14

AWK Tool in Unix

AWK was developed in 1978 at the famous Bell Laboratories by Aho, Weinberger, and Kernighan [4][1] to process structured data files. In programming languages it is very common to have a definition of a *record* which may have one or more data fields. In this context, it is common to define a file as a collection of records. Records are structured data items arranged in accordance with some specification, basically as a pre-assigned sequence of fields. The data fields may be separated by a space or a tab.

In a data processing environment it is very common to have such record-based files. For instance, an organisation may maintain a personnel file. Each record may contain fields like employee name, gender, date of joining the organisation, designation, etc. Similarly, if we look at files created to maintain pay accounts, student files in universities, etc., all have structured records with a set of fields.

AWK is ideal for the data processing of such structured set of records. AWK comes in many flavours [22]. There is *gawk* which is GNU AWK. Presently, we will assume the availability of the standard AWK program which comes bundled with every flavour of the Unix OS. AWK is also available in the MS environment.

14.1 THE DATA TO PROCESS

As AWK is used to process a structured set of records, we shall use a small file called *awk.test* given below. It has a structured set of records. The data in this file lists employee name, employee's hourly wage, and the number of hours the employee has worked.

(File awk.test)

```
bhatt    4.00    0
ulhas    3.75    2
ritu     5.0     4
vivek    2.0     3
```

We will use this candidate data file for a variety of processing re
need to compute the amount due to each employee and print i
write a C language program to do the task. However, using a
simpler and perhaps smarter. Note that if we have a tool, then
to use it. This is because it takes less time to get the results. Al
less error prone. Let us use the *awk* command with input file

```
bhatt@falerno [CRUD] =>awk '$3 > 0 {print $1, $2 * $3}' awk.test
ulhas   7.5
ritu    20
vivek   6
```

Note some features of the syntax above—the awk command, th
it, and the data file name. We shall next discuss first a few s
advanced features are explained through examples that are di

14.1.1 AWK Syntax

To run an AWK program we simply give an *awk* command w

```
awk [options] <awk_program> [input_file]
```

where the options may be like a file input instead of a quoted st
be noted:

- Note that in the syntax awk 'awk_program' [input_files], the
  be empty. That suggests that *awk* would take whatever
  the command is given.
- Also, note that fields in the data file are identified with
  In the example above we have a very small AWK progr
  reproduced below:

  '$3 > 0 {print $1, $2 * $3}'

  The interpretation is to print the name corresponding
  by taking a product of rate corresponding to $2 mult
  hours corresponding to $3. In this string the $ prefixed
  we wish to use.
- In preparing the output, {print} or {print $0} prints
  {print $1, $3} will print the selected fields.

In the initial example we had a one line awk program. Bas
pattern and check if that qualified the line for some processi
may have many patterns to match and actions to take on fin
that case the awk program may have several lines of code. Ty
have the following structure:

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

AWK Tool in Unix

at the famous Bell Laboratories by Aho, Weinberger, and
uctured data files. In programming languages it is very
f a *record* which may have one or more data fields. In this
a file as a collection of records. Records are structured data
ith some specification, basically as a pre-assigned sequence
be separated by a space or a tab.
nment it is very common to have such record-based files.
may maintain a personnel file. Each record may contain
nder, date of joining the organisation, designation, etc.
ted to maintain pay accounts, student files in universities,
s with a set of fields.
processing of such structured set of records. AWK comes
gawk which is GNU AWK. Presently, we will assume the
K program which comes bundled with every flavour of the
in the MS environment.

ESS

ructured set of records, we shall use a small file called
uctured set of records. The data in this file lists employee
and the number of hours the employee has worked.

the first character in the names of the authors of AWK.

to use it. This is because it takes less time to get the results. Also, the process is usually less error prone. Let us use the *awk* command with input file awk.test as shown below:

```
bhatt@falerno [CRUD] =>awk '$3 > 0 {print $1, $2 * $3}' awk.test
utras 7.5
ritu 20
vivek 6
```

Note some features of the syntax above—the awk command, the quoted string following it, and the data file name. We shall next discuss first a few simple syntax rules. More advanced features are explained through examples that are discussed in Section 14.2.

14.1.1 AWK Syntax

To run an AWK program we simply give an *awk* command with the following syntax:

```
awk [options] <awk_program> [input_file]
```

where the options may be like a file input instead of a quoted string. The following should be noted:

- Note that in the syntax awk 'awk_program' [input_files], the option on input files may be empty. That suggests that *awk* would take whatever is typed immediately after the command is given.
- Also, note that fields in the data file are identified with a $ symbol prefix as in $1. In the example above we have a very small AWK program. It is the quoted string reproduced below:

  '$3 > 0 {print $1, $2 * $3}'

  The interpretation is to print the name corresponding to $1, and the wages due by taking a product of rate corresponding to $2 multiplied with the number of hours corresponding to $3. In this string the $ prefixed integers identify the fields we wish to use.
- In preparing the output, {print} or {print $0} prints the whole line of output. {print $1, $3} will print the selected fields.

In the initial example we had a one line awk program. Basically, we tried to match a pattern and check if that qualified the line for some processing or action. In general, we may have many patterns to match and actions to take on finding a matching pattern. In that case the awk program may have several lines of code. Typically, such a program shall have the following structure:

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

If we have many operations to perform, we shall have many lines in the AWK program. It would be then imperative to put such a program in a file and AWKing it would require using a file input option as shown below. So if the awk_program is very long and kept in a file, use the –f option as shown below:

```
awk –f 'awk_program_file_name' [input_files]
```

where the awk_program_file_name contains the awk program.

14.2 PROGRAMMING EXAMPLES

We shall now give a few illustrative examples. Along with the examples we shall also discuss many other features that make the task of processing easier.

Example 1
Suppose we now need to find out if there was an employee who did no work. Clearly, his hours work field should be equal to 0. We show the AWK program to get that.

```
bhatt@falerno [CRUD] =>awk '$3 == 0 {print $1}' awk.test
bhatt
```

The basic operation here was to scan a sequence of input lines searching for the lines that match any of the patterns in the program. Patterns like $3 > 0 match the 3rd field when the field has a value > 0 in it.

An aside. Try a few errors and see the error detection in one line awk programs.

Example 2
In this example we shall show the use of some of the built-in variables which help in organising our data processing needs. These variables acquire meaning in the context of the data file. NF is a built-in variable which stores the number of fields and can be used in such context as {print NF, $1, $NF} which prints the number of fields, the first and the last field. Another built-in variable is NR, which takes the value of the number of lines read so far and can also be used in a print statement.

```
bhatt@falerno [CRUD] =>awk '$3 > 0 {print NR, NF, $1, $NF }' awk.test
3 3 ulhas 2
4 3 ritu 4
5 3 vivek 3
```

Example 3
The formatted data in files is usually devoid of any redundancy. However, one needs to generate a verbose output. This requires that we get the values and interspread the desired strings and generate a verbose and meaningful output. In this example, we will demonstrate such a usage.

```
bhatt@falerno [CRUD] =>awk '$3 > 0 {print "person", NR, $1, "be paid", $2*$3, "dollars"}' awk.test
person  3 ulhas be paid 7.5 dollars
person  4 ritu be paid 20 dollars
```

One can use *printf* to format the output like in the *C* program

```
bhatt@falerno [CRUD] =>awk '$3 > 0 {printf("%–8s be paid $%6.2f dollars \n", $1, $
ulhas   be paid $  7.50 dollars
ritu    be paid $ 20.00 dollars
vivek   be paid $  6.00 dollars
```

An aside: One could sort the output by <awk_program> | so program is piped to sort utility to get sorted result.

