Unix Programming

UNIT-IV

B.Tech(CSE)-V SEM

UNIT-IV : Regular Expressions

- **grep**,egrep, fgrep
- **Sed**- line addressing, context addressing, text editing, substitution.
- **Programming with awk**: syntax of awk programming statement, structure of awk script, variables ,records fields, and special variables, patterns, operators ,simple input files, awk programming- simple awk programming, awk control structures, looping, functions in awk.

grep

- grep is an acronym for “globally search a regular expression and print it”

- The grep filter searches for a particular pattern of characters and displays all the lines that contain the pattern. The pattern that is searched for is referred to as a regular expression. The grep filter can not be used without specifying a regular expression.

syntax:

$grep [options] regular-expr [filename] (or)

$grep regular-expr [filename] (or)

$grep regular-expr

grep (Contd..)

Example:

```
$grep  job
work
job
job
task
[ctrl+d]
$
```

In this example grep searches for the word job, when user types in the words work, job and task. When it finds the word job it displays the word job again on the standard output.

grep (Contd..)

Example	Description
$grep “rao” xyz	This displays those lines of the file xyz having string “rao”
$grep “[rR]ao” xyz	This displays those lines of the file xyz having string either “rao” (or) “Rao”
$grep “^rao” xyz	This displays those lines of the file xyz , which starting with string “rao”
$grep “rao$” xyz	This displays those lines of the file xyz , which ending with “rao”
$grep “^rao$” xyz	This displays those lines of the file xyz , which contains the string “rao” only. No more characters in the line.
$grep “^$” xyz	This displays empty lines of the file xyz.
$grep “^[rR]ao” xyz	This displays those lines of the file xyz having (starts) with either “rao” (or) “Rao”

grep (Contd..)

Options:

option	Description
-n	This prints each line of matching pattern, along with its line number. The number is printed at the beginning of the file.
-c	This prints only a count of the lines that match a pattern.
-v	This prints all the lines that do not match the pattern specified by the Regular Expression.

NOTE: options must be used / specified before the Regular Expression. Options can also be combined. For example, -n and –v can be used together as -nv .

Example: Searching for a string in Multiple files.

$ grep "rao" file *.*

The above command will search for "rao" string in multiple files at atime. It searches in all files with file1.txt, file2.txt and along with different extensions too like file1.c, file2.sh and so on.

fgrep (fixed grep)

- fgrep is used for a group of strings. One string has to be separated from other by a new line.

Syntax:

```
$fgrep   'regularexpr1
> regularexpr2
.....
> regularexprn'   filename
```

Example:

```
$fgrep  'rao
>ram
>raju'   sample
```

This command displays those lines having either rao/ram/raju. fgrep will not accept regular expression.

egrep (extended grep)

- egrep stands for extended grep. This is so because it has two additional metacharacters.
- The two additional metacharacters are the plus (+) character and the question mark (?) character.
- This command is the most powerful member of the grep command family.
- The foremost advantage of this command is that multiple search patterns can be handled very easily. The pipe (|) character is used to mention alternate patterns.

egrep (extended grep) Contd..

Syntax:

$grep -E Regularexpr filename

(or)

$egrep Regularexpr filename

+ -> matches one or more occurrences of the previous character.

? -> matches zero or one occurrences of the previous character.

For example: b+ matches b,bb,bbb etc, but it does not match nothing – unlike b*. The expression b? matches either a single instance of b (or) nothing.

egrep (extended grep) Contd..

Example 1:

$egrep "true?man" Hai

trueman

truman

Example 2: Matching Multiple Patterns (|, (and))

a) $egrep 'woodhouse | woodcock' hai

woodcock

woodhouse

b) $egrep 'wood(house | cock)' hai

woodcock

woodhouse

egrep (extended grep) Contd..

Expression	Significance
ch+	Matches one or more occurrences of character ch
ch?	Matches zero or one occurrences of character ch
expr1 | expr2	Matches expr1 or expr2
(x1|x2)x3	Matches x1x3 (or)x2x3

Examples:

Expression	Description
g+	At least one g
g?	Nothing (or) one g
GIF / JPEG	Matches expr1 or expr2GIF (or) JPEG
(lock/ver) wood	Matches lockwood (or) verwood

Sed: The stream Editor

- Sed is a multipurpose tool which combines the work of several filters. It is derived from ed, the original unix editor. Sed performs non-interactive operations on a data stream-hence its name.

- Sed function performs lot of functions on files like, searching, find , replace, insertion or deletion.

- Most common use of sed command in unix is for substitution (or) find and replace.

- By using sed we can edit files even without opening it, which is much quicker way to find and replace something in file, than first opening that file in vi editor and then changing it.

Sed: The stream Editor (Contd…)

- sed is a powerful text stream editor, can do insertion, deletion, search and replace (substitution).

- sed command in unix supports regular expression which allows it perform complex pattern matching.

Syntax:

$sed options 'address action' file(s)

- Address specifies either one line number to select a single line or a set of two lines, to select a group of contiguous lines.

- Action specifies print, insert, delete, substitute the text.

Sed: The stream Editor (Contd…)

- sed processes several instructions in a sequential manner.
- Each instruction operates on the output of the previous instruction.
- In this context, two options are relevant, and probably they are the only ones we will use with sed –
  - the –e option that lets us use multiple instructions, and
  - the –f option to take instructions from a file.
- Both options are used by grep in identical manner.

sed: The stream Editor (Contd...)

