AWK Tool in UNIX

Prof. P.C.P. Bhatt

Motivation for AWK

- Unix as we know provides tools. AWK is a tool that facilitates processing of structured data – more like what we see as a record structure in Pascal or a struct in C – essentially a data set with multiple fields in each element of the data.
- Incidently AWK as a name comes from the names of three persons Profs. Aho, Weinberger and Kernighan who were responsible for creating this tool
- Such a tool very useful for data-processing as in processing of records or even string processing like what we can now do with PERL.

The Basic Structure of AWK Program

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

Running AWK Programs

Use awk command to run an *awk* program

awk *'awk_program' [input_files]*

Using FileNme as an Argument

If the ‘awk_program’ is very long then it is better to keep the script on a file and then use the –f option as shown below:

awk –f *‘awk_program_file_name’[input_files]*

where awk_program_file_name contains the awk program

Test Data File

File name: awk.test

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

Which denotes each employee’s name and pay (rate per hour) and no. of hours worked.

Sample 1

```
bhatt@falerno[CRUD] => awk '$3 > 0 {print $1,$2*$3}' awk.test
```

```
ulhas   7.5
ritu    20
vivek   6
```

Example - 1

The employees who did not work :

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

```
bhatt
```

The basic operation is to scan a sequence of lines searching for the lines that match any of the patterns in the program $3 > 0 match when the condition is true.

Example - 2 (Using NF and NR)

NF is built in variable that stores the no. of fields.

{print NF, $1, $NF } Which prints no. of fields, first and last field.

NR, another built in variable is a no. of lines read so for and can used the print stmt.

```
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 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
person  5  vivek  be  paid  6    dollars
```

Example - 3 (Formatting the Output)

One can use printf to format the output like in C programs.

```
bhatt@falerno[CRUD] => awk '$3 > 0 {print ("%-8s be paid
$%6.2f dollars \n", $1, $2*$3, "dollars"}' awk.test
```

```
ulhas      be paid     $   7.50   dollars
ritu       be paid     $   20.5   dollars
vivek     be paid     $    6.00  dollars
```

Example - 4 (Data Validation)

Awk is excellent for data validation

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 hours worked –ve etc

Example - 5 (Putting Headers and Footers)

Tabulation can be done by using

BEGIN { print “Name Rate Hours “ }

```
bhatt@falerno[CRUD] => awk ‘BEGIN { print  “Name Rate Hours” ; print””} { print }’ awk.test
```

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

Example - 5 (Putting Headers and Footers)

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
```

User Defined Variables in AWK

Now we shall attempt some computing within awk. To perform computations we may sometimes need to employ user-defined variables. In this example “pay”shall be used as 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.

Example - 6 (User Defined Functions)

File name: prg2.awk

```
BEGIN { print "NAME     RATE
HOURS";print "" }
{pay = pay+$2*$3}
END {print NR "emplayees"
        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
```

Example – 7 (Built in Functions)

There are some built in functions that can be useful. For instance , length function helps one to compute the length of a field.

prg4.awk

```
{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 awk.test
                53 characters and
                12 words and
                4   lines in this file
```

Controlling The Sequence of Operations in AWK

Control Flow Statements :

if-else

while loop

for loop

We shall explore the examples for each statement.

Using if- else Statement - 1

```
Prg5.awk
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 }
End{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 employees in this bracket of salary . “
print "highest salary paid rate is for "maxemp ,"@
of:",maxrate
print emplist
print “”}
```

Using if- else Statement - 2

The result is shown below:

```
bhatt@falerno[CRUD] => !a awk -f prg5.awk data.awk
```

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

Using While Loop - 1

In this example, we simply compute the compound interest that accrues each year for a five year period.

```
#compound : interest computation .
#input   : amount rate yrs.
#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
 }
}
```

Using While Loop - 2

The result is shown below:

```
bhatt@falerno[CRUD] => !a awk -f prg6.awk data.awk

1000 0.06 5
        1       1060.00
        2       1123.60
        3       1191.02
        4       1262.48
        5       1338.23
```

Using “for” Statement - 1

```
# 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]
}
```

Using “for” Statement - 2

The result is shown below:

```
bhatt@falerno[CRUD] => !a awk -f prg6.awk data.awk

NAME         RATE       HOURS

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

AWK One Liners - 1

- Print the total no. of input lines : END { print NR}
- Print the 10th input line : NR = 10
- Print the last field of each line : {print $NF}
- Print the last field of last line : { field = $NF}END { print field}
- Print every input line with more than 4 fields : NF >4
- Print every input line i which the last field is more than 4 :

  $NF > 4

AWK One Liners - 2

- Print total number of fields in all input lines

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

- Print the total no. of lines containing bhatt

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

- Print the largest first field and the line that contains it : $1 > max {max = $1;maxline = $0}

```
END { print max, maxline }
```