Example 4
In the examples below we basically explore many selection pos selection of lines may be by comparison involving computation. use $2 > 3.0 to mean if the rate of payment is greater than 3.0 total due is > 5, as $2*$3 > 5.0, which is an example of comp

One may also use a selection by text content (essentially cor This is done by enclosing the test pattern as /bhatt/ to identify as in $1 == /bhatt/.

Tests on patterns may involve relational or logical operato AWK is excellent for data validation. Checks like the follov

- NF != 3 . . . no. of fields not equal to 3
- $2 < 2.0 . . . wage rate below min. stipulated
- $2 > 10.0 . . . exceeding max. . . .
- $3 < 0 . . . no. of hrs worked –ve etc.

It should be remarked that data validation checks are a very processing activity. Often an organisation may employ or outsou on-line data processing may result in disasters if the data is not with a wrong hourly wage field we may end up creating a pa wrong. One needs to ensure that the data is in expected range l up paying at a rate below the minimum legal wage or payi amounts to a low-paid worker!

Example 5
In these examples we demonstrate how we may prepare addi formatted data a look of a report under preparation. For ins headings for the tabulated output. One can generate meaningful a tabulated output. Usually, an AWK program may have a BEC some pre-processing that can help prepare headers before pr Similarly, an AWK program may be used to generate a trailer w next example illustrates such a usage. For our example the heac putting BEGIN {print "Name Rate Hours"} as preamble to the AWK p

```
bhatt@falerno [CRUD] =>awk 'BEGIN{ print"name rate hours"; print'' ''}
{print}' awk.test
name  rate  hours

bhatt    4.00      0
ulhas    3.75      2
```

ile_name contains the awk program.

EXAMPLES

llustrative examples. Along with the examples we shall also
es that make the task of processing easier.

id out if there was an employee who did no work. Clearly, his
equal to 0. We show the AWK program to get that.

```
0 {print $1}' awk.test
```

s to scan a sequence of input lines searching for the lines that
the program. Patterns like $3 > 0 match the 3rd field when
it.

s and see the error detection in one line awk programs.

ow the use of some of the built-in variables which help in
g needs. These variables acquire meaning in the context of
variable which stores the number of fields and can be used
$1, $NF} which prints the number of fields, the first and
variable is NR, which takes the value of the number of lines
sed in a print statement.

```
rint NR, NF, $1, $NF }' awk.test
```

usually devoid of any redundancy. However, one needs to
is requires that we get the values and interspread the
verbose and meaningful output. In this example, we will

```
"person", NR, $1, "be paid", $2*$3, "dollars"}' awk.test
```

```
be paid $ 6.00 dollars
```

An aside: One could sort the output by <awk_program> | sort i.e. the output of AWK program is piped to sort utility to get sorted result.

Example 4

In the examples below we basically explore many selection possibilities. In general, the selection of lines may be by comparison involving computation. As an example, we may use $2 > 3.0 to mean if the rate of payment is greater than 3.0. We may check for if the total due is > 5, as $2*$3 > 5.0, which is an example of comparison by computation.

One may also use a selection by text content (essentially comparison in my opinion). This is done by enclosing the test pattern as /bhatt/ to identify $1 being string "bhatt" as in $1 == /bhatt/.

Tests on patterns may involve relational or logical operators as $>=, || .

AWK is excellent for data validation. Checks like the following may be useful.

- NF != 3 . . . no. of fields not equal to 3
- $2 < 2.0 . . . wage rate below min. stipulated
- $2 > 10.0 . . . exceeding max. . . .
- $3 < 0 . . . no. of hrs worked –ve etc.

It should be remarked that data validation checks are a very important part of data processing activity. Often an organisation may employ or outsource data preparation. An on-line data processing may result in disasters if the data is not validated. For instance, with a wrong hourly wage field we may end up creating a pay cheque which may be wrong. One needs to ensure that the data is in expected range lest an organisation ends up paying at a rate below the minimum legal wage or paying extraordinarily high amounts to a low-paid worker!

Example 5

In these examples we demonstrate how we may prepare additional pads to give the formatted data a look of a report under preparation. For instance, we do not have headings for the tabulated output. One can generate meaningful headers and trailers for tabulated output. Usually, an AWK program may have a BEGIN keyword to identify some pre-processing that can help prepare headers before processing the data file. Similarly, an AWK program may be used to generate a trailer with END keyword. The next example illustrates such a usage. For our example the header can be generated by putting BEGIN {print "Name Rate Hours"} as preamble to the AWK program as shown below.

```
@faleno [CRUD] =>awk 'BEGIN{ print"name rate hours"; print"" ""}
int}' awk.test
me  rate  hours
    4.00    0
    3.75    2
    5.0     4
ek  2.0     3
```

Note that print " " prints a blank line and the next print reproduces the input. In general, BEGIN matches before the first line of input and END after the last line of input. The ; is used to separate the actions. Let us now look at a similar program with –f option.

file awk.prg is

```
BEGIN {print "NAME      RATE     HOURS"; print " "}
{print $1,"   ",$2,"   ",$3,"..."}
```

```
bhatt@falerno [CRUD] =>!a
awk –f awk.prg awk.test
NAME      RATE      HOURS

bhatt       4.00      0 . . .
ulhas       3.75      2 . . .
ritu        5.0       4 . . .
vivek       2.0       3 . . .
```

Example 6

Now we shall attempt some computing within awk. To perform computations we may sometimes need to employ user-defined variables. In this example, "pay" is a user-defined variable. The program accumulates the total amount to be paid in "pay". So the printing is done after the last line in the data file has been processed, i.e. in the END segment of awk program. In NR we obtain all the records processed (so the number of employees can be determined). We are able to do the computations like "pay" as a total as well as compute the average salary as the last step.

```
BEGIN {print "NAME RATE HOURS"; print " "}
{pay = pay + $2*$3}
END {print NR "employees"
     print "total amount paid is : ", pay
     print "with the average being :", pay/NR}
```

```
bhatt@falerno [CRUD] =>!a
awk –f prg2.awk awk.test
NAME      RATE      HOURS

4 employees
total amount paid is :      33.5
with the average being :    8.375
```

A better looking output could be produced by using *printf* statement as in *c*.

Here is another program with its output. In this program, note the computation of "maximum" values and also the concatenation of names in "emplist". These are user-defined data-structures. Note also the use of "last" to store the last record processed, i.e. $0 gets the record and we keep storing it in last as we go along.

```
BEGIN {print "NAME RATE HOURS"; print " "}
{pay = pay + $2*$3}
  $2 > maxrate {maxrate = $2; maxemp = $1}
  {emplist = emplist $1 " "}
END {print NR " employees"
     print "total amount paid is : ", pay
     print "with the average being :", pay/NR
     print "highest paid rate is for " maxemp, " @ of : ", maxrate
     print emplist
     print " "
     print "the last employee record is : ", last}
```

output is .....

```
bhatt@falerno [CRUD] =>!a
awk –f prg3.awk test.data
NAME      RATE      HOURS

4 employees
total amount paid is :  33.5
with the average being : 8.375
highest paid rate is for ritu  @ of :  5.0
bhatt ulhas ritu vivek

the last employee record is :  vivek  2.0   3
```

Example 7

There are some built-in functions that can be useful. For in
helps one to compute the length of the argument field as the
field. See the program and the corresponding output below

```
{nc = nc + length($1) + length($2) + length($3) + 4}
{nw = nw + NF}
END {print nc " characters and "; print " "
     print nw " words and "; print " "
     print NR, " lines in this file "}
```

```
bhatt@falerno [CRUD] =>!a
awk –f prg4.awk test.data
53 characters and

12 words and

4  lines in this file
```

