In this chapter:

- *How to Use Regular Expressions*
- *Escape Sequences*
- *Regular Expression Operators*
- *Using Character Lists*
- *gawk-Specific Regexp Operators*
- *Case Sensitivity in Matching*
- *How Much Text Matches?*
- *Using Dynamic Regexps*

2 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 in this chapter are simple. As we explain more about how regular expressions work, we will present more complicated instances.

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, the following prints the second field of each record that contains the string `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. (See the section "Control Statements in Actions" in Chapter 6, *Patterns, Actions, and Variables*.) For example:

```
exp ~ /regexp/
```

is true if the expression *exp* (taken as a string) matches *regexp*. The following example matches, or selects, all input records with the uppercase 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
```

This next example is true if the expression *exp* (taken as a character string) does *not* match *regexp*:

```
exp !~ /regexp/
```

The following example matches, or selects, all input records whose first field *does not* contain the uppercase 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 enclosed in slashes, such as `/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/`). Instead, they should be represented 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. Because a plain double quote ends 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 must write `\\` to put one backslash in the string or regexp. Thus, the string whose contents are the two characters `"` and `\` must be written `"\"\\"`.

Backslash also represents 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.

The following list describes all the escape sequences used in *awk* and what they represent. Unless noted otherwise, all these escape sequences apply to both string constants and regexp constants:

`\\` A literal backslash, `\`.

`\a` The "alert" character, Ctrl-g, ASCII code 7 (BEL). (This usually makes some sort of audible noise.)

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

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

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

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

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

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

`\`*nnn*

The octal value *nnn*, where *nnn* stands for 1 to 3 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* stands for a sequence of hexadecimal digits (`0`–`9`, and either `A`–`F` or `a`–`f`). Like the same construct in ISO C, the escape sequence continues until the first nonhexadecimal 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). This expression is used when you want to write a regexp constant that contains a slash. Because 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). This expression is used when you want to write a string constant that contains a double quote. Because 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*, a number of additional two-character sequences that begin with a backslash have special meaning in regexps. See the section "gawk-Specific Regexp Operators" later in this chapter.

In a regexp, a backslash before any character that is not in the previous list and not listed in the section "gawk-Specific Regexp Operators" later in this chapter means that the next character should be taken literally, even if it would normally be a regexp operator. For example, `/a\+b/` matches the three characters `a+b`.

For complete portability, do not use a backslash before any character not shown in the previous list.

Backslash Before Regular Characters

If you place a backslash in a string constant before something that is not one of the characters previously listed, POSIX *awk* purposely leaves what happens as 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"`. (Because this is such an easy bug both to introduce and to miss, *gawk* warns you about it.) Consider `FS = "[ \t]+\|[ \t]+"` to use vertical bars surrounded by whitespace as the field separator. There should be two backslashes in the string `FS = "[ \t]+\\|[ \t]+"`.

Leave the backslash alone
: Some other *awk* implementations do this. In such implementations, typing `"a\qc"` is the same as typing `"a\\qc"`.

Escape Sequences for Metacharacters

Suppose you use an octal or hexadecimal escape to represent a regexp metacharacter. (See the section "Regular Expression Operators" later in this chapter.) Does *awk* treat the character as a literal character or as a regexp operator?

Historically, such characters were taken literally. (d.c.) However, the POSIX standard indicates that they should be treated as real metacharacters, which is what *gawk* does. In compatibility mode (see the section "Command-Line Options" in Chapter 11, *Running awk and gawk*), *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/`.

Regular Expression Operators

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

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

Here is a list of metacharacters. All characters that are not escape sequences and that are not listed here 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 `@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*, because it anchors the pattern to match 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. The condition is not true in the following example:

```
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 an anchor and does not match the end of a line embedded in a string. The condition is not true as follows:

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

`.` (A period, or "dot.") This 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 such as `U.A`, which matches any three-character sequence that begins with `U` and ends with `A`.

In strict POSIX mode (see the section "Command-Line Options" in Chapter 11), the dot 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. A full discussion of what can be inside the square brackets of a character list is given in the section "Using Character Lists" later in this chapter.

`[^ ...]`
This is a *complemented character list*. The first character after the `[` *must* be a `^`. It matches any characters *except* those in the square brackets. For example, `[^awk]` matches any character that is not an `a`, `w`, or `k`.

`|` This is the *alternation operator* and it is used to specify alternatives. The `|` has the lowest precedence of all the regular expression operators. For example, `^P|[[:digit:]]` matches any string that matches either `^P` or `[[:digit:]]`. 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.

