sed and awk

Amitabha Sanyal

The sed model

- sed is a *stream editor*
- usage

  `sed` *options script input_files*

- sed cycle:
  Until all lines are read:
  a. read the next line from the input file into a buffer caled pattern space
  b. process the line in the pattern space according to the script
  c. transfer the contents of pattern space to the output stream.
- A separate buffer called hold space is used by some sed commands to hold and accumulate text between cycles.
- The script may be
  a. inlined: `sed` *'inlined_script' input_files*, or
  b. read from a file: `sed -f` *script_file_name input_files*
- This is the main link for sed.

options

- important options are:
  - `-n` : Don't print lines by default
    ```
    sed -n  '/Deceased/p'  covid_data.csv
    ```
  - `-f scriptfile`: Pick the script from `scriptfile`
  - `-s` : applied command on each file separately
    ```
    sed -n -s '1,10 p' Students.csv covid_data.csv
    ```
    otherwise the command is applied to the concatenation of files.
  - `-r` : Enables usage of extended regular expressions

- This link gives the complete list of options in sed.

script

- A script is of the form:

```
[address] [!] command
...
[address] [!] command
```

- A missing address means all lines. ! is negation: means all lines except the lines specified by `address`.

- `address` can be
  - A single-line address
  - A set-of-lines address
  - Range address
  - Nested address

address

- single line address.
  - show only line 3
    ```
    sed -n '3 p' Students.csv
    ```
  - show only last line
    ```
    sed -n '$ p' Students.csv
    ```
  - substitute "__" with " " on line 10 and print the line
    ```
    sed -n -e '10 s/__/  /' -e '10 p' Students.csv
    ```
    note use of `-e` to join more than one commands in the command line.

- set-of-lines address using regular expressions
  - Show all dual degree students
    ```
    sed -n -r '/19D[0-9]{6}/ p' Students.csv
    ```
    note use of extended regular expressions using `-r`
  - Show all directories in the current directory
    ```
    ls -al | sed -n  '/^d/ p'
    ```