Command	Description
a\	Append lines to output until one not ending in \.
c\	Change lines to following text as in a
d	Delete line; read next input line.
i\	Insert following text before next output
p	Prints line(s) on standard output
Q	quit
r file	read file, copy contents to output
w file	write line to file
3, $p	Prints lines 3 to end (-n option required)
$!p	Prints all lines except last line (-n option required)
s/s1/s2	Replaces first occurrence of expression s1 in all lines with expression s2
10,20s/-/:/	Replaces first occurrence of - in lines 10 to 20 with a :
s/s1/s2/g	Replaces all occurrence of expression s1 in all lines with expression s2
s/-/:/g	Replaces all occurrence of - in all lines with a :

sed: The stream Editor (Contd…)

Addresses: Addresses may be either line addresses or context addresses.

(1) **Line number addresses**: A line address is a decimal integer. As each line is read from the input, a line number counter is incremented. A line address selects a line when the line number is equal to line number counter. The counter runs cumulatively through multiple input files; it is not reset when a new input file is opened. As a special case, the character $ matches the last input line.

(2) **Context Addresses:** A context address is a regular expression enclosed in slashes('/'). Each line that matches the regular expression is operated on.

Sed: Line Editor (Contd..)

Examples:

Command 1: Display line multiple times

```
sed     '2p'     file.txt
sed     -n   '3p'     file.txt   #specific line => -n
sed     -n    '5p‘    file.txt
```

Command 2: Display last line[$]

```
sed    '$p‘     file.txt          #includes last line again along with original
sed    -n ‘   $p'    file.txt     #Specific
```

Sed: Line Editor (Contd..)

Examples:

Command 3: Range of lines

```
$sed –n ‘9,11p’ sample      #selects lines from anywhere of the file, between lines from 9 to 11.

$sed –n ‘1,2 p
7,9 p
$p’ sample
```

Command 4: Do not display specific lines

```
$sed -n '2!p‘ file.txt

$sed -n '2,4!p' file.txt      #do not display specific range of lines(!)
```

Sed: Line Editor(Contd…)

Using Multiple Instructions (-e and –f):

(i) **–e :** This option allows you to enter as many instructions as you wish, each preceded by the option.

(ii) **-f:** This option is used to direct the sed to take its instructions from the file using the command.

Example:

$sed -n -e '1,2p' -e '7,9' -e '$p' sample

sed- Context Addressing

Command 1: Display lines having a specific word

```
sed -n '/Amit/p' file.txt
sed -n '/[Aa]mit/p' file.txt        (Ignoring Case)
```

Command 2: Search lines and store in the file

```
sed -n   '/[Aa]/p'   file.txt
sed –n   '/[Aa]/w result.txt'  file.txt   (Write to result.txt)
```

Command 3: Replace context from the file

```
sed    's/10000/1500/'    file.txt
sed     's/hello/hi/'      file.txt
```

sed- Context Addressing

- **Command 4: Replace multiple context in one command**

  sed -e 's/100/150/' -e 's/200/250/' file.txt

- **Command 5: Replace data by matching some condition**

  sed '/Sai/s/100/150/' file.txt

  sed '/Msd/s/100/150/' file.txt **[Check the match(Msd) and Substitute]**

- **Command 6: Delete data**

  sed '/sai/d' file.txt

  sed '/[Aa]mit/d' file.txt

sed- Text editing

- sed supports inserting (i), appending (a), changing (c) and deleting (d) commands for the text.

```
$ sed 'ii\
#include <stdio.h>\
#include <unistd.h>
'sample.c > $$
```

- The above script will add two include lines in the beginning of sample.c file.

- Sed identifies the line without the \ as the last line of input. Redirected to $$ temporary file.

sed- Text editing

- To insert a blank line *after* each line of the file is printed (*double spacing text*), we have,

```
sed 'a\
' emp.lst
```

- **Deleting lines (d)**

```
sed '/director/d' emp.lst > olist   or
sed –n '/director/!p' emp.lst > olist
```

- Selects all lines except those containing *director*, and saves them in *olist*
- *Note that –n option not to be used with d*

sed- substitution

- Substitution is the most important feature of sed, and this is one job that sed does exceedingly well.

[address]s/expression1/expression2/flags

- Just similar to the syntax of substitution in vi editor, we use it in sed also.

sed 's/|/:/' emp lst | head –n 2

2233:a.k.shukla |gm |sales |12/12/52|6000

9876:jai sharma |director|production|12/03/50|7000

- In the above output, only the first instance of | in a line has been replaced. We need to use the g (global) flag to replace all the pipes.

sed 's/|/:/g' emp.lst | head –n 2

sed- substitution

- We can limit the vertical boundaries too by specifying an address (for first three lines only).

  sed '1,3s/|/:/g' emp.lst

- Replace the word director with member in the first five lines of emp.lst

  sed '1,5s/director/member/' emp.lst

- sed also uses regular expressions for patterns to be substituted.
- To replace all occurrence of agarwal, aggarwal and agrawal with simply Agarwal, we have,

  sed 's/[Aa]gg*[ar][ar]wal/Agarwal/g' emp.lst

sed- substitution

- We can also use ^ and $ with the same meaning. To add 2 as prefix to all emp-ids,

sed 's/^/2/' emp.lst | head –n1

22233 | a.k.shukla | gm | sales | 12/12/52 | 6000

- To add .00 suffix to all salary,

sed 's/$/.00/' emp.lst | head –n1

2233 | a.k.shukla | gm | sales | 12/12/52 | 6000.00

The awk- Pattern Scanning and Processing Language:

- awk is a filter program that was originally developed in 1977 by Aho, Weinberger and Kernighan as a pattern-scanning Language.
- The name awk is derived from the first letters of it's developers names.
- It is a programming language with C-like control structures, functions and variables.
- It was designed to work with structured files and text patterns.
- One of the very important feature of awk filter is it operates at the field level.