Example 8

AWK supports many control flow statements to facilitate pr
the *if-else* construct. Note the absence of "then" and how th
the case when the *if condition* evaluates to *true*. Also, in the
against division by 0.

```
BEGIN {print "NAME      RATE     HOURS"; print " "}
$2 > 6 {n = n+1; pay = pay + $2*$3}
$2 > maxrate {maxrate = $2; maxemp = $1}
{emplist = emplist $1 " "}
{last = $0}
```

```
HOURS"; print " "}
```

computing within awk. To perform computations we may
er-defined variables. In this example, "pay" is a user-define
ulates the total amount to be paid in "pay". So the printing
he data file has been processed, i.e. in the END segment o
n all the records processed (so the number of employees can
to do the computations like "pay" as a total as well as
s the last step.

```
print " "}
```

```
ay/NR}
```

be produced by using *printf* statement as in *c*.
ith its output. In this program, note the computation o
ıe concatenation of names in "emplist". These are user-
lso the use of "last" to store the last record processed, i.e.
o storing it in last as we go along.

```
rint " "}
```

```
$1}
```

```
print emplist
print " "
print "the last employee record is : ", last}
```

```
utput is .....

hatt@falerno [CRUD] =>!a
wk -f prg3.awk test.data
AME    RATE    HOURS

employees
tal amount paid is :  33.5
th the average being : 8.375
ghest paid rate is for ritu  @ of :  5.0
hatt ulhas ritu vivek

e last employee record is :  vivek  2.0  3
```

Example 7

There are some built-in functions that can be useful. For instance, the function "length" helps one to compute the length of the argument field as the number of characters in that field. See the program and the corresponding output below:

```
nc = nc + length($1) + length($2) + length($3) + 4}
nw = nw + NF}
ND {print nc " characters and "; print " "
    print nw " words and "; print " "
    print NR, " lines in this file "}
```

```
hatt@falerno [CRUD] =>!a
wk -f prg4.awk test.data
3 characters and

12 words and

4 lines in this file
```

Example 8

AWK supports many control flow statements to facilitate programming. We will first use the *if-else* construct. Note the absence of "then" and how the statements are grouped for the case when the *if condition* evaluates to *true*. Also, in the program, note the protection against division by 0.

```
EGIN {print "NAME      RATE      HOURS"; print " "}
2 > 6 {n = n+1; pay = pay + $2*$3}
2 > maxrate {maxrate = $2; maxemp = $1}
mplist = emplist $1 " "}
last = $0}
ND {print NR " employees in the company "
```

```
    if (n > 0) {print n, "employees in this bracket of salary. "
                print "with an average salary of ", pay/n, "dollars"
}   else print " no employee in this bracket of salary. "
    print "highest paid rate is for " maxemp, " @ of : ", maxrate
    print emplist
    print " " "}
```

This gives the result shown below:

```
bhatt@falerno [CRUD] =>!a
awk –f prg5.awk data.awk
NAME      RATE      HOURS

4 employees in the company
 no employee in this bracket of salary.
 highest paid rate is for ritu  @ of :  5.0
 bhatt ulhas ritu vivek
```

Next we shall use a "while" loop.[2] In this example, we simply compute the compound interest that accrues each year for a five-year period.

```
#compound interest computation
#input: amount rate years
#output: compounded value at the end of each year
{i = 1; x = $1;
    while (i <= $3)
            {x = x + (x*$2)
     printf("\t%d\t%8.2f\n",i, x)
     i = i + 1
    }
}
```

The result is shown below:

```
bhatt@falerno [CRUD] =>!a
awk –f prg6.awk
1000 0.06 5
    1       1060.00
    2       1123.60
    3       1191.02
    4       1262.48
    5       1338.23
```

AWK also supports a "for" statement as in for (i = 1; i <= $3; i = i + 1) =, which will have the same effect. AWK supports arrays, too, as the program below demonstrates.

```
# reverse – print the input in reverse order ...
BEGIN {print "NAME      RATE      HOURS"; print " " "}
{line_ar [NR] = $0}  # remembers the input line in array line_ar
END {# prepare to print in reverse order as input is over now
     for (i = NR; i >= 1; i = i–1)
     print line_ar[i]
}
```

The result is shown below.

```
bhatt@falerno [CRUD] =>awk –f prg7.awk data.awk
NAME      RATE      HOURS

vivek     2.0       3
ritu      5.0       4
ulhas     3.75      2
bhatt     4.00      0
```

14.2.1 Some One-liners

Next we mention a few one-liners that are now folklore in t community [4], [22]. It helps to remember some of these at the ti in AWK.

1. Print the total no. of input lines: END {print NR}.
2. Print the 10th input line: NR = 10.
3. Print the last field of each line: \{print \$NF\}.
4. Print the last field of the last input line:

```
{field = $NF}
      END {print field}
```

5. Print every input line with more than 4 fields: NF > 4.
6. Print every input line in which the last field is more tha
7. Print the total number of fields in all input lines.

```
{nf = nf + NF}
END   {print  nf}
```

8. Print the total number of lines containing the string "bl

```
/bhatt/ {nlines = nlines + 1}
END  {print nlines}
```

9. Print the largest first field and the line that contains it.

```
$1 > max {max = $1; maxline = $0}
END {print max, maxline}
```

10. Print every line that has at least one field: NF > 0.
11. Print every line with > 80 characters: length($0) > 80.
12. Print the number of fields followed by the line itself.

```
{print NF, $0}
```

13. Print the first two fields in opposite order: {print $2, $1
14. Exchange the first two fields of every line and then prin

```
{temp = $1; $1 = $2; $2 = temp; print}
```

15. Print every line with the first field replaced by the line n

```
$1 = NR; print}
```

'n below:

'.
5.0

" loop.[2] In this example, we simply compute the compound
'ear for a five-year period.

of each year

tement as in for (i = 1; i <= $3; i = i + 1) =, which will
)ports arrays, too, as the program below demonstrates.

...
'RS"; print "" ""}
line in array line_ar
; input is over now

'perator so I went around it ... it can be easily detected!

```
vivek     2.0     3
ntu       5.0     4
uhas      3.75    2
bhatt     4.00    0
```

14.2.1 Some One-liners

Next we mention a few one-liners that are now folklore in the AWK programming community [4], [22]. It helps to remember some of these at the time of writing programs in AWK.

1. Print the total no. of input lines: END {print NR}.
2. Print the 10th input line: NR = 10.
3. Print the last field of each line: \{print \$NF\}.
4. Print the last field of the last input line:

```
{field = $NF}
     END {print field}
```

5. Print every input line with more than 4 fields: NF > 4.
6. Print every input line in which the last field is more than 4: $NF > 4.
7. Print the total number of fields in all input lines.

```
{nf = nf + NF}
END   {print  nf}
```

8. Print the total number of lines containing the string "bhatt".

```
/bhatt/ {nlines = nlines + 1}
END  {print nlines}
```

9. Print the largest first field and the line that contains it.

```
$1 > max {max = $1; maxline = $0}
END {print max, maxline}
```

10. Print every line that has at least one field: NF > 0.
11. Print every line with > 80 characters: length($0) > 80.
12. Print the number of fields followed by the line itself.

```
{print NF, $0}
```

13. Print the first two fields in opposite order: {print $2, $1}.
14. Exchange the first two fields of every line and then print the line:

```
{temp = $1; $1 = $2; $2 = temp; print}
```

15. Print every line with the first field replaced by the line number:

```
$1 = NR; print}
```

16. Print every line after erasing the second field:

```
{$2 = "" ""; print}
```

17. Print in reverse order the fields of every line:

```
{for (i = NF; i > 0; i = i–1) printf("%s ", $i)
 printf("\n")}
```

18. Print the sums of fields of every line:

```
{sum = 0
 for ( i = 1; i <= NF; i = i+1 ) sum = sum + $i
 print sum}
```

19. Add up all the fields in all the lines and print the sum:

```
{for ( i = 1; i <= NF; i = i+1 ) sum = sum + $i}
END {print sum}
```

20. Print every line after replacing each field by its absolute value:

```
{for( i = 1; i <= NF; i = i+1) if ($i < 0) $i = –$i
 print}
```

14.3 AWK GRAMMAR

At this stage it may be worth our while to recapitulate some of the grammar rules. In particular we shall summarise the patterns which commonly describe the AWK grammar [4]. The reader should note how right across all the tools and utilities, Unix maintains the very same regular expression conventions for programming.

