it. For example:

```
[d\]]
```

matches either `d', or `]'. This treatment of `\' in character lists is compatible with other `awk` implementations, and is also mandated by POSIX. The regular expressions in `awk` are a superset of the POSIX specification for Extended Regular Expressions (EREs). POSIX EREs are based on the regular expressions accepted by the traditional `egrep` utility. **Character classes** are a new feature introduced in the POSIX standard. A character class is a special notation for describing lists of characters that have a specific attribute, but where the actual characters themselves can vary from country to country and/or from character set to character set. For example, the notion of what is an alphabetic character differs in the USA and in France. A character class is only valid in a regexp *inside* the brackets of a character list. Character classes consist of `[:', a keyword denoting the class, and `:]'. Here are the character classes defined by the POSIX standard.

`[:alnum:]`
: Alphanumeric characters.

`[:alpha:]`

Alphabetic characters.

`[:blank:]`
: Space and tab characters.

`[:cntrl:]`
: Control characters.

`[:digit:]`
: Numeric characters.

`[:graph:]`
: Characters that are printable and are also visible. (A space is printable, but not visible, while an `a' is both.)

`[:lower:]`
: Lower-case alphabetic characters.

`[:print:]`
: Printable characters (characters that are not control characters.)

`[:punct:]`
: Punctuation characters (characters that are not letter, digits, control characters, or space characters).

`[:space:]`
: Space characters (such as space, tab, and formfeed, to name a few).

`[:upper:]`
: Upper-case alphabetic characters.

`[:xdigit:]`
: Characters that are hexadecimal digits.

For example, before the POSIX standard, to match alphanumeric characters, you had to write `/[A-Za-z0-9]/`. If your character set had other alphabetic characters in it, this would not match them. With the POSIX character classes, you can write `/[[:alnum:]]/`, and this will match *all* the alphabetic and numeric characters in your character set. Two additional special sequences can appear in character lists. These apply to non-ASCII character sets, which can have single symbols (called **collating elements**) that are represented with more than one character, as well as several characters that are equivalent for **collating**, or sorting, purposes. (E.g., in French, a plain "e" and a grave-accented "`e" are equivalent.)

Collating Symbols
: A **collating symbol** is a multi-character collating element enclosed in `[.' and `.]'. For example, if `ch' is a collating element, then `[[.ch.]]' is a regexp that matches this collating element, while `[ch]' is a regexp that matches either `c' or `h'.

Equivalence Classes
: An **equivalence class** is a list of equivalent characters enclosed in `[=' and `=]'. Thus, `[[=e`e=]]' is regexp that matches either `e' or ``e'.

These features are very valuable in non-English speaking locales. **Caution:** The library functions that `gawk` uses for regular expression matching currently only recognize POSIX character classes; they do not recognize collating symbols or equivalence classes.

`[^ ...]`
: This is a **complemented character list**. The first character after the `[' *must* be a `^'. It matches any characters *except* those in the square brackets, or newline. For example:

```
[^0-9]
```

matches any character that is not a digit.

`|`

This is the **alternation operator**, and it is used to specify alternatives. For example:

```
^P|[0-9]
```

matches any string that matches either `^P' or `[0-9]'. This means it matches any string that starts with `P' or contains a digit. The alternation applies to the largest possible regexps on either side. In other words, `|' has the lowest precedence of all the regular expression operators.

`(...)`

Parentheses are used for grouping in regular expressions as in arithmetic. They can be used to concatenate regular expressions containing the alternation operator, `|'. For example, `@(samp|code)\{[^}]+\}' matches both `@code{foo}' and `@samp{bar}'. (These are Texinfo formatting control sequences.)

`*`

This symbol means that the preceding regular expression is to be repeated as many times as necessary to find a match. For example:

```
ph*
```

applies the `*' symbol to the preceding `h' and looks for matches of one `p' followed by any number of `h's. This will also match just `p' if no `h's are present. The `*' repeats the *smallest* possible preceding expression. (Use parentheses if you wish to repeat a larger expression.) It finds as many repetitions as possible. For example:

```
awk '/\(c[ad][ad]*r x\)/ { print }' sample
```

prints every record in `sample' containing a string of the form `(car x)', `(cdr x)', `(cadr x)', and so on. Notice the escaping of the parentheses by preceding them with backslashes.

`+`

This symbol is similar to `*', but the preceding expression must be matched at least once. This means that:

```
wh+y
```

would match `why' and `whhy' but not `wy', whereas `wh*y' would match all three of these strings. This is a simpler way of writing the last `*' example:

```
awk '/\(c[ad]+r x\)/ { print }' sample
```

`?`

This symbol is similar to `*', but the preceding expression can be matched either once or not at all. For example:

```
fe?d
```

will match `fed' and `fd', but nothing else.

