sed (**S**tream **ED**itor) refers to a Unix utility for parsing text files and the programming language it uses to apply textual transformations to a sequential stream of data. It reads input files line by line, applying the operation which has been specified via the command line (or a *sed script*), and then outputs the line. It was developed from 1973 to 1974 as a Unix utility by Lee E. McMahon of Bell Labs, and is available today on most operating systems

Usage

The following example shows a typical use of sed, where the *-e* option indicates that the sed expression follows:

```
sed -e 's/oldstuff/newstuff/g' inputFileName > outputFileName
```

In many versions, the -e is not required and the expression is automatically read. The *s* stands for substitute; the *g* stands for global, which means that all matching occurrences in the line would be replaced. After the first slash is the regular expression to search for and after the second slash is the expression to replace it with. The substitute command (s///) is by far the most powerful and most commonly used sed command.

Under Unix, sed is often used as a filter in a pipeline:

```
generate_data | sed -e 's/x/y/g'
```

That is, generate the data, and then make the small change of replacing *x* with *y*.

Several substitutions or other commands can be put together in a file called, for example, *subst.sed* and then be applied using the *-f* option to read the commands from the file:

```
sed -f subst.sed inputFileName > outputFileName
```

Besides substitution, other forms of simple processing are possible. For example, the following deletes empty lines or lines that only contain spaces:

```
sed -e '/^ *$/d' inputFileName
```

This example used some of the following regular expression metacharacters:

- The caret (`^`) matches the beginning of the line.
- The dollar sign (`$`) matches the end of the line.
- A period (`.`) matches any single character.
- The asterisk (`*`) matches zero or more occurrences of the previous character.

- A bracketed expression delimited by `[` and `]` matches any of the characters inside the brackets.

Complex sed constructs are possible, to the extent that it can be conceived of as a highly specialised, albeit simple, programming language. Flow of control, for example, can be managed by use of a label (a colon followed by a string which is to be the label name) and the branch instruction **b**; an instruction **b** followed by a valid label name will move processing to the block following the label; if the label does not exist then the branch will end the script.

[edit]History

sed is one of the very early Unix commands that permitted command line processing of data files. It evolved as the natural successor to the popular grep command. Cousin to the laterAWK, sed allowed powerful and interesting data processing to be done by shell scripts. sed was probably the earliest Unix tool that really encouraged regular expressions to be used ubiquitously.

sed and AWK are often cited as the progenitors and inspiration for Perl; in particular the s/// syntax from the example above is part of Perl's syntax.

sed's language does not have variables and has only primitive GOTO and branching functionality; nevertheless, the language is Turing-complete. [1]

GNU sed includes several new features such as in-place editing of files (i.e., replace the original file with the result of applying the sed program). In-place editing is often used instead of edscripts: for example,

```
sed -i 's/abc/def/' file
```

can be used instead of

```
ed file
1,$ s/abc/def/
w
q
```

There is an extended version of sed called **Super-sed** (ssed) that includes regular expressions compatible with Perl.

Another version of sed is **minised**, originally reverse-engineered from the 4.1BSD sed by Eric S. Raymond and currently maintained by René Rebe. **minised** was used by the GNU projectuntil the GNU project wrote a new version of sed based on the new GNU regular expression library. The current **minised** contains some extensions to BSD sed but is not as feature-rich as GNU sed. Its advantage is that it is very fast and uses little memory. It is used on embedded systems and is the version of sed provided with Minix.

[edit]Samples

This example will enable sed, which usually only works on one line, to remove newlines from sentences where the second sentence starts with one space.

Consider the following text:

```
This is my cat
 my cat's name is betty
This is my dog
 my dog's name is frank
```

The sed script below will turn it into:

```
This is my cat my cat's name is betty
This is my dog my dog's name is frank
```

Here's the script:

```
sed 'N;s/\n / /;P;D;'
```

- (N) add the next line to the work buffer
- (s) substitute
- (/\n /) match: \n and one space
- (/ /) replace with: one space
- (P) print the top line of the work buffer
- (D) delete the top line from the work buffer and run the script again

[edit]The Address Command (submatches)

More complex substitutions are possible using the "Address" command:

```
/pattern1/s/pattern2/replacement/flags
```

Here, 'pattern1' is the address. The command following the address will only execute if the current line being processed matches the pattern (address) specified. This will replace pattern2 with replacement where pattern1 is matched. Likewise:

```
/pattern1/!s/pattern2/replacement/flags
```

will replace pattern2 where pattern1 is *not* matched.