1. BEGIN {statements}: These statements are executed once, before any input is processed.
2. END {statements}: These statements are executed when all the lines in the data input file have been read.
3. expr. {statements}: These statements are executed at each input line where the expr is true.
4. /regular expr/{statements}: These statements are executed at each input line that contains a string matched by a regular expression.
5. compound pattern {statements}: A compound pattern combines patterns with && (AND), || (OR) and ! (NOT) and parentheses; the statements are executed at each input line where the compound pattern is true.
6. pattern1, pattern2 {statements}: A range pattern matches each input line from a line matched by "pattern1" to the next line matched by "pattern2", inclusive; the statements are executed at each matching line.
7. "BEGIN" and "END" do not combine with any other pattern. "BEGIN" and "END" also always require an action. Note "BEGIN" and "END" technically do not match any input line. With multiple "BEGIN" and "END" the actions happen in the order of their appearance.
8. A range pattern cannot be part of any other pattern.
9. "FS" is a built-in variable for field separator.

Note that expressions like $3/$2 > 0.5 = match when they evaluate to true. Also, "The" < "Then" and "Bonn" > "Berlin". Now let us look at some string matching

1. /regexpr/ matches an input line if the line contains th
   example: /India/ matches "India" (with space on both
   presence of India in "Indian".
2. expr ~ /regexpr/ matches, if the value of the expr conta
   regexpr. As an example, $4 ~ /India/ matches all input
   contains "India" as a substring.
3. expr !~/regexpr/ same as above except that the conditio
   example, $4 !~/India/ matches when the fourth field doe

The following is the summary of the Regular Expression (

```
^C      :   matches a character C at the beginning of a string
C$      :   matches a character C at the end of a string
^C$     :   matches a string consisting of the single character C
^.$     :   matches single character strings
^...$   :   matches exactly three character strings
...     :   matches any three consecutive characters
\.$     :   matches a period at the end of a string
*       :   zero or more occurrences
?       :   zero or one  occurrence
+       :   one or more occurrence
```

The regular expression meta-characters are:

1. \ ^ $. [ ] | ( ) * + ?
2. A basic RE is one of the following:

```
A nonmeta-character such as A that matches itself
An escape sequence that matches a special symbol: \t matches a tab
A quoted meta-character such as \* that matches meta-ch literally
^, which matches beginning of a string
$, which matches end of a string
., which matches any single character
A character class such as [ABC] matches any of the A or B or C
Character abbreviations  [A–Za–z] matches any single character
A complemented character class such as [^0–9] matches non-digit char
```

3. These operators combine REs into larger ones:

```
alternation      :  A|B matches A or B
concatenation    :  AB matches A immediately followed by B
closure          :  A* matches zero or more A's
positive closure :  A+ matches one or more A's
zero or one      :  A? matches null string or one A
parentheses      :  (r) matches same string as r does
```

The operator precedence being |, concatenation, (*, +, ?) in increasin

The escape sequences are:

\b : backspace; \f : form feed; \n : new-line; \c literally c (\\ for \)
\r : carriage return; \t : tab; \ddd octal value ddd

= i+1) sum = sum + $i

ds in all the lines and print the sum:

```
i+1 ) sum = sum + $i}
```

ter replacing each field by its absolute value:

```
+1) if ($i < 0) $i = –$i
```

rth our while to recapitulate some of the grammar rules. In
rise the patterns which commonly describe the AWK grammar
e how right across all the tools and utilities, Unix maintains
ression conventions for programming.

These statements are executed once, before any input is

se statements are executed when all the lines in the data input

ese statements are executed at each input line where the expr

: These statements are executed at each input line that
tched by a regular expression.
ents}: A compound pattern combines patterns with && (AND)
parentheses; the statements are executed at each input line
pattern is true.
ents}: A range pattern matches each input line from a line
1" to the next line matched by "pattern2", inclusive; the
ted at each matching line.
" do not combine with any other pattern. "BEGIN" and
equire an action. Note "BEGIN" and "END" technically do
ine. With multiple "BEGIN" and "END" the actions happen
appearance.
ot be part of any other pattern.
iable for field separator.

$3/$2 > 0.5 = match when they evaluate to true. Also,
" > "Berlin". Now, let us look at some string matching
ms, the following rules apply.

regexpr. As an example, $4 ~ /India/ matches all input lines where the fourth field contains "India" as a substring.

3. expr !~/regexpr/ same as above except that the condition of match is opposite. As an example, $4 !~/India/ matches when the fourth field does not have a substring India.

The following is the summary of the Regular Expression (RE) matching rules.

```
^C      :   matches a character C at the beginning of a string
C$      :   matches a character C at the end of a string
^C$     :   matches a string consisting of the single character C
^.$     :   matches single character strings
^...$   :   matches exactly three character strings
...     :   matches any three consecutive characters
\.$     :   matches a period at the end of a string
*       :   zero or more occurrences
?       :   zero or one  occurrence
+       :   one or more occurrence
```

The regular expression meta-characters are:

1. \ ^ $. [ ] | ( ) * + ?
2. A basic RE is one of the following:

   A nonmeta-character such as A that matches itself
   An escape sequence that matches a special symbol: \t matches a tab
   A quoted meta-character such as * that matches meta-ch literally
   ^, which matches beginning of a string
   $, which matches end of a string
   ., which matches any single character
   A character class such as [ABC] matches any of the A or B or C
   Character abbreviations [A–Za–z] matches any single character
   A complemented character class such as [^0–9] matches non-digit characters

3. These operators combine REs into larger ones:

```
alternation       :   A|B matches A or B
concatenation     :   AB matches A immediately followed by B
closure           :   A* matches zero or more A's
positive closure  :   A+ matches one or more A's
zero or one       :   A? matches null string or one A
parentheses       :   (r) matches same string as r does
```

The operator precedence being |, concatenation, (*, +, ?) in increasing value. i.e. *, +, ? bind stronger than |

The escape sequences are:

\b ; backspace; \f : form feed; \n : new-line; \c literally c (\\ for \)
\r : carriage return; \t : tab; \ddd octal value ddd

Some useful patterns are:

/^[0–9]+$/	: matches any input that consists of only digits
/^(\+|–)?[0–9]+\.?[0–9]/	: matches a number with optional sign and optional fractional part
/^[A–Za–z][A-Za–z0–9]*$/	: matches a letter followed by any letters or digits, an awk variable
/^[A–Za–z]$\|^[A–Za–z][A-Za-z0–9]*$/	: matches a letter or a letter followed by any letters or digits (a variable in Basic)

Now we will see the use of FILENAME, a built-in variable, and the use of range operators in the RE.

14.3.1 More Examples

For the next set of examples we shall consider a file which has a set of records describing the performance of some of the well-known cricketers of the recent past. This data is easy to obtain from any of the cricket sites like khel.com or cricinfo.com or the International Cricket Council's website. The data we shall be dealing with is described in Table 14.1.