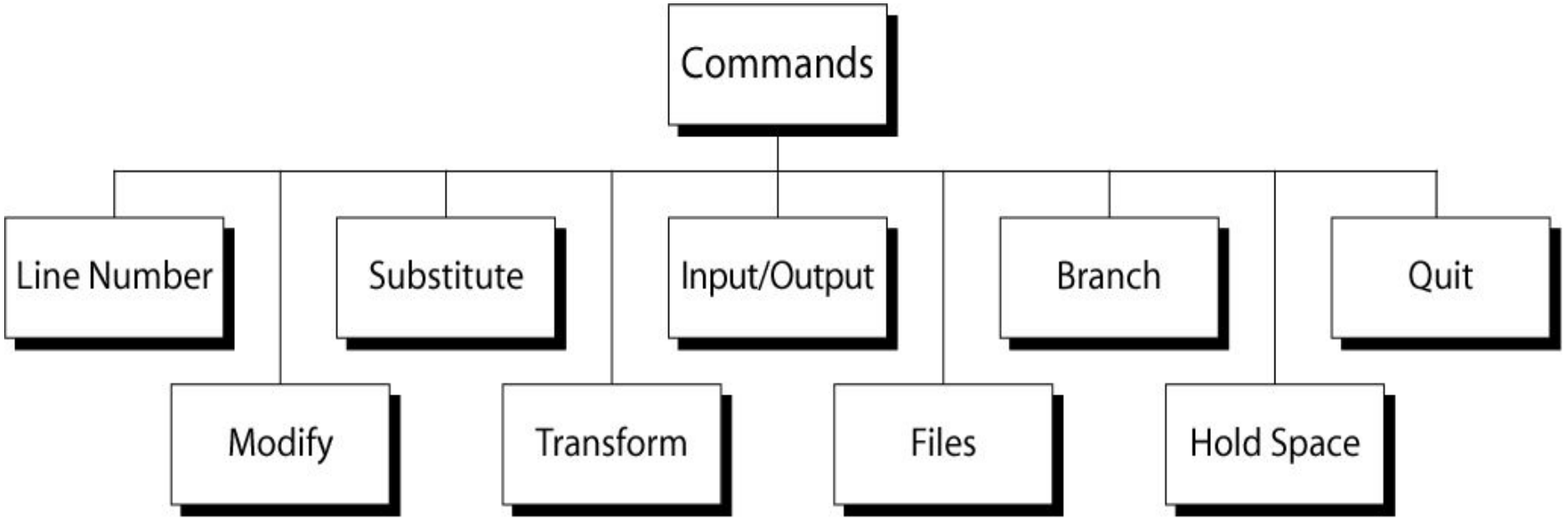
address

- range address.
  - show the lines between 1 and 10
    ```
    sed -n '1,10 p' spy.py
    ```
  - show the lines between an if statement and line 10:
    ```
    sed -n '/^if/,10 p' spy.py
    ```
  - show the lines between a `if` and *its closest* `else`:
    ```
    sed -n '/^if/,/^else/ p' spy.py
    ```

- nested address
  - Show all `print` statements between 20 and 30
    ```
    sed -n '20,30{/print/ p}' spy.py
    ```

- complement address (!)
  - Show all statements other than `print` between 20 and 30
    ```
    sed -n '20,30{/print/! p}' spy.py
    ```

Categories of sed commands

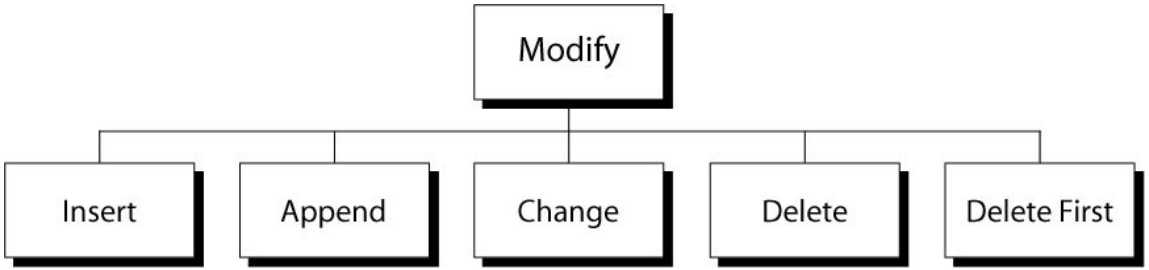


sed commands

- line command (=)
  - prints line numbers of matching statements

```
sed -n '20,30{/print/! =}' spy.py
```

modify commands



sed commands

- modify commands
  - insert: inserts text before address:

```
sed '1 i\
     2019 Batch' Students.csv
```

    inserts the text " 2019 Batch" before the first line.
    can only be used with the single-line and set-of-lines address types
  - append: appends text after address. Can only be used with the single-line and set-of-lines address types.
  - change: replaces an entire matched line with new text

```
sed -n -r -e  '/19D[0-9]{6}/ p'  -e '/19D[0-9]{6}/ c\ Dual Degree Student' Students.csv
```

  - delete: deletes an entire matched line. Both change and delete can be used with any address type.

THE SUBSTITUTE (S) COMMAND

- Syntax:

  `[addr1][,addr2] s/search/replace/[flags]`

- Replaces the string that matched the search pattern with the replacement string. `search` pattern can be a regular expression
- flags:
  - global (g), i.e. replace all occurrences
  - specific substitution count (integer), default 1
- Example: Remove all occurrences of `the "` character.

```
sed -r 's/\"//g' Students.csv
```

THE SUBSTITUTE COMMAND

- Substitution back references `&` and `\n`
- `&`
  - 

  - Convert roll nos to email addresses:

```
sed -r 's/19[0-9A-Z]{7}/&@iitb.ac.in/' Students.csv
```