The awk command: Features

- Field- Oriented file processing.
- Regular Expressions.
- Pre-defined Variables.
- Numeric Operations.
- Comparison operators.
- Arrays.
- Control statements.
- Report Generation.

Syntax of awk programming statement:

- The basic syntax of awk as a command is:

  $awk [options] 'program' filelist

- Where
  - Use of options is optional.
  - Filelist will have zero or more input filenames.
  - Program will have one or more statements with the following syntax:

    pattern {action}

- The pattern component of a program statement indicates the basis for a line or record selection and manipulation.

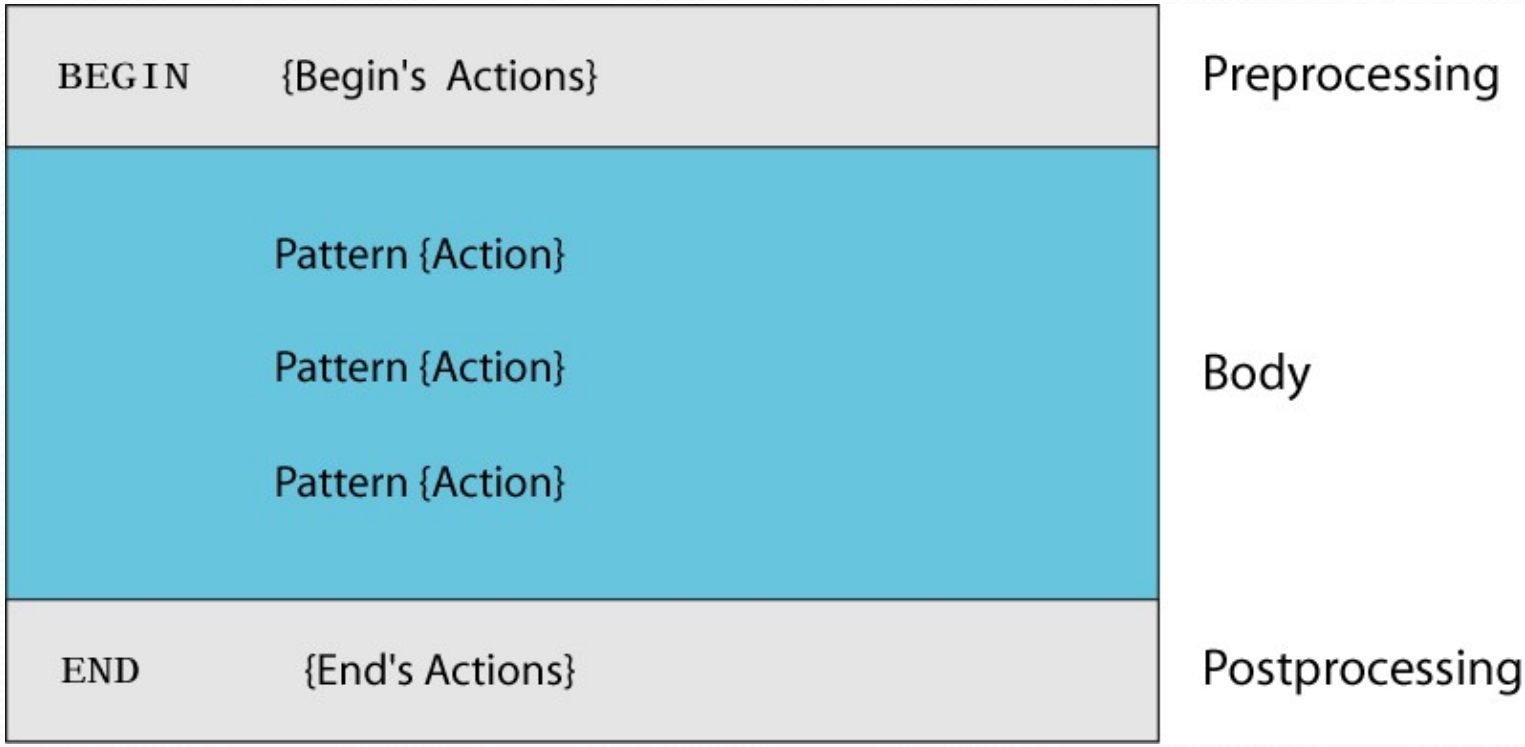
Syntax of awk programming statement:

- The action part of every program statement is surrounded by a pair of curly brackets.
- The action part is made up of C-like statements, which performs actions on the lines or records selected based upon the pattern component.
- The patterns can be simple words or regular expressions or they can be more complicated conditions.
- awk employs two options:

Option	Description
-F	Specifies the input field separator.
-f	Specifies that the program is on a separate file.