Table 14.1 The cricket data file

Name	*Country*	*Matches*	*Runs*	*Average*	*Highest*	*100s*	*Wkts*	*Bowlavg*	*RPO*	*Best*	*Catches*
SGavaskar	IND	125	10122	51.12	236	34	1	206.0	3.25	1–34	108
MAmarnath	IND	69	4378	42.50	138	11	32	55.69	2.91	4–43	47
BSBedi	IND	67	656	8.99	50	0	266	28.71	2.14	10–194	26
Kapildev	IND	131	5248	31.05	163	8	434	29.65	2.78	11–146	64
SalimMalik	PAK	103	5768	43.70	237	15	5	82.8	3.38	1–3	65
ImranKhan	PAK	88	3807	37.69	136	6	362	22.81	2.55	14–116	28
MarkTaylor	AUS	104	7525	43.5	334	19	1	26.0	3.71	1–11	157
DLillie	AUS	70	905	13.71	73	0	355	23.92	2.76	11–123	23
DBradman	AUS	52	6996	99.94	334	29	2	36.0	2.7	1–15	32
ABorder	AUS	156	11174	50.56	265	27	39	39.10	2.28	11–96	156
MHolding	WI	60	910	13.75	73	0	249	23.69	2.79	14–149	22
CliveLlyod	WI	110	7515	46.68	242	19	10	62.20	2.17	2–22	90
VRichards	WI	121	8540	50.24	291	24	32	61.38	2.28	3–51	122
GBoycott	ENG	108	8114	47.73	246	22	7	54.57	2.43	3–47	33
MartinCrowe	NZ	77	5444	45.37	299	17	14	48.29	2.95	3–107	71
RHadlee	NZ	86	3124	27.17	151	2	431	22.30	2.63	15–123	39

In Table 14.1, we have information such as the name of the player, his country affiliation, matches played, runs scored, wickets taken, etc. We should be able to scan the cricket.data file and match lines to yield desired results. For instance, if we were to look for players with more than 10,000 runs scored, we shall expect to see SGavaskar and ABorder. Similarly, for anyone with more than 400 wickets we should expect to see Kapildev and RHadlee.[3]

So let us begin with our example programs.

Example 1

```
# In this program we identify Indian cricketers and mark them with ***.
# We try to find cricketers with most runs, most wickets and most catches
BEGIN {FS = "\t" # make the tab as field separator
printf("%12s %5s %7s %4s %6s %7s %4s %8s %8s %3s %7s %7s\n\n",
"Name","Country","Matches","Runs","Batavg","Highest","100s","Wkts",
"Bowlavg","Rpo","Best","Catches")}
$2 ~/IND/ { printf("%12s %5s %7s %6s %6s %7s %4s %8s %8s %4s %7s %
$1,$2,$3,$4,$5,$6,$7,$8,$9,$10,$11,$12,"***")}
$4 > runs {runs = $4;name1 = $1}
$8 > wickets {wickets = $8;name2 = $1}
$12 > catches {catches = $12;name3 = $1}
END
{printf("\n %15s is the highest scorer with          %6s runs",name1,runs)
 printf("\n %15s is the highest wicket taker with     %8s wickets",name2,wickets)
 printf("\n %15s is the highest catch taker with      %7s catches\n",name3,catches
}
bhatt@falerno [AWK] =>!a
awk –f cprg.1 cricket.data
```

Name	Country	Matches	Runs	Batavg	Highest	100s	Wkts	Bowl
SGavaskar	IND	125	10122	51.12	236	34	1	206
MAmarnath	IND	69	4378	42.50	138	11	32	55
BSBedi	IND	67	656	8.99	50	0	266	28
Kapildev	IND	131	5248	31.05	163	8	434	29

```
     ABorder is the highest scorer with          11174 runs
     Kapildev is the highest wicket taker with     434 wickets
     MTaylor is the highest catch taker with       157 catches
```

Example 2

In this example we use the built-in variable FILENAME and a

```
# In this example we use FILENAME built in variable and print data from
# first three lines of cricket.data file. In addition we print data from
# ImranKhan to Allen Border

BEGIN {FS = "\t"  # make the tab as the field separator
      printf("%25s \n","First three players")
NR == 1, NR == 5 {print FILENAME ": " $0}
{printf("%12s %5s \n", $1,$2)}
/ImranKhan/, /ABorder/ {num = num + 1; line[num] = $0}
END
{printf("Player list from ImranKhan to Allen Border \n",
 printf("%12s %5s %7s %4s %6s %7s %4s %8s %8s %3s %7s %7s\n\n",
"Name","Country","Matches","Runs","Batavg","Highest","100s","Wkts",
"Bowlavg","Rpo","Best","Catches")}
     for(i=1; i <= num; i = i+1)
     print(line[i] ) }

bhatt@falerno [AWK] =>!a
```

ıse of FILENAME, a built-in variable, and the use of range

:s

s we shall consider a file which has a set of records describing
f the well-known cricketers of the recent past. This data is
he cricket sites like khel.com or cricinfo.com or the International
The data we shall be dealing with is described in Table 14.1.

Table 14.1 The cricket data file

uns	*Average*	*Highest*	*100s*	*Wkts*	*Bowlavg*	*RPO*	*Best*	*Catches*
0122	51.12	236	34	1	206.0	3.25	1–34	108
378	42.50	138	11	32	55.69	2.91	4–43	47
656	8.99	50	0	266	28.71	2.14	10–194	26
248	31.05	163	8	434	29.65	2.78	11–146	64
768	43.70	237	15	5	82.8	3.38	1–3	65
807	37.69	136	6	362	22.81	2.55	14–116	28
525	43.5	334	19	1	26.0	3.71	1–11	157
005	13.71	73	0	355	23.92	2.76	11–123	23
996	99.94	334	29	2	36.0	2.7	1–15	82
74	50.56	265	27	39	39.10	2.28	11–96	156
10	13.75	73	0	249	23.69	2.79	14–149	22
15	46.68	242	19	10	62.20	2.17	2–22	90
40	50.24	291	24	32	61.38	2.28	3–51	122
14	47.73	246	22	7	54.57	2.43	3–47	83
14	45.37	299	17	14	48.29	2.95	3–107	71
24	27.17	151	2	431	22.30	2.63	15–123	99

ation such as the name of the player, his country affiliation,
wickets taken, etc. We should be able to scan the cricket.data
lesired results. For instance, if we were to look for players
scored, we shall expect to see SGavaskar and ABorder.
re than 400 wickets we should expect to see Kapildev and

Courtney Walsh.

```
printf("%12s %5s %7s %4s %6s %7s %4s %8s %8s %3s %7s %7s\n\n",
Name","Country","Matches","Runs","Batavg","Highest","100s","Wkts",
Bowlavg","Rpo","Best","Catches")}
2 ~/IND/ { printf("%12s %5s %7s %6s %6s %7s %4s %8s %8s %4s %7s %7s %3s\n",
1,$2,$3,$4,$5,$6,$7,$8,$9,$10,$11,$12,"***")}
4 > runs {runs = $4;name1 = $1}
8 > wickets {wickets = $8;name2 = $1}
12 > catches {catches = $12;name3 = $1}
END
printf("\n %15s is the highest scorer with          %6s runs",name1,runs)
printf("\n %15s is the highest wicket taker with    %8s wickets",name2,wickets)
printf("\n %15s is the highest catch taker with     %7s catches\n",name3,catches)
}
hatt@falerno [AWK] =>!a
wk -f cprg.1 cricket.data
```