`(...)`
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}`.

`*` This symbol means that the preceding regular expression should 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 also matches just `p` if no `h`s are present.

The `*` repeats the *smallest* possible preceding expression. (Use parentheses if you want to repeat a larger expression.) It finds as many repetitions as possible. For example, `awk '/\(c[ad][ad]*r x\)/ { print }' sample` prints every

* In other literature, you may see a character list referred to as either a *character set*, a *character class*, or a *bracket expression*.

9 October 2001 01:40

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 `*`, except that 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. The following is a simpler way of writing the last `*` example:

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

`?`
This symbol is similar to `*`, except that the preceding expression can be matched either once or not at all. For example, `fe?d` matches `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`, `whhhhy`, or `whhhhhy`, only.

`wh{2,}y`
Matches `whhy` or `whhhy`, and so on.

Interval expressions were not traditionally available in *awk*. They were added as part of the POSIX standard to make *awk* and *egrep* consistent with each other.

However, because 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 the section "Command-Line Options" in Chapter 11), then interval expressions are allowed in regexps.

For new programs that use `{` and `}` in regexp constants, it is good practice to always escape them with a backslash. Then the regexp constants are valid and work the way you want them to, using any version of *awk*.*

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.

* Use two backslashes if you're using a string constant with a regexp operator or function.

In POSIX *awk* and *gawk*, the `*`, `+`, and `?` operators stand for themselves when there is nothing in the regexp that precedes them. For example, `/+/` matches a literal plus sign. However, many other versions of *awk* treat such a usage as a syntax error.

If *gawk* is in compatibility mode (see the section "Command-Line Options" in Chapter 11), POSIX character classes and interval expressions are not available in regular expressions.

Using Character Lists

Within a character list, a *range expression* consists of two characters separated by a hyphen. It matches any single character that sorts between the two characters, using the locale's collating sequence and character set. For example, in the default C locale, `[a-dx-z]` is equivalent to `[abcdxyz]`. Many locales sort characters in dictionary order, and in these locales, `[a-dx-z]` is typically not equivalent to `[abcdxyz]`; instead it might be equivalent to `[aBbCcDdxXyYz]`, for example. To obtain the traditional interpretation of bracket expressions, you can use the C locale by setting the LC_ALL environment variable to the value `C`.

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 the actual characters 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 between the United States and 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 `:]`. Table 2-1 lists the character classes defined by the POSIX standard.

Table 2-1. POSIX Character Classes

Class	Meaning
`[:alnum:]`	Alphanumeric characters.
`[:alpha:]`	Alphabetic characters.
`[:blank:]`	Space and tab characters.
`[:cntrl:]`	Control characters.
`[:digit:]`	Numeric characters.
`[:graph:]`	Characters that are both printable and visible. (A space is printable but not visible, whereas an `a` is both.)
`[:lower:]`	Lowercase alphabetic characters.
`[:print:]`	Printable characters (characters that are not control characters).
`[:punct:]`	Punctuation characters (characters that are not letters, digits, control characters, or space characters).
`[:space:]`	Space characters (such as space, tab, and formfeed, to name a few).
`[:upper:]`	Uppercase alphabetic characters.
`[:xdigit:]`	Characters that are hexadecimal digits.

For example, before the POSIX standard, you had to write `/[A-Za-z0-9]/` to match alphanumeric characters. If your character set had other alphabetic characters in it, this would not match them, and if your character set collated differently from ASCII, this might not even match the ASCII alphanumeric characters. With the POSIX character classes, you can write `/[[:alnum:]]/` to match 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. They can also have several characters equivalent for *collating*, or sorting, purposes. (For example, in French, a plain "e" and a grave-accented "è" are equivalent.) These sequences are:

Collating symbols
: Multicharacter collating elements enclosed between `[.` and `.]`. For example, if `ch` is a collating element, then `[[.ch.]]` is a regexp that matches this collating element, whereas `[ch]` is a regexp that matches either `c` or `h`.

Equivalence classes
: Locale-specific names for lists of characters that are equal. The name is enclosed between `[=` and `=]`. For example, the name `e` might be used to represent all of "e," "è," and "é." In this case, `[[=e=]]` is a regexp that matches any of `e`, `é`, or `è`.

These features are very valuable in non-English-speaking locales.