- `\n`
  - 

  - Swap the roll no and the name columns:

```
sed -r 's/(\"19[0-9A-Z]{7}\",)(\"[ a-zA-Z]*\",)/\2\1/'
Students.csv,  or, better
sed -r 's/(\"19[0-9A-Z]{7}\"),(\"[ a-zA-Z]*\")/\2,\1/'
Students.csv
```

THE TRANSFORM COMMAND `y`

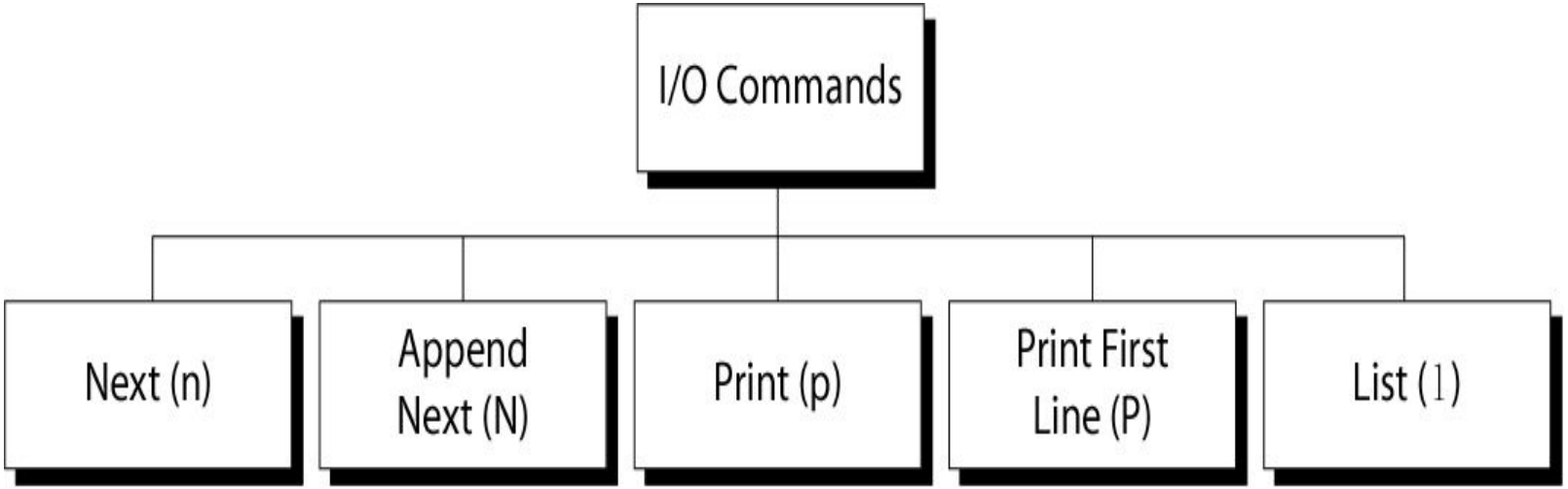
- Syntax:

```
[addr1][,addr2]y/listofChars1/ListofChars2/
```

  - Similar to the `tr` command. Replaces characters from the first list to corresponding characters of the second list.
  - `listofChars1` and `listofChars2` should have the same length.
  - Example:

```
sed '1,4y/19D/19d/' Students.csv
```

SED IO COMMANDS



SED IO COMMANDS `n` AND `N`

- `n` (default `sed` workflow):
  - Read a line from input stream to pattern space, removing any trailing newline.
  - Execute commands if address specifications are met.
  - Unless '`-n`' option is used, print contents of pattern space to output stream.
- `N`
  - After reading a line to the pattern space, add a newline, and then append the next line of input to the pattern space.
  - Execute commands.
  - Print to output stream. Useful for processing to two or more lines at the same time.

```
sed -r 'N; s/\n//g' Students.csv
```

  - Joins adjacent lines removing the intervening newline.