Name	Country	Matches	Runs	Batavg	Highest	100s	Wkts	Bowlavg	RPO	Best	Catches
SGavaskar	IND	125	10122	51.12	236	34	1	206.00	3.25	1–34	108 ***
Amarnath	IND	69	4378	42.50	138	11	32	55.69	2.91	4–43	47 ***
BSBedi	IND	67	656	8.99	50	0	266	28.71	2.14	10–194	26 ***
Kapildev	IND	131	5248	31.05	163	8	434	29.65	2.78	11–146	64 ***

```
    ABorder is the highest scorer with          11174 runs
    Kapildev is the highest wicket taker with     434 wickets
    MTaylor is the highest catch taker with       157 catches
```

Example 2

In this example we use the built-in variable FILENAME and also match a few patterns.

```
In this example we use FILENAME built in variable and print data from
first three lines of cricket.data file. In addition we print data from
ImranKhan to Allen Border

EGIN {FS = "\t"  # make the tab as the field separator
    printf("%25s \n","First three players")
R == 1, NR == 5 {print FILENAME ": " $0}
rintf("%12s %5s \n", $1,$2)}
mranKhan/, /ABorder/ {num = num + 1; line[num] = $0}
ND
rintf("Player list from ImranKhan to Allen Border \n",
printf("%12s %5s %7s %4s %6s %7s %4s %8s %8s %3s %7s %7s\n\n",
Name","Country","Matches","Runs","Batavg","Highest","100s","Wkts",
Bowlavg","Rpo","Best","Catches")}
    for(i=1; i <= num; i = i+1)
    print(line[i] ) }

hatt@falerno [AWK] =>!a
wk -f cprg.2 cricket.data
```

First three players
cricket.data SGavaskar IND
cricket.data MAmarnath IND
cricket.data BSBedi IND

Players list from ImranKhan to Allen Border

Name	Country	Matches	Runs	Batavg	Highest	100s	Wkts	Bowlavg	RPO	Best	Catches
ImranKhan	PAK	88	3807	37.69	136	6	362	22.81	2.55	14–116	28
MarkTaylor	AUS	104	7525	43.5	334	19	1	26.0	3.71	1–11	157
DLillie	AUS	70	905	13.71	73	0	355	23.92	2.76	11–123	23
DBradman	AUS	52	6996	99.94	334	29	2	36.0	2.7	1–15	32
ABorder	AUS	156	11174	50.56	265	27	39	39.10	2.28	11–96	156

At this time it may be worth our while to look at the list of some of the the built-in variables that are available in AWK (see Table 14.2).

Table 14.2 Built-in variables in AWK

Var name	*Meaning*	*Default*
ARGC	Number of command line arguments	–
ARGV	Array of command line arguments	–
FILENAME	Name of current file name	–
FNR	Record number in current file	–
FS	Control the input field separator	""
NF	Number of fields in current record	–
NR	Number of records read so far	–
OFMT	Output format for numbers	"%6g"
OFS	Output field separator	"\n"
RLENGTH	Length of string matched by match function	–
RS	Controls the input record separator	"\n"
RSTART	Start of string matched by match function	–
SUBSEP	Subscript separator	"\034"

14.3.2 More on AWK Grammar

Continuing our discussion with some aspects of the AWK grammar, we shall describe the nature of statements and expressions permissible within AWK. The statements are essentially invoke actions. In this description, "expression" may be with constants, variables, assignments, or function calls. Essentially, these statements are program actions as described below:

1. print expression-list
2. printf(format, expression-list)
3. if (expression) statement
4. while (expression) statement

6. for (variable in array) expression note: "in" is a ke
7. do statement while (expression)
8. break: immediately leave the innermost enclosing w
9. continue: start next iteration of the innermost encl
10. next: start the next iteration of the main input loo
11. exit: go immediately to the END action
12. exit expression: same with return expression as sta
13. {statements}

; : is an empty statement

We should indicate the following here:

1. Primary Expressions are: numeric and string function calls, and array elements
2. The following operators help to compose express
   (a) assignment operators =, +=, –=, *=, /=,
   (b) conditional operator ?:
   (c) logical operators || && !
   (d) matching operators =~ and !~
   (e) relational operators <, <=, ==, !=, >=
   (f) concatenation operator (see the string oper
   (g) arithmetic +, –, *, /, % ^ unary + and –
   (h) incrementing and decrementing operators post-fix)
   (i) parentheses and grouping as usual.

For constructing the expressions the following built-in fun

atan2(y,x)	arctan of y/x in the range of –pi to +pi
cos(x)	cosine of x; x in radians
exp(x)	
int(x)	truncated towards 0 when x > 0
log(x)	natural (base e) log of x
rand(x)	random no. r, where 0<= r < 1
sin(x)	sine of x; x in radians
sqrt(x)	square root of x
srand(x)	x is the new seed for rand()

To compute base 10 log use log(x)/log(10)
randint = int(n* rand()) + 1 sets randint to a random no. between 1 and n inclusive

Example 3

We next look at string operations. Let us first construct a concatenation operation is rather implicit. String expres constants, vars, fields, array elements, function values, ar other. The program {print NR ": " $0} concatenates as expecte In the example below, we shall use some of these facilities

Runs	Batavg	Highest	100s	Wkts	Bowlavg	RPO	Best	Catches
3807	37.69	136	6	362	22.81	2.55	14–116	28
7525	43.5	334	19	1	26.0	3.71	1–11	157
905	13.71	73	0	355	23.92	2.76	11–123	29
6996	99.94	334	29	2	36.0	2.7	1–15	32
11174	50.56	265	27	39	39.10	2.28	11–96	166

worth our while to look at the list of some of the the built
ble in AWK (see Table 14.2).

ble 14.2 Built-in variables in AWK

	Meaning	*Default*
Number of command line arguments		–
rray of command line arguments		–
ame of current file name		–
ecord number in current file		–
ontrol the input field separator		" "
umber of fields in current record		–
umber of records read so far		–
utput format for numbers		"%6g"
utput field separator		"\n"
ngth of string matched by match function		–
ntrols the input record separator		"\n"
art of string matched by match function		–
bscript separator		"\034"

rammar

n some aspects of the AWK grammar, we shall describe the
pressions permissible within AWK. The statements are
this description, "expression" may be with constants
nction calls. Essentially, these statements are program

n-list)
t
nent
on; expression) statement

10. next: start the next iteration of the main input loop
11. exit: go immediately to the END action
12. exit expression: same with return expression as status of program
13. {statements}

; : is an empty statement

We should indicate the following here:

1. Primary Expressions are: numeric and string constants, variables, fields, function calls, and array elements
2. The following operators help to compose expressions:
   (a) assignment operators =, +=, −=, *=, /=, % = $ =, ^
   (b) conditional operator ?:
   (c) logical operators || && !
   (d) matching operators =~ and !~
   (e) relational operators <, <=, ==, !=, >=
   (f) concatenation operator (see the string operator below)
   (g) arithmetic +, −, *, /, % ^ unary + and −
   (h) incrementing and decrementing operators ++ and −− (pre- as well as post-fix)
   (i) parentheses and grouping as usual.

or constructing the expressions the following built-in functions are available:

atan2(y,x)	arctan of y/x in the range of –pi to +pi
cos(x)	cosine of x; x in radians
exp(x)	
int(x)	truncated towards 0 when x > 0
log(x)	natural (base e) log of x
rand(x)	random no. r, where 0<= r < 1
sin(x)	sine of x; x in radians
sqrt(x)	square root of x
srand(x)	x is the new seed for rand()

