7. Awk

7.1 Introduction

The basic function of awk is to search input files for lines that contain certain patterns. When a line matches one of the patterns, awk performs specified actions on that line. awk continues to process input lines in this way until it reaches the end of the input files.

An awk program consists of a series of rules. Each rule specifies one pattern to search for and one action to perform upon finding the pattern.

Syntactically, an awk programs looks like

```
pattern { action }
pattern { action }
...
```

The action is enclosed in braces to separate it from the pattern. Rules are usually separated by new lines.

7.2 Fields and Records

Awk views each line of the input file as a **record**. Each record is a collection of **fields** which are separated by a **field separator** such as space or comma. Default field separator is space or tab. Default record separator is newline.

For example, the following is an input file containing 7 records where each record has 5 fields.

```
A101  Mathematics 30  30  40
A102  Physics     40  40  40
A103  Physics     50  50  50
A104  Physics     35  35  35
A105  Mathematics 25  25  25
A106  Mathematics 32  48  42
A107  Chemistry   22  32  42
```

Listing 7.1: Sample Data 1 (marks)

7.2.1 Buffers

Awk provides two types of buffers

- **Field Buffers**
  There are field buffers for each field in the current record of the input file. Each field buffer has a name starting with dollar sign (`$`) followed by a field number. Field numbers begin with 1. For example, `$1` represents the first field, `$2` represents second field and so on.
- **Record Buffer**
  There is only one record buffer, `$0`, which represents the whole record.

7.2.2 Variables

There are two types of variables in awk.

System Variables

More than twelve system variables are used by awk. Some of the system variables are listed in Table 7.1.

Variable	Function	Default
`FS`	Input field separator	space or tab
`RS`	Input record separator	newline
`OFS`	Output field separator	space or tab
`ORS`	Output record separator	newline
`NF`	Number of non-empty fields in the current record	
`NR`	Number of records read from the input file	

Table 7.1: System Variables

User-Defined Variables

An awk program can contain any number of user-defined variables. These variable can contain numbers, strings or arrays. Variable names must start with a letter and can be followed by any sequence of letters, digits or underscores. They do not need to be declared. e.g. `total`.

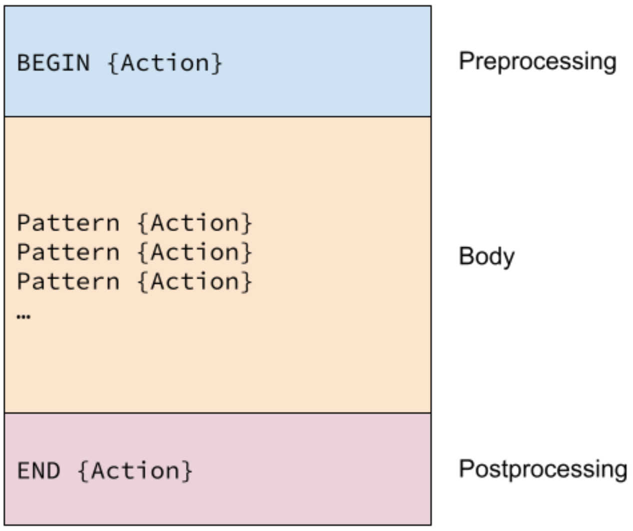

Figure 7.1: Structure of an awk Script

7.3 Awk Scripts

Awk scripts can be divided into three parts: begin, body and end (Figure 7.1).

1. BEGIN
   Action specified in the curly brackets after the BEGIN keyword will be executed only once before awk starts processing the input file. This part is optional.
2. Body
   The body operates in a loop, processing each record in the input file. It checks for each pattern, and when a match is found, it executes the action that follows that pattern. This process is repeated for each record in the file.
3. END
   Action specified after the keyword END will be executed once, after all the records in the input file are read. This part is optional.

7.4 Running awk Programs

An awk program can be run (invoked) in different ways.

7.4.1 Running from the Command Line

If the program is short, it can be easily included in the command that runs awk, as given below.

```
awk 'program' input-file1 input-file2 ...
```

While running this way, the script is enclosed in quotes. A simple awk program which takes the marks file (Listing 7.1) as input is given below.

```
$ awk '$2=="Mathematics" {print $1}' marks
A101
A105
A106
```

This script checks whether the second field ($2) matches "Mathematics". If it does, it prints the registration number of that student.

7.4.2 Running the Script in a File

If the program is long, it is more convenient to put it in a file and run it with a command, as given below

```
awk -f program-file input-file1 input-file2 ...
```

For example consider the following script.

```
BEGIN{
  print("Mathematics")
  total=0
}
$2=="Mathematics"{
  total = total + $3+$4+$5
  print($1,$3+$4+$5)
}
END{
  print("Grand Total: ",total)
}
```

Listing 7.2: marks.awk

The script can be executed as shown below.

```
$ awk -f marks.awk marks
Mathematics
A101 100
A105 75
A106 122
Grand Total: 297
```

7.5 Operators in Awk

7.5.1 Arithmetic Operators

The list of awk arithmetic operators in order from the highest precedence to the lowest are given below.