AWK One Liners - 3

- Print every line that has at least one field: NF>0
- Print every line with > 80 chs: length($0) > 80
- Print the no. of fields followed by the line it self

  {print NF, $0}
- Print the first two fields in opposite order

  {print $2, $1}
- Exchange the first two fields of every line and then and then print the line

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

AWK One Liners - 5

- Print every line with first field replaced by line no. :

  {$1 = NR; print }

- Print every line after erasing second field :

  { $2 = “ “ ; print }

- Print in reverse order the fields of every line

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

AWK One Liners - 6

- Print the sums of fields of every line :

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

- Add up all the fields in all lines and print the sum

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

- 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}
```

AWK Grammar - 1

1. BEGIN*{statements}:* These statements are executed once before any input is processed.
2. END*{statements}:* These statements are executed once 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 regular expression.

AWK Grammar - 2

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.the expr 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 action happen in the order of their appearance.

AWK Grammar - 3

8. A range pattern cannot be part of any other pattern.
9. “FS" is a built-in variable for field separator.

AWK Grammar - 4

String Matching Patterns :

1. /regexpr/ matches an input line if the line contains the specified substring. As an example : /India/ matches “India " (with space on both the sides), just as it detects presence of India in “Indian".

2. expr ~ /regexpr/ matches, if the value of the expr contains a substring matched by 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".

Regular Expressions - 1

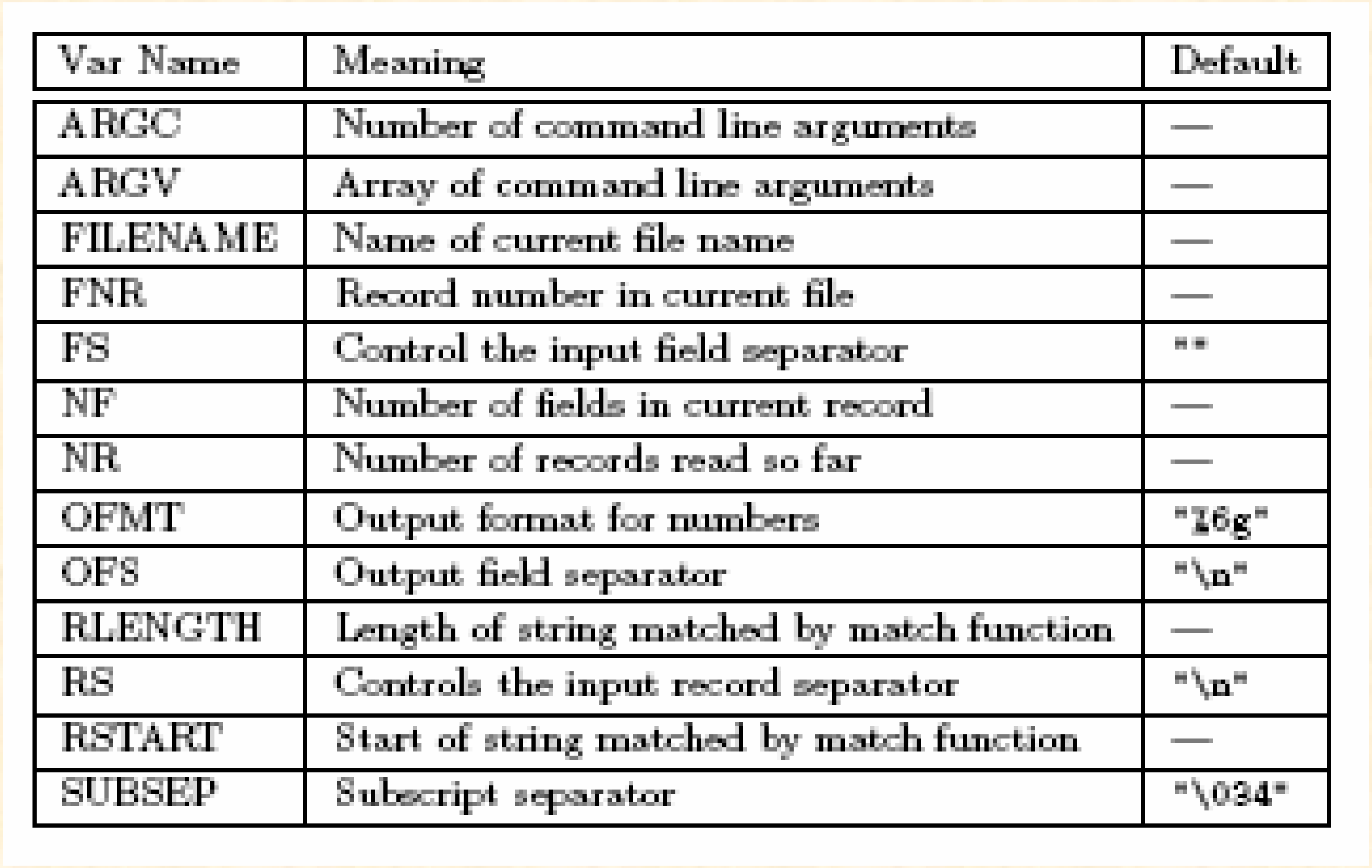
The following is the summary of the Regular Expression matching rules.

^C	: matches a C at the beginning of a string
C$	: matches a 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

Regular Expressions - 2

...	: 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

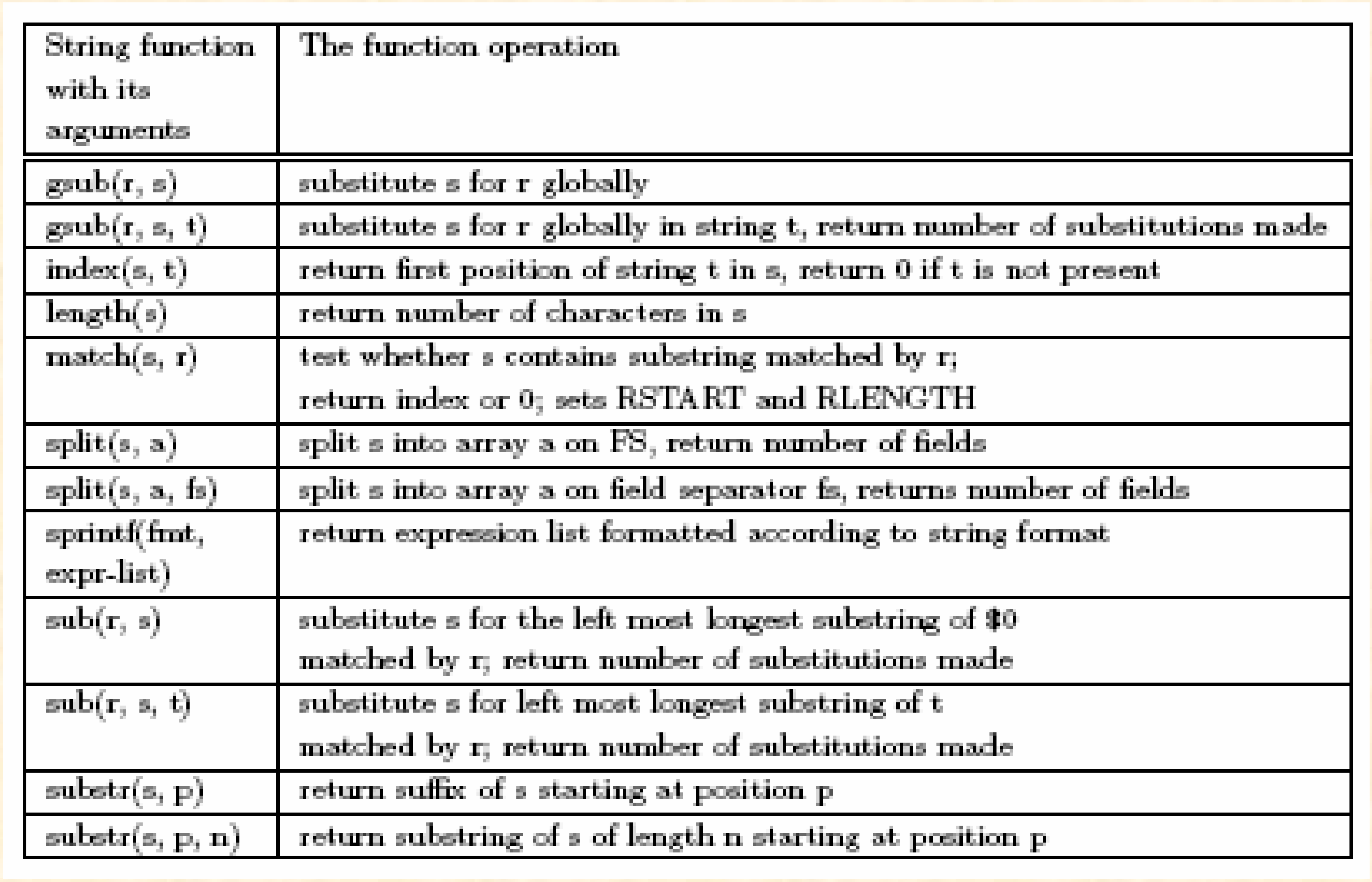
Built-in Variables



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"

Built-in Variables in AWK.

String Functions



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
substr(s, p)	return suffix of s starting at position p
substr(s, p, n)	return substring of s of length n starting at position p

Various String Function in AWK.