To compute base 10 log use log(x)/log(10)
randint = int(n* rand()) + 1 sets randint to a random no.
between 1 and n inclusive

xample 3

e next look at string operations. Let us first construct a small recogniser. The string oncatenation operation is rather implicit. String expressions are created by writing onstants, vars, fields, array elements, function values, and others placed next to each ther. The program {print NR ": " $0} concatenates as expected ": " to each line of output. the example below, we shall use some of these facilities which we have discussed.

```
# this program is a small illustration of building a recogniser
BEGIN {
      sign = "[+–]?"
      decimal = "[0–9]+[.]?[0–9]*"
      fraction = "[.][0–9]+"
      exponent ="([eE]" sign "[0–9]+)?"
      number ="^" sign "(" decimal "|" fraction ")" exponent "$"
      }
$0 ~ number {print}
```

Note that in this example if /pattern/ were to be used then meta-characters would be recognised with an escape sequence, i.e. using \. for . and so on. Run this program using gawk with the data given below:

```
1.2e5
129.0
abc
129.0
```

You should see the following as output:

```
bhatt@falerno [AWK] =>gawk –f flt_num_rec.awk flt_num.data
1.2e5
129.0
129.0
```

Example 4

Now we shall use some string functions that are available in AWK. We shall match partial strings and also substitute strings in output (like the substitute command in vi editor). AWK supports many string-oriented operations. These are listed in Table 14.3.

Table 14.3 Various string function in AWK

String function with its arguments	*The function operation*
gsub(r, s)	substitute s for r globally
gsub(r, s, t)	substitute s for r globally in string t, return number of substitutions made
index(s, t)	return first position of string t in s, return 0 if t is not present
length(s)	return number of characters in s
match(s, r)	test whether s contains substring matched by r; return index or 0; sets RSTART and RLENGTH
split(s, a)	split s into array a on FS, return number of fields
split(s, a, fs)	split s into array a on field separator fs, returns number of fields
sprintf(fmt, expr-list)	return expression list formatted according to string format
sub(r, s)	substitute s for the left most longest substring of $0 matched by r; return number of substitutions made
sub(r, s, t)	substitute s for left most longest substring of t matched by r; return number of substitutions made

Let us now suppose we have the following AWK program line:

```
x = sprintf("%10s %6d",$1, $2)
```

This program line will return x in the specified format behaviour of the program segment given below:

```
{x = index("ImranKhan", "Khan"); print "x is :", x}
bhatt@falerno [AWK] =>!a
awk –f dummy.rec
x is : 6
```

In the response above, note that the index on the string h Also, if we use gsub command, it will act like vi substitution c

```
{gsub( /KDev/, "kapildev" ); print}.
```

Let us examine the program segment below:

```
BEGIN {OFS = "\t" }
{$1 = substr($1, 1, 4); x = length($0); print $0, x}
{s = s substr($1, 1, 4) "   "}
END {print s}
```

Clearly, the output lines would be like the one shown below line of output.

```
SGav  IND   125   10122   51.12 236 34    1   206.00   3.25 1–34   108   54
```

We next indicate how we may count the number of centuries scc players from Pakistan.

```
/IND/ {century["India"] += $7}
/PAK/ {century["Pakistan"] += $7}
/AUS/ {catches["Australia"] += $12; k = k+1; Aus[k] = $0}
END {print "The Indians have scored    ", century["India"], "centuries"
     print "The Pakistanis have scored  ", century["Pakistan"], "centuries"
     print "The Australians have taken ", catches["Australia"], "catches"}
```

The response is:

```
The Indians have scored        53   centuries
The Pakistanis have scored     21   centuries
The Australians have taken     368  catches
```

Example 5

Now we shall demonstrate the use of Unix pipe within the AW obtains output and then pipes it to give us a sorted output.

```
# This program demonstrates the use of pipe.

BEGIN{FS = "\t"}
{wickets[$2] += $8}
```

[0-9]+)?
lecimal "|" fraction ")" exponent "$"

:ample if /pattern/ were to be used then meta-characters would b
pe sequence, i.e. using \. for . and so on. Run this program usin
n below:

utput:
_num_rec.awk flt_num.data

ring functions that are available in AWK. We shall match partia
:e strings in output (like the substitute command in vi editor)
ng-oriented operations. These are listed in Table 14.3.

le 14.3 Various string function in AWK

The function operation
: s for r globally
: s for r globally in string t, return number of substitutions made
st position of string t in s, return 0 if t is not present
mber of characters in s
ıer s contains substring matched by r;
lex or 0; sets RSTART and RLENGTH
) array a on FS, return number of fields
) array a on field separator fs, returns number of fields
ression list formatted according to string format
s for the left most longest substring of $0
y r; return number of substitutions made
s for left most longest substring of t
r r; return number of substitutions made
ix of s starting at position p
string of s of length n starting at position p

behaviour of the program segment given below:

```
x = index("ImranKhan", "Khan"); print "x is :", x}
bhatt@falerno [AWK] =>!a
awk –f dummy.rec
x is : 6
```

In the response above, note that the index on the string begins with 1.
Also, if we use gsub command, it will act like vi substitution command as shown below:

```
gsub(/KDev/, "kapildev" ); print}.
```

Let us examine the program segment below:

```
BEGIN {OFS = "\t" }
{$1 = substr($1, 1, 4); x = length($0); print $0, x}
{s = s substr($1, 1, 4) "  "}
END {print s}
```

Clearly, the output lines would be like the one shown below. We have shown only one line of output.

```
SGav  IND  125  10122  51.12 236 34   1  206.00  3.25 1–34  108  54
```

We next indicate how we may count the number of centuries scored by Indian players and players from Pakistan.

```
/IND/ {century["India"] += $7}
/PAK/ {century["Pakistan"] += $7}
/AUS/ {catches["Australia"] += $12; k = k+1; Aus[k] = $0}
END {print "The Indians have scored   ", century["India"], "centuries"
     print "The Pakistanis have scored  ", century["Pakistan"], "centuries"
     print "The Australians have taken ", catches["Australia"], "catches"}
```

The response is:

```
The Indians have scored        53  centuries
The Pakistanis have scored     21  centuries
The Australians have taken     368 catches
```

Example 5

Now we shall demonstrate the use of Unix pipe within the AWK program. This program obtains output and then pipes it to give us a sorted output.

```
# This program demonstrates the use of pipe.
BEGIN{FS = "\t"}
{wickets[$2] += $8}
```

```
END {for (c in wickets)
     printf("%10s\t%5d\t%10s\n", c, wickets[c], "wickets") | "sort -t'\t' +1rn"}

bhatt@falerno [AWK] =>awk –f recog5.awk cricket.data
        IND      733      wickets
         NZ      445      wickets
        AUS      397      wickets
        PAK      367      wickets
         WI      291      wickets
        ENG        7      wickets
```

Suppose we have a program line as follows:

```
egrep "BRAZIL" cricket.data | awk 'program'
```

The obvious response is reproduced below:[4]

Normally, a file or a pipe is created and opened only during the run of a program. If the file, or pipe, is explicitly closed and then reused, it will be reopened. The statement close(expression) closes a file, or pipe, denoted by expression. The string value of expression must be the same as the string used to create the file, or pipe, in the first place. Close is essential if we write and read on a file, or pipe, alter in the same program. There is always a system defined limit on the number of pipes, or files that a program may open.