- `x ^ y`
  Exponentiation; `x` raised to `y`. `3 ^ 2` has the value `9`.
- `x ** y`
  Exponentiation; `x` raised to `y`. `3 ** 2` has the value `9`.
- `-x`
  Negation.
- `+x`
  Unary plus
- `x * y`
  Multiplication. `3 * 2` has the value `6`.
- `x / y`
  Division; because all numbers in awk are floating-point numbers, the result of division will be a floating point number. `3 / 2` has the value `1.5`.
- `x % y`
  Modulus operator. `3 % 2` has the value `1`.
- `x + y`
  Addition. `3 + 2` has the value `5`.
- `x - y`
  Subtraction. `3 - 2` has the value `1`.

An example is shown in Listing 7.2.

7.5.2 Relational Operators

Relational expressions compare two values. When the two values are numeric, an algebraic comparison is used and when they are string, string comparison is used. The relational operators in awk are given in Table 7.2.

`x < y`	True if `x` is less than `y`
`x <= y`	True if `x` is less than or equal to `y`
`x > y`	True if `x` is greater than `y`
`x >= y`	True if `x` is greater than or equal to `y`
`x == y`	True if `x` is equal to `y`
`x != y`	True if `x` is not equal to `y`

Table 7.2: Relational Operators

7.5.3 Conditional Operator

General form of conditional expression is

```
exp1 ? exp2 : exp3
```

There are three subexpressions. The first, `exp1`, is always computed first. If it is `true` (not zero or not null), then `exp2` is computed next, and its value becomes the value of the whole expression. Otherwise,`exp3` is computed next, and its value becomes the value of the whole expression.

7.5.4 Logical Operators

Logical expressions are used to combine two or more expressions. The logical operators are given in Table 7.3

`!expr`	True if `expr` is false. False if `expr` is true.
`expr1 && expr2`	True if both `expr1` and `expr2` is true. False otherwise.
`expr1 || expr2`	True if one of `expr1` or `expr1` is true. False otherwise.

Table 7.3: Logical Operators

7.5.5 Assignment Operators

An assignment is an expression that stores a value into a variable. The other assignment operators are given in Table 7.4.

`a+=b`	`a = a - b`
`a-=n`	`a = a - b`
`a*=b`	`a = a * b`
`a/=b`	`a = a / b`
`a%=b`	`a = a % b`
`a^=b`	`a = a ^ b`
`a**=b`	`a = a ** b`

Table 7.4: Assignment Operators

7.5.6 Increment and Decrement Operators

Increment (`++`) and decrement (`--`) operators increase or decrease the value of a variable by one. Increment operator can be used to increment a variable either before or after taking its value. To pre-increment a variable `v`, write `++v`. This adds one to the value of `v` - that new value is also the value of the expression. Writing the `++` after the variable specifies post-increment. This increments the variable value just the same; the difference is that the value of the increment expression itself is the variable's old value.

The decrement operator `--` works just like `++`, except that it subtracts one instead of adding it.

7.6 Patterns

An awk programs looks like

```
pattern { action }
pattern { action }
...
```

Patterns in awk can be

- `BEGIN`
- `END`

- an expression
- a range

Pattern is optional. If pattern is not specified for a rule, then the action written in curly bracket for that rule will be executed for all the records in the input file.

7.6.1 BEGIN and END

Action specified in the curly brackets after the BEGIN pattern will be executed only once before awk starts processing the input file. It can be used to set field separator or other system variables. For example

```
BEGIN{
  FS=","
}
```

END pattern is used at the conclusion of the script. A common use of this pattern is to print the accumulated result in user-defined variables as in Listing 7.2.

7.6.2 Expressions

Awk supports following expressions

- Regular expressions
- Arithmetic
- Relational
- Logical

Regular Expressions

Awk regular expressions are the regular expressions defined in egrep. The operators required to use regular expressions are

`~`	Regular expression must match the text
`!~`	Regular expression must not match the text

The regular expressions must be enclosed between a pair of forward slashes (/). Some examples are

Pattern	Meaning
`$2 ~ /^N.*/`	Second field must begin with “N"
`$1 !~ /^N.*/`	First field must not begin with “N"

Arithmetic Expressions

When the expression is arithmetic, it matches the record if the value of the arithmetic expression is non-zero and does not match the record if the value of the expression is zero.