Structure of awk script:

- awk scripts are divided into three major parts:



- comment lines start with #

Structure of awk script:

- BEGIN: pre-processing
  - performs processing that must be completed before the file processing starts (i.e., before awk starts reading records from the input file).
  - useful for initialization tasks such as to initialize variables and to create report headings.
  - A typical BEGIN section statement that identifies the input field separator as the colon (:) character is:

    'BEGIN {FS ":" }'
- BODY: Processing
  - contains main processing logic to be applied to input records.
  - like a loop that processes input data one record at a time:
    - if a file contains 100 records, the body will be executed 100 times, one for each record.

Structure of awk script:

- END: post-processing
  - contains logic to be executed after all input data have been processed
  - logic such as printing report grand total should be performed in this part of the script
  - A typical END section statement that prints number of records processed is:

‘ END { print NR }’

Operational mechanism of awk:

- The working of awk is similar to sed:
  - awk picks up the records or lines from the input file one by one and applies all the program statements present on the program file to each line.
  - Applying all the program lines means pattern portion of the every program statement is compared with the presently picked up line one by one.
  - Whenever the pattern portion of the program statement matches, the action mentioned in the action portion of the matched program statement is carried out on the present input line.
  - The only difference between sed and awk is:
  - awk being a language, the pattern portion of awk statement might be made up of any of the operators, decision-making statements, looping control structures, regular expressions and so on.

Variables in awk:

- awk supports two types of variables:
  - User-defined variables.
  - Built-in Variables.
- **User-defined variables:**
  - These are names of storage locations that hold either strings or numbers.
  - These names are constructed using only alphanumeric and underscores. Such names must begin with a letter.
  - Similar to shell, it is not necessary to either initialize or type declare the variables in awk also.
  - The variables once defined get set to zero or a null string automatically. The type of variable is decided depending on the context.
  - In case of a tie between string and numeric operators, the context will be decided as the string.

Variables in awk:

- **Built-in Variables:**
  - Built-in variables are predefined. Names of these variables are constructed using uppercase letters.

Variable	Meaning
FILENAME	Name of the current input file.
FS	Input field separator (default: blank and tab)
NF	Number of fields in Input record.
NR	Number of current record.
OFS	Output field separator (default: blank or tab).
ORS	Output Record Separator (default: new line).
RS	Input Record Separator (default: new line).
ARGC	Number of command line arguments.
ARGV	Command line arguments array.

Records, fields and special variables:

- The awk treats every line of an input file as a record. This input file could be either a text file or a database file.
- Each unit or word of a record is known as a field. Thus a record is made up of many fields.
- The fields are separated by a blank or tab character by default.
- The default value of the field separator is available in a built-in variable FS. The default value of FS can be changed if required.
- Whenever awk picks up a line or a record for processing, it automatically splits every record into a number of fields.
- Contents of each of these fields are automatically saved in special variables $1, $2, $3 and so on.
- The information regarding the total number of fields in a record will be available in built-in variable NF.

The $0: Another special variable:

- The awk picks up one line or a record at a time for processing.
- The current line or record that is being processed will be available in a special variable called $0.

Patterns:

- awk allows the use of different types of patterns.
- As an awk script is executed, patterns are evaluated against each of the records or lines found in the input file.
- Whenever a pattern matches a record or line, the action mentioned in the action part of the awk program statement is taken.
- An awk program statement may not have a pattern at all and such cases are called no pattern case. In such case, action is taken on all the records or lines of the input file.
- The BEGIN and END are two special patterns.
- Whenever a pattern is present, it is made up of an expression. An expression may be an arithmetic expression, relational expression, logical expression, or a regular expression.
- Regular expressions of awk are similar to those of egrep. Thus patterns can be constructed by using any of the metacharacters used with both grep and egrep.

Patterns:

- Regular expressions are always written within a pair of forward slashes (/).
- The awk has an operator called the match operator represented by ~ and an operator called no-match operator represented by !~.
- In the case of match operator, the regular expression must match the text whereas in the case of no-match operator the regular expression must not match the text.

Operators:

- Similar to shell and other programming languages, we can use arithmetic, relational, logical and assignment operators in awk programming also.

Category	Operators	Description
Arithmetic Operators	+ -	Plus, Minus
	* / %	Multiply, divide, remainder
Logical Operators	||	Logical OR
	&&	Logical AND
	!	Negation or Complementation
Relational Operators	> >= < <= == !=	Relational operators
Assignment Operators	= += -= *= /= %=	Assignment and their short hand notations
Match operators	~ !~	These are match and no-match operators respectively
Increment and decrement operators	++ --	Increment, decrement (prefix or postfix)

Sample Input Files

The contents of two structured files named phone.lst and marks.pu have been used as input files.

```
sekhar@DESKTOP-UVKV03T:~$ cat phone.lst
mgv     murthy          267757
hs      prabhakara      245092
kak     murthy          268088
vn      narayana        251833
mgv     krishna         245020
sekhar@DESKTOP-UVKV03T:~$
```

```
sekhar@DESKTOP-UVKV03T:~$ cat marks.pu
Radhika         72      67      96
Darshana        86      97      93
Anil            88      96      91
Prasanna        75      86      79
Vinay           45      99      88
sekhar@DESKTOP-UVKV03T:~$
```