The library functions that *gawk* uses for regular expression matching currently recognize only POSIX character classes; they do not recognize collating symbols or equivalence classes.

gawk-Specific Regexp Operators

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 deal with word matching. For our purposes, a *word* is a sequence of one or more letters, digits, or underscores (_):

`\w`	Matches any word-constituent character—that is, it matches any letter, digit, or underscore. Think of it as short-hand for `[[:alnum:]_]`.
`\W`	Matches any character that is not word-constituent. Think of it as shorthand for `[^[:alnum:]_]`.
`\<`	Matches the empty string at the beginning of a word. For example, `/\<away/` matches `away` but not `stowaway`.
`\>`	Matches the empty string at the end of a word. For example, `/stow\>/` matches `stow` but not `stowaway`.
`\y`	Matches the empty string at either the beginning or the end of a word (i.e., the word boundary). For example, `\yballs?\y` matches either `ball` or `balls`, as a separate word.
`\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, *gawk*'s regexp library routines consider the entire string to match as the buffer. The operators are:

`` \` ``	Matches the empty string at the beginning of a buffer (string).
`\'`	Matches the empty string at the end of a buffer (string).

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

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 too confusing. The current method of using `\y` for the GNU `\b` appears to be the lesser of two evils.

The various command-line options (see the section "Command-Line Options" in Chapter 11) control how *gawk* interprets characters in regexps:

No options
: In the default case, *gawk* provides all the facilities of POSIX regexps and the previously described GNU regexp operators. 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, nor are the POSIX character classes (`[[:alnum:]]`, etc.). 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 lowercase `w` and not an uppercase `W`.

The simplest way to do a case-independent match is to use a character list—for example, `[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 perform 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 the section "String-Manipulation Functions" in Chapter 8, *Functions*). For example:

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

converts the first field to lowercase before matching against it. This works in any POSIX-compliant *awk*.

Another method, specific to *gawk*, is to set the variable `IGNORECASE` to a nonzero value (see the section "Built-in Variables" in Chapter 6). When `IGNORECASE` is not zero, *all* regexp and string operations ignore case. Changing the value of `IGNORECASE` dynamically controls the case-sensitivity of the 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 straightforward way to set `IGNORECASE` just for the pattern of a particular rule.* To do this, use either character lists or `tolower`. However, one thing you can do with `IGNORECASE` only is dynamically turn case-sensitivity on or off for all the rules at once.

`IGNORECASE` can be set on the command line or in a `BEGIN` rule (see the section "Other Command-Line Arguments" in Chapter 11; also see the section "Startup and cleanup actions" in Chapter 6). Setting `IGNORECASE` from the command line is a way to make a program case-insensitive without having to edit it.

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

Beginning with *gawk* 3.0, the equivalences between upper- and lowercase 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, which 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 the section "Command-Line Options" in Chapter 11). Case is always significant in compatibility mode.

* Experienced C and C++ programmers will note that it is possible, using something like `IGNORECASE = 1 && /foObAr/ { ... }` and `IGNORECASE = 0 || /foobar/ { ... }`. However, this is somewhat obscure and we don't recommend it.

How Much Text Matches?

Consider the following:

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

This example uses the `sub` function (which we haven't discussed yet; see the section "String-Manipulation Functions" in Chapter 8) 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. *awk* (and POSIX) regular expressions always match the leftmost, *longest* sequence of input characters that can match. Thus, all four `a` characters are replaced with `<A>` in this example:

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

For simple match/no-match tests, this is not so important. But when doing 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 the section "How Input Is Split into Records" and the section "Specifying How Fields Are Separated" in Chapter 3, *Reading Input Files*).

Using Dynamic Regexps

The righthand 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 to a string if necessary; the contents of the string are used as the regexp. A regexp that is computed in this way is called a *dynamic regexp*:

```
BEGIN { digits_regexp = "[[:digit:]]+" }
$0 ~ digits_regexp    { print }
```

This sets `digits_regexp` to a regexp that describes one or more digits, and tests whether the input record matches this regexp.

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 lefthand side of the operator with the pattern on the right. This is true of any string-valued expression (such as `digits_regexp`, shown previously), 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 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:

- 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.
- It is 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.
- Using regexp constants is better form; it shows clearly that you intend a regexp match.

Using \n in Character Lists of Dynamic Regexps

Some commercial versions of *awk* do not allow the newline character to be used inside a character list for a dynamic regexp:

```
$ awk '$0 ~ "[ \t\n]"'
awk: newline in character class [
]...
 source line number 1
 context is
        >>>  <<<
```

But a newline in a regexp constant works with no problem:

```
$ awk '$0 ~ /[ \t\n]/'
here is a sample line
here is a sample line
Control-d
```

gawk does not have this problem, and it isn't likely to occur often in practice, but it's worth noting for future reference.