Relational Expressions

Relational expressions compare two values. When the two values are numeric, an algebraic comparison is used and when they are string, string comparison is used. The relational operators in awk are given in Table 7.2.

Logical Expressions

Logical expressions are used to combine two or more expressions. The logical operators are given in Table 7.3

7.6.3 Range Patterns

Range pattern is a made up of two simple patterns separated by a comma.

```
start-pattern,end-pattern
```

- The range starts with the record that matches the `start-pattern` and ends with the next record that matches the `end-pattern`.
- If the range pattern matches more than one set of records, then the action is done for each set.
- If the end pattern is not matched with any record, then the action begins with the matching start record and ends with the last record in the file.

For example, consider the following script.

```
$3==30,$3==35{
  print $0
}
```

If this script is run for the input file 7.1, it will carry out the specified action for the record that matches the starting-pattern. Subsequently, it will continue to perform the same action for each successive record until it encounters the one that matches the end-pattern. The output will be

```
A101  Mathematics 30  30  40
A102  Physics     40  40  40
A103  Physics     50  50  50
A104  Physics     35  35  35
```

7.7 Actions

An awk program or script consists of a series of rules. A rule contains a pattern and an action, either of which (but not both) may be omitted. The purpose of the action is to tell awk what to do once a match for the pattern is found.

An action consists of one or more awk statements, enclosed in braces (`{...}`). Each statement specifies one thing to do. The statements are separated by newlines or semicolons. The braces around an action must be used even if the action contains only one statement, or if it contains no statements at all.

The following types of statements are supported in awk:

- **Expressions**
  Call functions or assign values to variables. Executing this kind of statement simply computes the value of the expression.
  e.g. `total = total + $3`
- **Output statements**
  Such as `print` and `printf`.
- **Control statements**
  Specify the control flow of awk programs.
- **Compound statements**
  Enclose one or more statements in braces. A compound statement is used in order to put several statements together in the body of an `if`, `while`, `do`, or `for` statement.
- **Input statements**
  Using the commands like `getline`, `next`.

7.8 Output Statements

7.8.1 The `print` Statement

The `print` statement is used for simple output. The items to print can be constant strings or numbers, fields of the current record (such as `$1`), variables, or any awk expression. Numeric values are converted to strings and then printed. Each `print` action writes a separate line. Multiple fields or variables must be separated with commas. If no data are specified, then the entire record is printed.

```
print item1 , item2 , ...
```

7.8.2 The `printf` Statement

The `printf` statement in awk is same as the `printf` function in C. Each `printf` statement consists of a **format string**, enclosed in double quotes, and a list of zero or more values to be printed. The general form of `printf` statement is:

```
printf("control string",arg1,arg2,....argn);
```

Control string consists of three types of items:

- Characters that will be printed on the screen as they appear
- Format specifications that define the output format for display of each item. This contains \% symbol followed by a conversion code which specifies the type of data being printed. Conversion code for different data types are

`c`	Character
`d`	Integer
`s`	String
`o`, `x`	Octal/Hexadecimal
`f`, `e`, `g`	floating point

- Escape sequence characters such as `\n`, `\t` etc.

The control string indicates how many arguments follow and what their types are. The arguments `arg1, arg2,...argn` are the variables whose values are formatted and printed according to the specifications of the control string. The arguments should match in number, order and type with the format specifications.

7.9 Control Statements

Control statements, such as `if`, `while`, and so on, control the flow of execution in awk programs. Most of awk's control statements are patterned after similar statements in C.

7.9.1 The `if-else` Statement

```
if (condition)
{
  then-body
}
else
{
  else-body
}
```

If the `condition` is true, `then-body` is executed; otherwise, `else-body` is executed. The else part of the statement is optional. The `condition` is considered `false` if its value is zero or the null string; otherwise, the `condition` is `true`. For example,

```
if (x % 2 == 0)
  print "x is even"
else
  print "x is odd"
```

In this example, if the expression `x % 2 == 0` is true, then the first print statement is executed; otherwise, the second print statement is executed.

7.9.2 The `while` Statement

The `while` statement is the simplest looping statement in awk. It repeatedly executes a statement as long as a condition is true.

```
while(condition)
{
  body
}
```

`body` is a statement called the body of the loop, and `condition` is an expression that controls how long the loop keeps running. The `while` statement first tests the `condition`. If the `condition` is true, it executes the statement body. Then, `condition` is tested again, and if it is still true, body executes again. This process repeats until the `condition` is false. For example,

```
BEGIN{
  x=1;
  while(x<=10)
  {
    print x;
    x++;
  }
}
```

This will print numbers from 1 to 10.

7.9.3 The `do-while` Statement

The do loop is a variation of the while looping statement. The do loop executes the body once and then repeats the body as long as the condition is true.

```
do
{
  body
}while(condition)
```

Even if the condition is false at the start, the body executes at least once.