For example, if you have a file (text.txt) containing the following lines:

```
Hello world.
```

```
Hello world. I love you.
```

And you want to replace "world" with "mom", but only on those lines that contain the word "you", you can use:

```
sed -e '/you/s/world/mom/g' text.txt
```

will result in:

```
Hello world.
Hello mom. I love you.
```

You can negate this behavior with:

```
sed -e '/you/!s/world/mom/g' text.txt
```

which will result in the opposite:

```
Hello mom.
Hello world. I love you.
```

[edit]Exotic examples

Despite the limited possibilities of sed, there are sed scripts for games as sokoban or arkanoid. Even some debuggers have been developed in sed. [2]

[edit]Social usage

In online discussion forums, particularly internet relay chat, involving people familiar with UNIX command line interfaces, corrections to typing errors are sometimes expressed using a syntax shared between several UNIX utilities, including sed, ex, awk and perl.[1]

[edit]Further reading

- Dale Dougherty & Arnold Robbins (March 1997). *sed & awk*, 2nd Edition, O'Reilly. ISBN 1-56592-225-5.
- Arnold Robbins (June 2002). *sed and awk Pocket Reference*, 2nd Edition, O'Reilly and Associates. ISBN 0-596-00352-8.
- Peter Patsis (1998-12-30). *UNIX AWK and SED Programmer's Interactive Workbook (UNIX Interactive Workbook)*. Prentice Hall. ISBN 0-13-082675-8.
- The sed FAQ
- GNU sed manual

AWK is a general purpose programming language that is designed for processing text-based data, either in files or data streams. The name AWK is derived from the family names of its authors — Alfred **A**ho, Peter **W**einberger, and Brian **K**ernighan; however, it is not commonly pronounced as a string of separate letters but rather to sound the same as the name of the bird, auk (which acts as an emblem of the language such as on The AWK Programming Language book cover). `awk`, when written in all lowercase letters, refers to the Unix or Plan 9 program that runs other programs written in the AWK programming language.

AWK is an example of a programming language that extensively uses the string datatype, associative arrays (that is, arrays indexed by key strings), and regular expressions. The power, terseness, and limitations of AWK programs and sed scripts inspired Larry Wall to write Perl. Because of their dense notation, all these languages are often used for writing one-liner programs.

AWK is one of the early tools to appear in Version 7 Unix and gained popularity as a way to add computational features to a Unix pipeline. A version of the AWK language is a standard feature of nearly every modern Unix-like operating system available today. AWK is mentioned in theSingle UNIX Specification as one of the mandatory utilities of a Unix operating system. Besides the Bourne shell, AWK is the only other scripting language available in a standard Unix environment. Implementations of AWK exist as installed software for almost all other operating systems.

Structure of AWK programs

An AWK program is a series of

```
pattern { action }
```

pairs, where *pattern* is typically an expression and *action* is a series of commands. Each line of input is tested against all the patterns in turn and the *action* executed if the expression is true. Either the *pattern* or the *action* may be omitted. The *pattern* defaults to matching every line of input. The default *action* is to print the line of input.

In addition to a simple AWK expression, the *pattern* can be *BEGIN* or *END* causing the *action* to be executed before or after all lines of input have been read, or *pattern1, pattern2* which matches the

range of lines of input starting with a line that matches *pattern1* up to and including the line that matches *pattern2* before again trying to match against *pattern1* on future lines.

In addition to normal arithmetic and logical operators, AWK expressions include the tilde operator, ~, which matches a regular expression against a string. As a handy default, */regexp/* without using the tilde operator matches against the current line of input.

[edit] AWK commands

AWK commands are the statement that is substituted for *action* in the examples above. AWK commands can include function calls, variable assignments, calculations, or any combination thereof. AWK contains built-in support for many functions; many more are provided by the various flavors of AWK. Also, some flavors support the inclusion of dynamically linked libraries, which can also provide more functions.

For brevity, the enclosing curly braces (*{ }*) will be omitted from these examples.

[edit] The *print* command

The *print* command is used to output text. The output text is always terminated with a predefined string called the output record separator (ORS) whose default value is a newline. The simplest form of this command is:

```
print
```

This displays the contents of the current line. In AWK, lines are broken down into *fields*, and these can be displayed separately:

`print $1`

Displays the first field of the current line

`print $1, $3`

Displays the first and third fields of the current line, separated by a predefined string called the output field separator (OFS) whose default value is a single space character

Although these fields (*$X*) may bear resemblance to variables (the $ symbol indicates variables in perl), they actually refer to the fields of the current line. A special case, *$0*, refers to the entire line. In fact, the commands "`print`" and "`print $0`" are identical in functionality.

The *print* command can also display the results of calculations and/or function calls:

```
print 3+2
print foobar(3)
print foobar(variable)
print sin(3-2)
```

Output may be sent to a file:

```
print "expression" > "file name"
```

[edit]Variables and Syntax

Variable names can use any of the characters [A-Za-z0-9_], with the exception of language keywords. The operators + - * / represent addition, subtraction, multiplication, and division, respectively. For string concatenation, simply place two variables (or string constants) next to each other. It is optional to use a space in between if string constants are involved. But you can't place two variable names adjacent to each other without having a space in between. String constants are delimited by double quotes. Statements need not end with semicolons. Finally, comments can be added to programs by using # as the first character on a line.