`{`*n*`}`
`{`*n*`,}`
`{`*n*`,`*m*`}`

One or two numbers inside braces denote an **interval expression**. If there is one number in the braces, the preceding regexp is repeated *n* times. If there are two numbers separated by a comma, the preceding regexp is repeated *n* to *m* times. If there is one number followed by a comma, then the

preceding regexp is repeated at least *n* times.

`wh{3}y`
: matches `whhhy' but not `why' or `whhhhy'.

`wh{3,5}y`
: matches `whhhy' or `whhhhy' or `whhhhhy', only.

`wh{2,}y`
: matches `whhy' or `whhhy', and so on.

Interval expressions were not traditionally available in awk. As part of the POSIX standard they were added, to make awk and egrep consistent with each other. However, since old programs may use `{' and `}' in regexp constants, by default gawk does *not* match interval expressions in regexps. If either `--posix' or `--re-interval' are specified (see section Command Line Options), then interval expressions are allowed in regexps.

In regular expressions, the `*', `+', and `?' operators, as well as the braces `{' and `}', have the highest precedence, followed by concatenation, and finally by `|'. As in arithmetic, parentheses can change how operators are grouped.

If gawk is in compatibility mode (see section Command Line Options), character classes and interval expressions are not available in regular expressions.

The next section discusses the GNU-specific regexp operators, and provides more detail concerning how command line options affect the way gawk interprets the characters in regular expressions.

Additional Regexp Operators Only in gawk

GNU software that deals with regular expressions provides a number of additional regexp operators. These operators are described in this section, and are specific to gawk; they are not available in other awk implementations.

Most of the additional operators are for dealing with word matching. For our purposes, a **word** is a sequence of one or more letters, digits, or underscores (`_').

`\w`
: This operator matches any word-constituent character, i.e. any letter, digit, or underscore. Think of it as a short-hand for `[[:alnum:]_]'.

`\W`
: This operator matches any character that is not word-constituent. Think of it as a short-hand for `[^[:alnum:]_]'.

`\<`
: This operator matches the empty string at the beginning of a word. For example, `/\<away/' matches `away', but not `stowaway'.

`\>`
: This operator matches the empty string at the end of a word. For example, `/stow\>/' matches `stow', but not `stowaway'.

`\y`
: This operator matches the empty string at either the beginning or the end of a word (the word

boundary). For example, `\yballs?\y' matches either `ball' or `balls' as a separate word.

`\B`
: This operator matches the empty string within a word. In other words, `\B' matches the empty string that occurs between two word-constituent characters. For example, /\Brat\B/ matches `crate', but it does not match `dirty rat'. `\B' is essentially the opposite of `\y'.

There are two other operators that work on buffers. In Emacs, a **buffer** is, naturally, an Emacs buffer. For other programs, the regexp library routines that gawk uses consider the entire string to be matched as the buffer.

For awk, since `^' and `$' always work in terms of the beginning and end of strings, these operators don't add any new capabilities. They are provided for compatibility with other GNU software.

`\``
: This operator matches the empty string at the beginning of the buffer.

`\'`
: This operator matches the empty string at the end of the buffer.

In other GNU software, the word boundary operator is `\b'. However, that conflicts with the awk language's definition of `\b' as backspace, so gawk uses a different letter.

An alternative method would have been to require two backslashes in the GNU operators, but this was deemed to be too confusing, and the current method of using `\y' for the GNU `\b' appears to be the lesser of two evils.

The various command line options (see section Command Line Options) control how gawk interprets characters in regexps.

No options
: In the default case, gawk provide all the facilities of POSIX regexps and the GNU regexp operators described above. However, interval expressions are not supported.

`--posix`
: Only POSIX regexps are supported, the GNU operators are not special (e.g., `\w' matches a literal `w'). Interval expressions are allowed.

`--traditional`
: Traditional Unix awk regexps are matched. The GNU operators are not special, interval expressions are not available, and neither are the POSIX character classes ([[:alnum:]] and so on). Characters described by octal and hexadecimal escape sequences are treated literally, even if they represent regexp metacharacters.

`--re-interval`
: Allow interval expressions in regexps, even if `--traditional' has been provided.

Case-sensitivity in Matching

Case is normally significant in regular expressions, both when matching ordinary characters (i.e. not metacharacters), and inside character sets. Thus a `w' in a regular expression matches only a lower-case `w' and not an upper-case `W'.

The simplest way to do a case-independent match is to use a character list: `[Ww]'. However, this can be cumbersome if you need to use it often; and it can make the regular expressions harder to read. There are two alternatives that you might prefer.

One way to do a case-insensitive match at a particular point in the program is to convert the data to a single case, using the `tolower` or `toupper` built-in string functions (which we haven't discussed yet; see section Built-in Functions for String Manipulation). For example:

```
tolower($1) ~ /foo/  { ... }
```

converts the first field to lower-case before matching against it. This will work in any POSIX-compliant implementation of `awk`.