Simple awk programs: Example-1

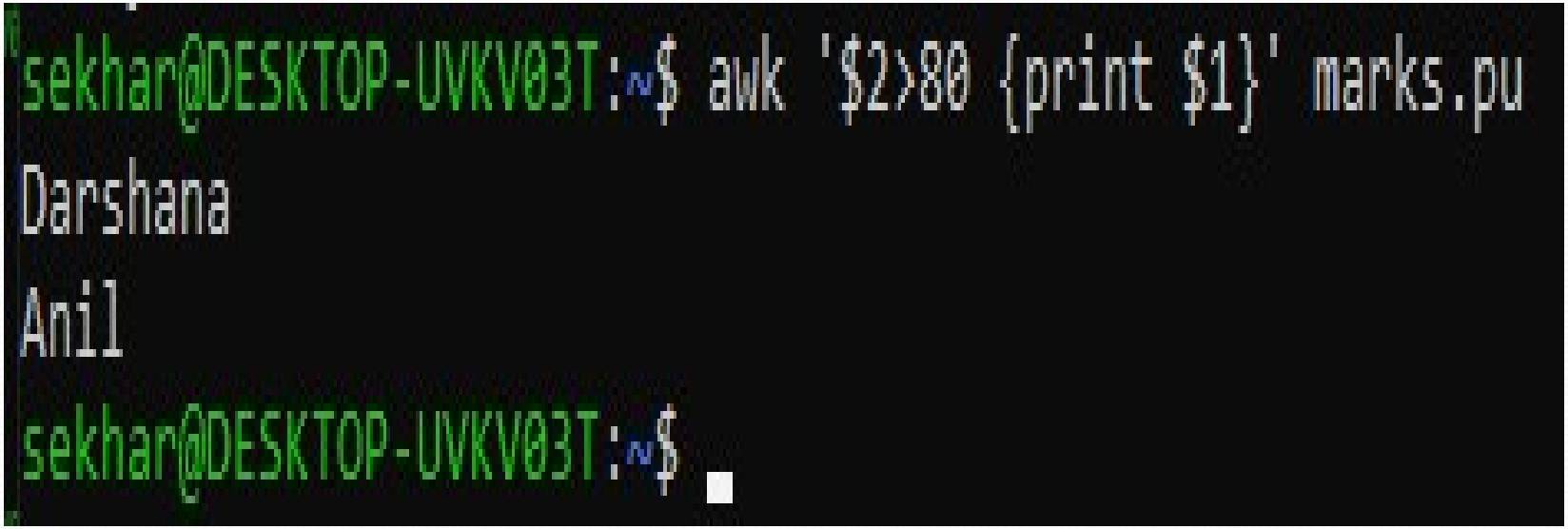
$awk '{print $1,$2}' marks.pu # displays field1 and field2 values of file marks.pu

```
sekhar@DESKTOP-UVKV03T:~$ awk '{print $1,$2}' marks.pu
Radhika 72
Darshana 86
Anil 88
Prasanna 75
Vinay 45
sekhar@DESKTOP-UVKV03T:~$
```

Simple awk programs: Example-2

$awk '$2>80 {print $1}' marks.pu

#displays the names of students whose marks in subject1 is >80



Simple awk programs: Example-3

$ awk '$2+$3+$4>=240 {print $1, $2+$3+$4}' marks.pu

```
sekhar@DESKTOP-UVKV03T:~$ awk '$2+$3+$4>=240 {print $1, $2+$3+$4}' marks.pu
Darshana 276
Anil 275
Prasanna 240
sekhar@DESKTOP-UVKV03T:~$
```