SED IO COMMANDS P AND P

- p
  - copies the entire contents of the pattern space to output
  - will print same line twice unless the option '`-n`' is used
- P
  - prints only the first line of the pattern space
  - prints the contents of the pattern space up to and including a new line character
  - any text following the first new line is not printed

```
sed -n 'N; p' Students.csv | wc -l
```

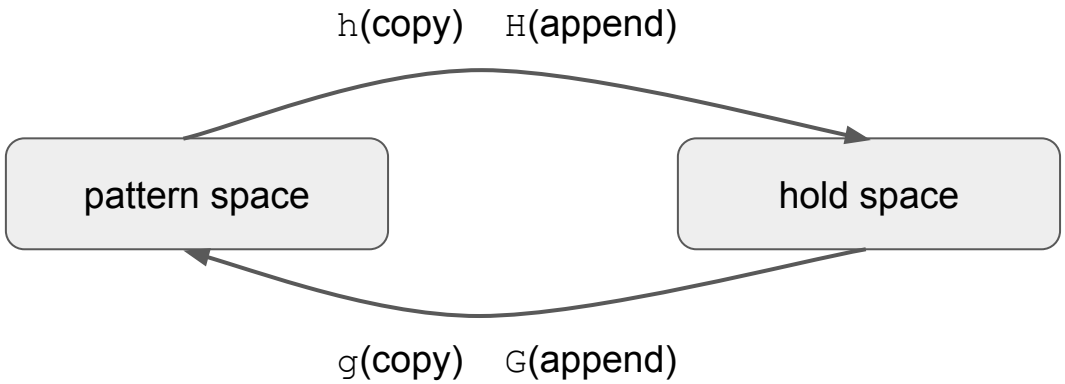
returns 146

```
sed -n 'N; P' Students.csv | wc -l
```

returns 73

SED IO COMMANDS `g`, `G`, `h` AND `H`.

- To move text back and forth between pattern space and hold space.



- Example: transfer the names of the dual degree students to the end.

```
sed -r -e '/19D[0-9]{6}/ H' -e '/19D[0-9]{6}/ d'-e '$ G'
Students.csv
```

SED IO COMMANDS G, G, H AND H.

- Same example as a script:

```
#!/bin/sh
sed -r '
 /19D[0-9]{6}/ {
    H
    d
}
 $ {
    G
}' Students.csv
```

- The curly braces and the sed commands must appear on different lines.

SED IO COMMANDS `R` AND `W`.

- Allows reading from and writing to files.
  - `r filename` - reads `filename` and writes it to the output stream. It is not copied to the pattern space.

    ```
    sed '$r extras.csv' Students.csv
    ```

    adds `extras.csv` to the output stream at the end of `Students.csv`

  - `w filename` - writes from the pattern space to `filename`.

    ```
    sed -r -n '/19D[0-9]{6}/w dd-students.csv' Students.csv
    ```

    writes the matched students to the file `dd-students.csv`

SED IO COMMANDS B AND Q.

- b - branch unconditionally to label (or end of script)

```
sed -n -r '
 /19D[0-9]{6}/ b save
w others.csv
b
:save
w dd-students.csv
' Students.csv
```

•Can be up to 7 characters
•Must be on a line by itself
•Must begin with a colon
•No spaces after it and after the colon

Separates dual degree students and others into separate files.

- q - quits sed.

```
sed -e '50q' datafile
```

Quits after printing the first 50 lines.