Another method, specific to `gawk`, is to set the variable `IGNORECASE` to a non-zero value (see section Built-in Variables). When `IGNORECASE` is not zero, *all* regexp and string operations ignore case. Changing the value of `IGNORECASE` dynamically controls the case sensitivity of your program as it runs. Case is significant by default because `IGNORECASE` (like most variables) is initialized to zero.

```
x = "aB"
if (x ~ /ab/) ...   # this test will fail

IGNORECASE = 1
if (x ~ /ab/) ...   # now it will succeed
```

In general, you cannot use `IGNORECASE` to make certain rules case-insensitive and other rules case-sensitive, because there is no way to set `IGNORECASE` just for the pattern of a particular rule. To do this, you must use character lists or `tolower`. However, one thing you can do only with `IGNORECASE` is turn case-sensitivity on or off dynamically for all the rules at once.

`IGNORECASE` can be set on the command line, or in a `BEGIN` rule (see section Other Command Line Arguments; also see section Startup and Cleanup Actions). Setting `IGNORECASE` from the command line is a way to make a program case-insensitive without having to edit it.

Prior to version 3.0 of `gawk`, the value of `IGNORECASE` only affected regexp operations. It did not affect string comparison with `==', `!=', and so on. Beginning with version 3.0, both regexp and string comparison operations are affected by `IGNORECASE`.

Beginning with version 3.0 of `gawk`, the equivalences between upper-case and lower-case characters are based on the ISO-8859-1 (ISO Latin-1) character set. This character set is a superset of the traditional 128 ASCII characters, that also provides a number of characters suitable for use with European languages.

The value of `IGNORECASE` has no effect if `gawk` is in compatibility mode (see section Command Line Options). Case is always significant in compatibility mode.

How Much Text Matches?

Consider the following example:

```
echo aaaabcd | awk '{ sub(/a+/, "<A>"); print }'
```

This example uses the `sub` function (which we haven't discussed yet, see section Built-in Functions for String Manipulation) to make a change to the input record. Here, the regexp `/a+/` indicates "one or more `` `a' `` characters," and the replacement text is `` `<A>' ``.

The input contains four `` `a' `` characters. What will the output be? In other words, how many is "one or more"---will `awk` match two, three, or all four `` `a' `` characters?

The answer is, `awk` (and POSIX) regular expressions always match the leftmost, *longest* sequence of input characters that can match. Thus, in this example, all four `` `a' `` characters are replaced with `` `<A>' ``.

```
$ echo aaaabcd | awk '{ sub(/a+/, "<A>"); print }'
-| <A>bcd
```

For simple match/no-match tests, this is not so important. But when doing regexp-based field and record splitting, and text matching and substitutions with the `match`, `sub`, `gsub`, and `gensub` functions, it is very important. Understanding this principle is also important for regexp-based record and field splitting (see section How Input is Split into Records, and also see section Specifying How Fields are Separated).

Using Dynamic Regexps

The right hand side of a `` `~' `` or `` `!~' `` operator need not be a regexp constant (i.e. a string of characters between slashes). It may be any expression. The expression is evaluated, and converted if necessary to a string; the contents of the string are used as the regexp. A regexp that is computed in this way is called a **dynamic regexp**. For example:

```
BEGIN { identifier_regexp = "[A-Za-z_][A-Za-z_0-9]+" }
$0 ~ identifier_regexp    { print }
```

sets `identifier_regexp` to a regexp that describes `awk` variable names, and tests if the input record matches this regexp.

Caution: When using the `` `~' `` and `` `!~' `` operators, there is a difference between a regexp constant enclosed in slashes, and a string constant enclosed in double quotes. If you are going to use a string constant, you have to understand that the string is in essence scanned *twice*; the first time when `awk` reads your program, and the second time when it goes to match the string on the left-hand side of the operator with the pattern on the right. This is true of any string valued expression (such as `identifier_regexp` above), not just string constants.

What difference does it make if the string is scanned twice? The answer has to do with escape sequences, and particularly with backslashes. To get a backslash into a regular expression inside a string, you have to type two backslashes.

For example, `/\*/` is a regexp constant for a literal `` `*' ``. Only one backslash is needed. To do the same thing with a string, you would have to type `"\\*"`. The first backslash escapes the second one, so that the string actually contains the two characters `` `\' `` and `` `*' ``.

Given that you can use both regexp and string constants to describe regular expressions, which should you use? The answer is "regexp constants," for several reasons.

1. String constants are more complicated to write, and more difficult to read. Using regexp constants makes your programs less error-prone. Not understanding the difference between the two kinds of constants is a common source of errors.
2. It is also more efficient to use regexp constants: `awk` can note that you have supplied a regexp and store it internally in a form that makes pattern matching more efficient. When using a string constant, `awk` must first convert the string into this internal form, and then perform the pattern matching.
3. Using regexp constants is better style; it shows clearly that you intend a regexp match.

Go to the first, previous, next, last section, table of contents.