[edit]User-defined functions

In a format similar to C, function definitions consist of the keyword `function`, the function name, argument names and the function body. Here is an example of a function.

```
function add_three (number, temp) {
  temp = number + 3
  return temp
}
```

This statement can be invoked as follows:

```
print add_three(36)      # Outputs 39
```

Functions can have variables that are in the local scope. The names of these are added to the end of the argument list, though values for these should be omitted when calling the function. It is convention to add some whitespace in the argument list before the local variables, in order to indicate where the parameters end and the local variables begin.

[edit]Sample applications

[edit]Hello World

Here is the ubiquitous "Hello world program" program written in AWK:

```
BEGIN { print "Hello, world!" }
```

[edit]Print lines longer than 80 characters

Print all lines longer than 80 characters. Note that the default action is to print the current line.

```
length > 80
```

The AWK Programming Language now specifies an explicit $0 in the length function:

```
length($0) > 80
```

[edit]Print a count of words

Count words in the input, and print lines, words, and characters (like wc)

```
{
    w += NF
    c += length + 1
}
END { print NR, w, c }
```

[edit]Sum last word

```
{ s += $NF }
END { print s + 0 }
```

As there is no pattern for the first line of the program, every line of input matches by default so the *s += $NF* action is executed. *s* is incremented by the numeric value of *$NF* which is the last word on the line as defined by AWK's field separator, by default white-space. *NF* is the number of fields in the current line, e.g. 4. Since *$4* is the value of the fourth field, *$NF* is the value of the last field in the line regardless of how many fields this line has, or whether it has more or fewer fields than surrounding lines. $ is actually a unary operator with the highestoperator precedence.

(If the line has no fields then *NF* is 0, *$0* is the whole line, which in this case is empty apart from possible white-space, and so has the numeric value 0.)

At the end of the input the *END* pattern matches so *s* is printed. However, since there may have been no lines of input at all, in which case no value has ever been assigned to *s*, it will by default be an empty string. Adding zero to a variable is an AWK idiom for coercing it from a string to a numeric value. (Concatenating an empty string is to coerce from a number to a string, e.g. *s ""*. Note, there's no operator to concatenate strings, they're just placed adjacently.) With the coercion the program prints *0* on an empty input, without it an empty line is printed.

[edit]Match a range of input lines

```
$ yes Wikipedia | awk '{ printf "%6d  %s\n", NR, $0 }' | awk 'NR %
4 == 1, NR % 4 == 3' | head -7
     1  Wikipedia
```

```
     2  Wikipedia
     3  Wikipedia
     5  Wikipedia
     6  Wikipedia
     7  Wikipedia
     9  Wikipedia
$
```

The yes command repeatedly prints the letter "y" on a line. In this case, we tell the command to print the word "Wikipedia". The first awk command prints each line numbered. The printf function acts like the one in most other programming languages. The second awk command works as follows: *NR* is the number of records, typically lines of input, AWK has so far read, i.e. the current line number, starting at 1 for the first line of input. *%* is the modulo operator. *NR % 4 == 1* is true for the first, fifth, ninth, etc., lines of input. Likewise, *NR % 4 == 3* is true for the third, seventh, eleventh, etc., lines of input. The range pattern is false until the first part matches, on line 1, and then remains true up to and including when the second part matches, on line 3. It then stays false until the first part matches again on line 5. The head command is then issued to only print the first seven lines of that output, thus ensuring that the program actually quits (as yes never terminates itself).

The first part of a range pattern being constantly true, e.g. *1*, can be used to start the range at the beginning of input. Similarly, if the second part is constantly false, e.g. *0*, the range continues until the end of input:

```
/^--cut here--$/, 0
```

prints lines of input from the first line matching the regular expression ^*--cut here--$* to the end.

[edit]Calculate word frequencies

Word frequency, uses associative arrays:

```
BEGIN { FS="[^a-zA-Z]+"}

{ for (i=1; i<=NF; i++)
     words[tolower($i)]++
}

END { for (i in words)
    print i, words[i]
}
```

The BEGIN block sets the field separator to any sequence of non-alphabetic characters. Note that separators can be regular expressions. After that, we get to a bare action, which performs the action on every input line. In this case, for every field on the line, we add one to the number of times that word, first converted to lowercase, appears. Finally, in the END block, we print the words with their frequencies. The line

```
for (i in words)
```

creates a loop that goes through the array words, setting i to each *subscript* of the array. This is different from most languages, where such a loop goes through each value in the array. This means that you print the word with each count in a simple way.

[edit]Self-contained AWK scripts

As with many other programming languages, self-contained AWK script can be constructed using the so-called "shebang" syntax.

For example, a UNIX command called `hello.awk` that prints the string "Hello, world!" may be built by creating a file named `hello.awk` containing the following lines:

```
#!/usr/bin/awk -f
BEGIN { print "Hello, world!"; exit }
```