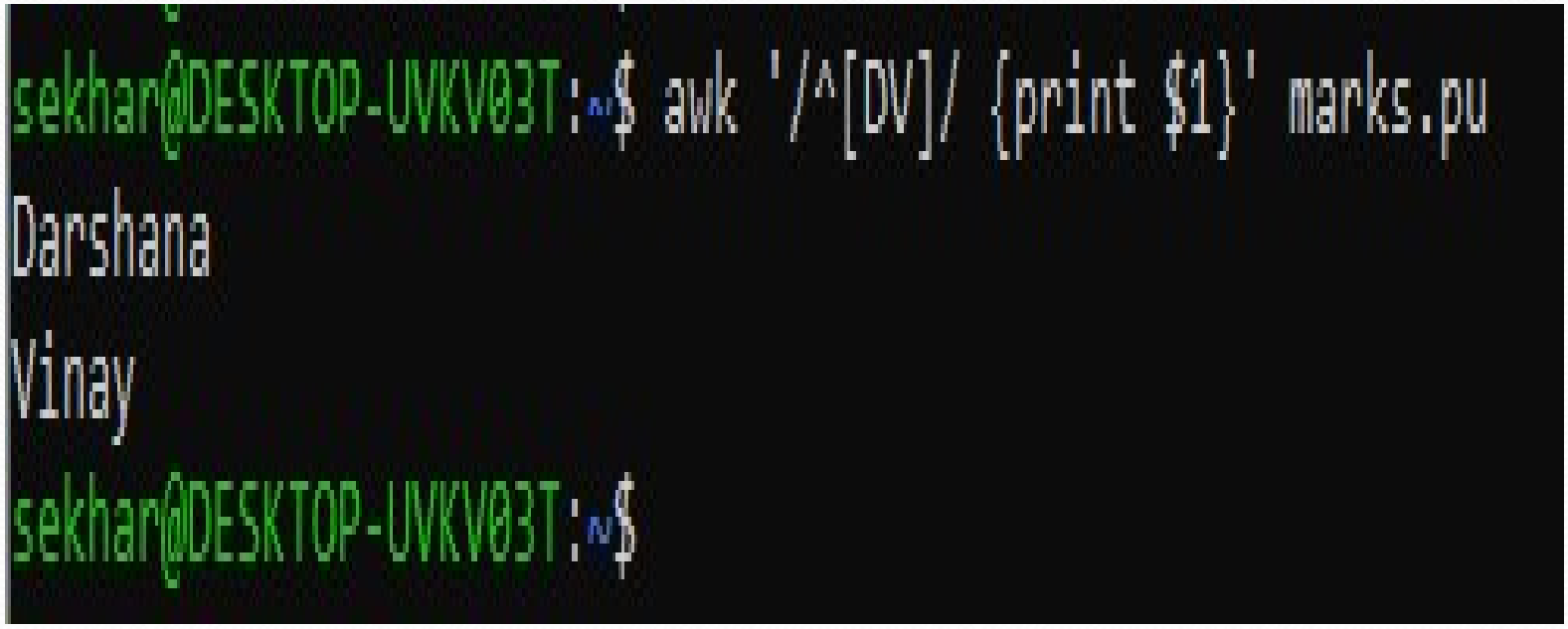
Simple awk programs: Example-4

$ awk '$2 >=60 && $2 <=80 {print $1,$2}' marks.pu

```
sekhar@DESKTOP-UVKV03T:~$ awk '$2 >=60 && $2 <=80 {print $1,$2}' marks.pu
Radhika 72
Prasanna 75
sekhar@DESKTOP-UVKV03T:~$
```

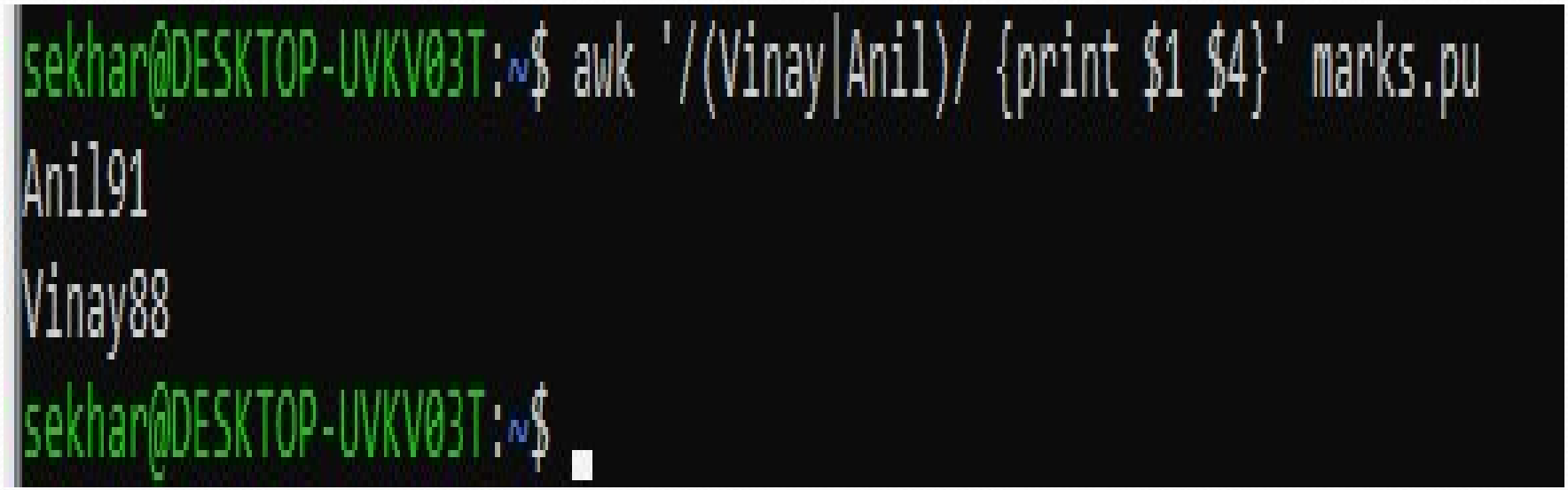
Simple awk programs: Example-5

$awk '/^[DV]/ {print $1}' marks.pu



Simple awk programs: Example-6

$awk '/(Vinay|Anil)/ {print $1 $4}' marks.pu



Simple awk programs: Example-7

$awk '/(Vinay|Anil)/ {printf $1 $4}' marks.pu

Simple awk programs: Example-8

$awk ‘NR ==2, NR==4 {printf(“%4s %-12s %7d\n”, $1,$2,$3)}’ phone.lst

```
sekhar@DESKTOP-UVKV03T:~$ awk 'NR==2, NR==4 {printf("%4s %-12s %7d \n",$1,$2,$3)}' phone.lst
  hs prabhakara    245092
 kak murthy        268888
  vn narayana      251833
sekhar@DESKTOP-UVKV03T:~$
```

Simple awk programs: Example-9

Being a filter program, the awk can take its input from the output of another program.

```
$date | awk '{print "The day is:", $1
              print "The month is:", $2
              print "The year is:", $6}'
```

```
sekhar@DESKTOP-UVKV03T:~$ date | awk '{print "The day is:", $1
> print "The month is:", $2
> print "The year is:", $6}'
The day is: Wed
The month is: Sep
The year is: 2020
sekhar@DESKTOP-UVKV03T:~$
```

Simple awk programs: Example-10

- An awk program is a filter that can take input piped or redirected to it from another file.
- Also its output can be redirected or piped as input to another program.