AWK

- Named after Aho, Weinberger and Kernighan
- Scripting language used for manipulating data and generating reports

- The awk cycle:
  - scans a file line by line
  - splits each input line into fields
  - compares input line/fields to pattern
  - performs action(s) on matched lines

- Useful for:
  - transforming data files
  - producing formatted reports

AWK SYNTAX

Basic awk syntax:

```
awk [options] 'script' file(s)
awk [options] -f scriptfile file(s)
```

Options:

- `-F` to change input field separator
- `-f` to name script file

Basic awk program

Consists of patterns & actions:

```
pattern {action}
```

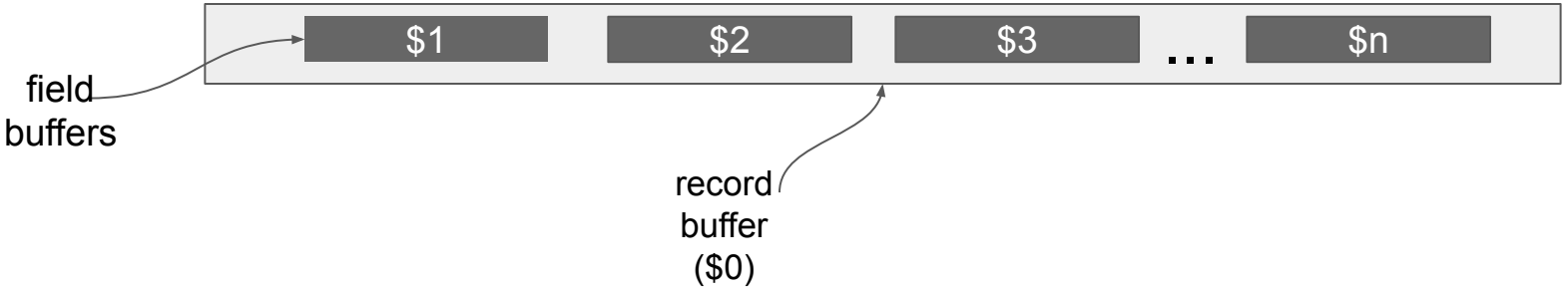
- if pattern is missing, action is applied to all lines
- if action is missing, the matched line is printed
- must have either pattern or action

Example:

```
awk '/for/' testfile
```

- prints all lines containing string “for” in testfile

AWK'S VIEW OF DATA



- A field is a unit of data in a line
- Each field is separated from the other fields by a field separator
  - The default field separator is whitespace
- A record is the collection of fields in a line
- A data file is made up of records

Example of a record:

"190050131","Kandibanda Vishwanth","__","Advisor : Prof. Ashutosh Kumar Gupta"

AWK PREDEFINED VARIABLES

`FS` -- Field separator (default=whitespace)
`RS` -- Record separator (default=\n)
`NF` -- Number of fields in current record
`NR` -- Number of the current record
`OFS` -- Output field separator (default=space)
`ORS` -- Output record separator (default=\n)
`FILENAME` -- Current filename

Example:

```
ls -al | awk '{print NR, $9}'
```

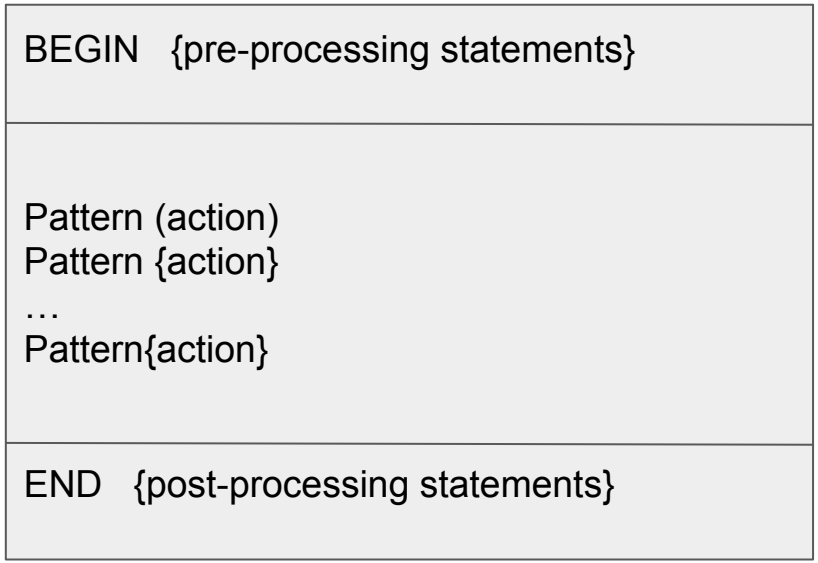
will number and print the files in the current directory.

```
awk -F, '/Aniket/{print NR, $1, $2}' Students.csv
```

print the line number, roll number and the full names of students named 'Aniket'