One good use of pipes is in organising input. There are several ways of providing the input data with the most common arrangement being:

```
awk 'program' data
```

AWK reads standard input if no file names are given; thus a second common arrangement is to have another program pipe its output into AWK. For example, egrep selects input lines containing a specified regular expression, but does this much faster than AWK does. So, we can type in a command egrep 'IND' countries.data | awk 'program' to get the desired input

```
bhatt@falerno [AWK] =>egrep 'BRAZIL' cricketer.data | awk –f recog3.awk
egrep: NO such file or directory
```

Example 6

Now we shall show the use of command line arguments. An AWK command line may have any of the several forms given below:

```
awk 'program' f1 f2 ...
awk –f programfile f1 f2 ...
awk –Fsep 'program' f1 f2 ...
awk –Fsep programfile f1 f2 ...
```

If a file name has the form var=text (note no spaces), however, it is treated as an assignment of text to var, performed at the time when that argument would be otherwise a file. This type of assignment allows vars to be changed before and after a file is read.

The command line arguments are available to AWK program in a built-in array called ARGV. The value of ARGC is one more than the number of arguments. With the command line awk –f progfile a v=1 b=i, ARGC is 4 and the array ARGV has the following values: ARGV[0] is awk, ARGV[1] is a, ARGV[2] is v=1 and finally, ARGV[3] is b. AR(number of arguments as awk is counted as the zeroth argumer

Here is another sample program with its response shown:

```
#echo – print command line arguments
BEGIN{ for( i = 1; i < ARGC; i++ )
       print("%s , \n", ARGV[i] )}

outputs

bhatt@falerno [AWK] =>!g
gawk –f cmd_line1.awk cricket.data
cricket.data
```

Example 7

Our final example shows the use of shell scripts.[5] Suppose program in file sh1.awk. We shall have to proceed as follows:

Step 1. Make the file sh1.awk as gawk '{print $1}' $*.

Step 2. Change the mode of file sh1.awk to make it executable

```
bhatt@falerno [AWK] => chmod +x sh1.awk
```

Step 3. Now execute it under the shell command

```
bhatt@falerno [AWK] => sh sh1.awk cricket.data file1.data file2.data
```

Step 4. See the result.

```
bhatt@falerno [AWK] =>sh sh1.awk cricket.data
SGavaskar
.....
.....
MartinCrowe
RHadlee
```

Here is an interesting program that swaps the fields:

```
#field swap bring field 4 to 2; 2 to 3 and 3 to 4
#usage : sh2.awk 1 4 2 3 cricket.data to get the effect
gawk '
BEGIN { for (i = 1; ARGV[i] ~ /^[0–9]+$/; i++) {# collect numbers
fld[++nf] = ARGV[i]
#print " the arg is :", fld[nf]
ARGV[i] = "" }
#print "exited the loop with the value of i : ", i
if (i >= ARGC) #no file names so force stdin
ARGV[ARGC++] = "–"
}
```

[5]Shell scripts are covered subsequently. For completeness of AWK pr(

wickets
wickets
wickets
wickets
wickets

line as follows:

ogram'

roduced below:[4]

created and opened only during the run of a program. If the
osed and then reused, it will be reopened. The statement
or pipe, denoted by expression. The string value of expression
ing used to create the file, or pipe, in the first place. Close
read on a file, or pipe, alter in the same program. There is
t on the number of pipes, or files that a program may open
in organising input. There are several ways of providing the
mmon arrangement being:

no file names are given; thus a second common arrangement
pipe its output into AWK. For example, egrep selects input
egular expression, but does this much faster than AWK does.
d egrep 'IND' countries.data | awk 'program' to get the desired input

cricketer.data | awk –f recog3.awk

command line arguments. An AWK command line may have
en below:

=text (note no spaces), however, it is treated as an assignment
e time when that argument would be otherwise a file. This
rs to be changed before and after a file is read.
ents are available to AWK program in a built-in array called
e more than the number of arguments. With the command
GC is 4 and the array ARGV has the following values: ARGV[0]

—

```
echo – print command line arguments
BEGIN{ for( i = 1; i < ARGC; i++ )
    print("%s , \n", ARGV[i] )}
```

utputs

```
bhatt@falerno [AWK] =>!g
awk –f cmd_line1.awk cricket.data
cricket.data
```

Example 7

Our final example shows the use of shell scripts.[5] Suppose we wish to have a shell program in file sh1.awk. We shall have to proceed as follows:

Step 1. Make the file sh1.awk as gawk '{print $1}' $*.

Step 2. Change the mode of file sh1.awk to make it executable.

```
bhatt@falerno [AWK] => chmod +x sh1.awk
```

Step 3. Now execute it under the shell command

```
bhatt@falerno [AWK] => sh sh1.awk cricket.data file1.data file2.data
```

Step 4. See the result.

```
bhatt@falerno [AWK] =>sh sh1.awk cricket.data
SGavaskar
.....
.....
MartinCrowe
RHadlee
```

Here is an interesting program that swaps the fields:

```
#field swap bring field 4 to 2; 2 to 3 and 3 to 4
#usage : sh2.awk 1 4 2 3 cricket.data to get the effect
gawk '
BEGIN { for (i = 1; ARGV[i] ~ /^[0–9]+$/; i++) {# collect numbers
fld[++nf] = ARGV[i]
#print " the arg is :", fld[nf]
ARGV[i] = "" }
#print "exited the loop with the value of i : ", i
if (i >= ARGC) #no file names so force stdin
ARGV[ARGC++] = "–"
}
```

Shell scripts are covered subsequently. For completeness of AWK programming, we have included his example here.

```
# {print "testing if here "}
{for (i = 1; i <= nf; i++)
    #print
    printf("%8s", $fld[i])
   }
    {print '' ''}' $*
bhatt@falerno [AWK] =>!s
sh sh2.awk 1 2 12 3 4 5 6 7 8 9 10 11 cricket.data
SGavaskar  IND  108  125 10122  51.12  236  34   1  206.00   3.25   1–34
.....
.....
ABorder    AUS  156  156 11174  50.56  265  27  39   39.10   2.28  11–96
```

In the examples above, we have described a very powerful tool. It is hoped that with these examples the reader will feel comfortable with the Unix tools suite.

EXERCISES

1. How can we get not only a line containing AAA but the previous line as well?
2. How can we make awk to be case-insensitive?
3. How can we force a numeric/non-numeric comparison?
4. How can we control formatting the output (the dynamic-width control using *printf* strings), like in the *c* programming language?
5. Why does FS =":"; print $1 not split the first record?
6. How does one replace "string1" with "string2" for all the files in ./directory?
7. We are trying to sort a file on the left of the "*" mark to yield a file on the right of the "*" mark.

dd	1	3	*	dd	1	3
mff	4	4	*	lgse	3	4
lgse	3	4	*	mff	4	4
zsw	6	6	*	dhd	4	5
dhd	4	5	*	zsw	6	6
a	8	8	*	a	8	8

   Basically the numbers in the second and third columns are added to get the sorted order.
8. Compare and contrast awk with perl.

15 Shell Scripts

A shell, as we remarked in Chapter 1, offers a user an interfa user obtains OS services through an OS shell. When a user log The login shell offers the first interface that the user interacts console, or through a window interface which emulates a ter command interface to use an OS, and the login shell immediat as the user logs in). Come to think of it, a shell is essentia command interpreter!! In this chapter we shall explore ways enhancement in a user's productivity.

15.1 FACILITIES OFFERED BY UNIX SHELLS

In Figure 15.1, we show how a user interacts with any Unix distinguishes between the commands and a request to use a too operational environment. In that case the shell hands in environment. As an example, if a user wishes to use the editor t it has its own states like edit and text mode, etc. In case we l then it has a well-understood interpretation across all the shells is particular to a specific shell, then it needs interpretation in shell by interpreting its semantics.

Every Unix shell offers two key facilities. In addition to interpreter, it also offers a very useful programming environmen environment accepts programs written in shell programming l in this chapter we shall learn about the shell programming la use of it for enhancing productivity. A user obtains maximum only to automate a sequence of commands but also to make sequences and invoke repeated executions.