  $awk '{printf "%-9s %5d\n", $1,$2+$3+$4}' marks.pu | sort –r + 1 > result

```
sekhar@DESKTOP-UVKV03T:~$ awk '{printf "%-9s %5d\n",$1, $2+$3+$4}' marks.pu | sort -r +1 >result
sekhar@DESKTOP-UVKV03T:~$ cat -n result
     1  Darshana   276
     2  Prasanna   240
     3  Radhika    235
     4  Vinay      232
```

awk control structures:

- Like many programming languages, awk consists of both decision-making and loop-control structures.
- **The if....else:**
  - The syntax of if-else construct is:

```
if (expression)
{
        statements-1
}
else
{
        statements2
}
```

- The else part of this construct is optional. When the program control comes across this construct, first the expression is evaluated.
- If this evaluation results in true, then only the statements1 part is executed. If false, statements2 part is executed.

awk control structures:

- **The conditional Operator:**
  - The syntax as well as the behavior of this operator is exactly the same as that of the conditional operator available in the C language.
  - The syntax of this operator is:

    expr ? action1 : action2
  - When the program control first comes across this construct, the expr will be evaluated.
  - If this evaluation results in true, then only the action1 part will be executed. Otherwise, only the action2 part will be executed.

awk control structures: Looping

- **The while:**
- This is an entry-controlled loop structure. The syntax of this structure is:

```
while(expression)
{
   statements
}
```

- The statements will be executed repeatedly as long as the expression will be true.

awk control structures: Looping

- **The do:**
- This is an exit-controlled Loop. The syntax of this structure is:

```
do
    Statements
while(expression)
```

- When the program construct comes across this construct, the statements between the keywords do and while are executed once.
- Afterwards, the statements between the do and while are executed repeatedly as long as the expression with the while is true.

awk control structures: Looping

- **The for:**
- This is one of the widely used loop-control structures.
- The syntax of this construct is exactly same as that of for construct available in C. The syntax is:

```
{for(expression1;condition;expression2)
        statements
}
```

- Here, expression1 causes the loop initialization, expression2 updates the loop control variable and the condition performs the necessary limit test.

awk control structures: Looping

- **The break and continue:**
- These are the two statements that also affect the control flow of a loop.
- The break statement breaks out of the loop such that no more iterations of the loop are performed.
- The continue statement stops the current iteration before reaching the end of the loop and starts a new iteration from the top of the loop.

awk control structures: Looping

- **The for … in:**
- This construct is useful in processing associative arrays.
- The syntax of this construct is:

```
for(index_name in array_name)
    Statements
```

- When the program control comes across this construct, the statements are executed for all index values in the mentioned array.

Functions in awk:

- awk also permits the use of functions like many other programming languages.
- Functions are generally used to carry out simple yet important tasks repeatedly or frequently.
- awk has two types of functions:
  - Pre-defined functions.
  - User-defined functions.
- **Pre-defined functions:**
- These are the functions that are directly available in the language. awk has many pre-defined functions.

Predefined Functions:

- **The length function:**
- This function takes one or no arguments. The general format of this function is:

  length(string)

- The execution of this function returns the total number of characters present in the string.
- In case no argument is given, the entire current record (available in $0) will be taken as an argument and its length will be returned.
- **Example:**

  awk 'length($0) == 15 {print $0}' marks.pu

- In the above example, a record of length equal to 15 will be selected and printed. Here, complete record has been printed because of the use of the variable $0.

Predefined Functions:

- **The index function:**
- This function returns the first position of a substring with in a string. The general format of this function is:

  index(string,substring)

- In case the substring is not found, it returns a 0 (zero).
- **Example:**

```
awk '/^R/ {print "The substring ika begins at position number:", index($1,"ika"),"in the pattern",$1}' marks.pu
```

- In the above example, all the records on the input file marks.pu that begin with 'R' have been picked up and all the first fields that have 'ika' as a substring in them have been printed.

Predefined functions:

- **The substr function:**
- This function extracts and returns a substring from a string. It has the following two formats:

  substr(string,position,length)

  substr(string,position)
- The only difference between two formats is in the lengths of the substring extracted and returned.
- Both return the substring from string starting at the position mentioned.
- If a length is specified, then the number of characters of the substring returned is equal to the length mentioned.
- When length is not mentioned explicitly, everything upto the end of the string from the position is returned.

  date | awk '{print "The current year is",substr($6,3)}‘
- In the above command, awk being a filter gets its input from the date command, prints the last two characters of the sixth field starting from the third position.

Predefined functions:

- **The split function:**
- This function splits any given string into elements of an array. The general format of this function is:

  split(string,array,separator)
- The splitting takes place on the basis of the specified separator character.
- If a field is not mentioned, the value of the FS will be taken as the field separator.
- The array's indices start from 1 and go up to a value that is equal to the number of elements in the array.

```
awk -F\| '/murthy/ {split($0,arr_dar, "r"); print arr_dar[2]}' phone.lst
```

- When the above command is executed, the selected record(s) is/are split on the field separator character 'r' and the split elements are saved in the array arr_dar.
- This type of awk command can be used to pick up a full name when only the last name is entered.