7.9.4 The `for` Statement

General form is

```
for (initialization; condition; alteration)
  body
```

The `initialization`, `condition`, and `alteration` parts are awk expressions, and `body` stands for any awk statement.

The `for` statement starts by executing `initialization`. Then, as long as the `condition` is true, it repeatedly executes `body` and then `alteration`. For example,

```
BEGIN{
  for(i=1;i<=10;i++)
  {
    print i;
  }
}
```

this will print numbers from 1 to 10.

7.9.5 The `break` and `continue` Statements

The `break` statement is used to immediately exit from the loop in which it is written.

```
BEGIN{
  i=0;
  while(i<10)
  {
    i++;
    if(i==5)
    {
      break;
    }
    print i;
  }
}
```

This will print numbers from 1 to 4.

The `continue` statement is used to skip remaining statements in the current iteration of the loop. The program moves immediately to the next iteration of the loop and continues as normal.

```
BEGIN{
  i=0;
  while(i<10)
  {
    i++;
    if(i==5)
    {
      continue;
    }
    print i;
  }
}
```

This will print numbers from 1 to 10 except 5.

7.9.6 The `next` Statement

The `next` statement forces awk to immediately stop processing the current record and go on to the next record. This means that no further rules are executed for the current record, and the rest of the current rule's action is not executed. For example, consider the following script

```
BEGIN{
  total=0
}
$2=="Mathematics"{
  next
}
{
  total+=$3+$4+$5
}
END{
  print total
}
```

Executing this for the input file 7.1 will print the total marks of all students except students of Mathematics and the output will be

```
$ awk -f next.awk marks
471
```

7.9.7 The `exit` Statement

The `exit` statement causes awk to immediately stop executing the current rule and to stop processing input; any remaining input is ignored. The `exit` statement is written as follows:

```
exit [return code]
```

When an `exit` statement is executed from a `BEGIN` rule, the program stops processing everything immediately. No input records are read. However, if an `END` rule is present, the `END` rule is executed. An `exit` statement that is not part of a `BEGIN` or `END` rule stops the execution of any further automatic rules for the current record, skips reading any remaining input records, and executes the `END` rule if there is one.

7.10 String Functions

7.10.1 The `length` Function

The `length` function returns the number of characters in a string. The syntax for the `length` function is

```
length(string)
```

Consider the following input file containing records with only one field

```
mathematics
increase
pin
blue
orange
yellow
```

Listing 7.3: Random Data

The following script will print the number of characters of each record.

```
{print $1,length($1)}
```

The output of this script is

```
$awk -f string.awk random
mathematics 11
increase 8
pin 3
blue 4
orange 6
yellow 6
```

7.10.2 The `index` Function

The `index` function returns the first position of a substring within a string. Syntax is

```
index(string,substring)
```

- The `substring` can be a string constant or variable
- If the `substring` is not found, it returns zero.

The following script will search for the substring “`in`” in each record of the input file 7.3.

```
{print $1,index($1,"in")}
```

The output of this script is

```
$ awk -f string.awk random
mathematics 0
increase 1
```

```
pin 2
blue 0
orange 0
yellow 0
```

7.10.3 The `substr` Function

The `substr` function extracts a substring from a string. It has two formats

```
substr(string,position)
substr(string,position,length)
```

Both formats returns a substring from the `string` starting at `position`. If the `length` is specified, it returns up to `length` characters. If `length` is not specified, it returns everything to the end of the string. The following script will extract substring starting from the third index for the input file 7.3.

```
{print $1,substr($1,3)}
```

The output of this script is

```
$ awk -f string.awk random
mathematics thematics
increase crease
pin n
blue ue
orange ange
yellow llow
```

7.10.4 The `sub` Function

The `sub` function is used to substitute a string with another string. General form is

```
sub(regexp,replacement_string,string)
```

The `sub` function returns 1 (true) if the substitution is successful and returns 0 (false) if the target string is not found. The following script will substitute the regular expression that matches the string with the replacement string for the input file 7.3.

```
{
  sub(/in/,"out",$1)
  print $1
}
```

The output of this script is

```
$ awk -f string.awk random
mathematics
outcrease
pout
blue
orange
yellow
```

7.10.5 The `toupper` and `tolower` Functions

The `toupper` function converts the lowercase letters in a string to uppercase. The `tolower` function converts the uppercase letters in a string to lowercase.

The following script will convert each record into uppercase for the input file 7.3.

```
{print $1, toupper($1)}
```

The output of this script is

```
$ awk -f string.awk random
mathematics MATHEMATICS
increase INCREASE
pin PIN
blue BLUE
orange ORANGE
yellow YELLOW
```