Predefined functions:

- **Print functions:**
- There are three print functions in awk.
- They are:
  - **print**- that prints out in an unformatted way.
  - **printf**- that prints out in a formatted way.
  - **sprintf**- that prints out strings in a formatted way.
- By default, all these print functions send their output on to the standard output file.

Predefined functions:

- **The print function:**
- This function prints the specified data on to the standard output. Each print action must be written on a separate line.
- When multiple fields are being printed, they must be separated with commas.
- If nothing is specified, then the entire current line or record is printed.
- By default, the input field separator is taken as the output field separator.
- If necessary, output field separator can be defined using the built-in variable OFS.
- If strings are to be printed, they must be enclosed within quotes.

Predefined functions:

- **The printf function:**
- This function is used to print the data in any formatted manner.
- This function is similar to printf function in C.
- Similar to C, each printf function consists of a format string with in double quotes and a list of zero or more number of elements that could be a variable or an expression or a string.
- The format string contains field specifiers that begin with % sign and ends with a format code.
- This format code holds information regarding the total width in which the data is to be printed, information about left or right adjusted printing as well as information on the required precision.

Predefined functions:

- **<u>The printf function format specifiers:</u>**

%d, %i	decimal integer
%c	single character
%s	string of characters
%f	floating point number
%o	octal number
%x	hexadecimal number
%e	scientific floating point notation
%%	the letter “%”

Predefined functions:

- **The sprintf function:**
- This function uses the same format specifications as the printf function. This function does not print the results.
- It combines two or more fields into one string and returns the resultant string.
- This assigned variable could be assigned to a variable which could be used later in the script.

The sprintf function: Example

```
sekhar@DESKTOP-UVKV03T:~$ cat sprintf.awk
{str_var = sprintf(" " "%-9s  %4d %6d %7d %7d \n", $1,$2,$3,$4,$2+$3+$4)
        len_var=length(str_var)
        print " "len_var" " str_var
}
sekhar@DESKTOP-UVKV03T:~$ awk -f sprintf.awk marks.pu
 41  Radhika      72     67      96     235

 41  Darshana     86     97      93     276

 41  Anil         88     96      91     275

 41  Prasanna     75     86      79     240

 41  Vinay        45     99      88     232

sekhar@DESKTOP-UVKV03T:~$
```

The getline function: (predefined)

- This function helps in getting an input value interactively.
- The input may come either from the standard input file or any other designated file.
- These filenames are given in the form of a string by enclosing it within a pair of double quotes.
- The standard input file used here is /dev/tty.
- This function upon execution returns either 1 or 0 or -1.
  - 1 indicates the successful reading of a line.
  - 0 indicates the reaching of the end-of-file character.
  - -1 is returned under error conditions.

The getline function:

```
sekhar@DESKTOP-UVKV03T:~$ awk 'BEGIN {printf "Enter the name:"
> getline < "/dev/tty"
> print}'
Enter the name:sri vasavi
sri vasavi
sekhar@DESKTOP-UVKV03T:~$
```

The system function: (predefined)

- We can execute any UNIX command using system function.

  awk 'BEGIN { system("date")}‘

- The execution of the above command displays current date as follows:

  Fri Sep 11 20:01:33 IST 2020

- The command to be executed must be within double quotes.

User-defined functions:

- Like any other programming languages, we can write our own functions in awk also.
- Once a function is defined, it may be used as any other built-in function.
- A function is defined using the keyword function, a function name followed by parameters list and the body of the function.
- The general format is:

```
function function_name(argument1, argument2, ...)
{
   function body
}
```

User-defined functions:

```
#!/bin/awk -f
function add(num1,num2)
{
        print num1 + num2
   }
  BEGIN {
          add(ARGV[1],ARGV[2])
}
```

$./add.awk 2 3 #This command adds two numbers 2 and 3 which are being passed at command line.

awk examples:

AWK script to count the number of lines in a file that do not contain vowels:

```
sekhar@DESKTOP-UVKV03T:~$ cat novowels1.sh
#!/bin/awk -f
echo "Enter file name"
read file
awk '$0!~/[aeiou]/{ count++ }
END{print "The number of lines that does not contain vowels are: ",count}' $file
sekhar@DESKTOP-UVKV03T:~$ cat test.txt
sri
vasavi
ptpl
tpg
sekhar@DESKTOP-UVKV03T:~$ sh novowels1.sh
Enter file name
test.txt
The number of lines that does not contain vowels are:  2
sekhar@DESKTOP-UVKV03T:~$
```

Awk examples:

Awk script to find number of characters, words and lines in a file:

```
sekhar@DESKTOP-UVKV03T:~$ cat characters.awk
BEGIN{print "Lines.\t characters \t words"}
#BODY section
{
        len=length($0)
        total_len =len
        print(NR,":\t",len,":\t",NF,$0)
        words =NF
}
END{
        print("\n total")
        print("characters :\t" total len)
        print("lines :\t" NR)
}
sekhar@DESKTOP-UVKV03T:~$ awk -f characters.awk phone.lst
Lines.    characters        words
1 :       18 :     3 mgv   murthy              267757
2 :       20 :     3 hs    prabhakara          245092
3 :       18 :     3 kak   murthy              268088
4 :       18 :     3 vn    narayana            251833
5 :       20 :     3 mgv   krishna             245020

 total
characters :     20
lines : 5
sekhar@DESKTOP-UVKV03T:~$
```