AWK SCRIPT STRUCTURE

- awk scripts are divided into three major parts:



- comment lines start with #

- BEGIN: Pre-processing
  - processing to be done before awk starts reading records from the input file
  - useful for initialization: initialize variables, create report headings etc.
- Body
  - contains logic to be applied to input file, one record at a time.
- END: Post-processing
  - contains processing to be done the records in the input file have been processed
  - Useful for reporting aggregates (total, average), conclusion.
- BEGIN and END are patterns

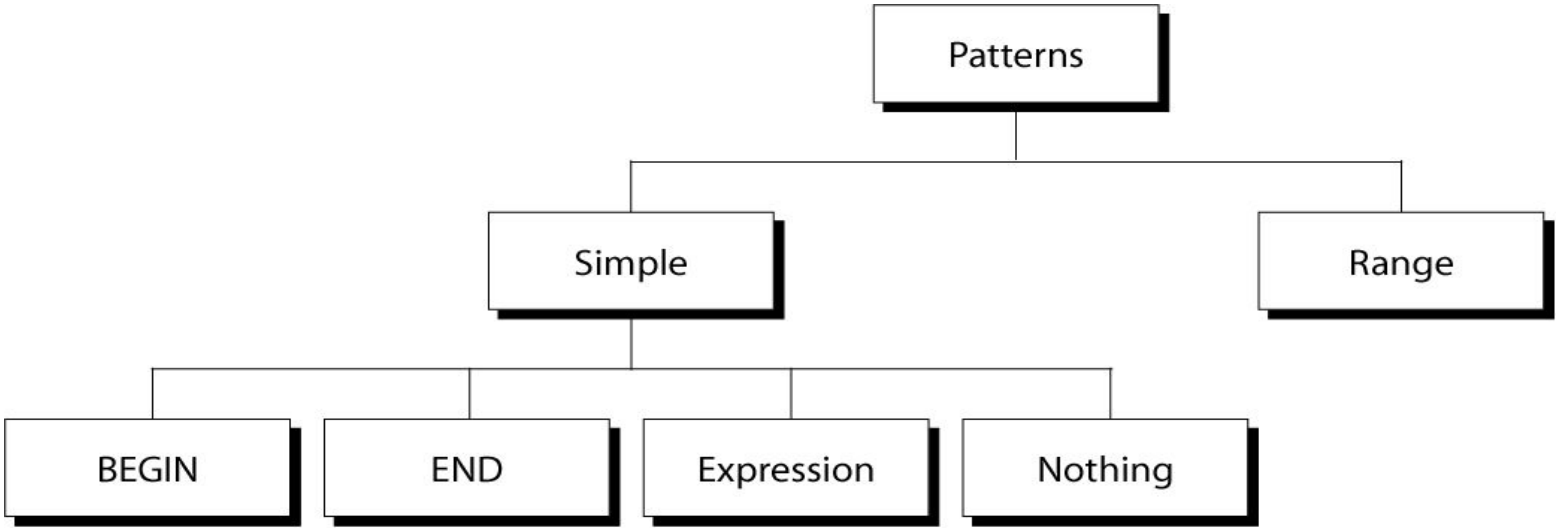
STRUCTURE OF BODY

- pattern {statement}

- pattern {statement; statement...statement}

- pattern {
  statement
  statement
  …
  statement
  }

CATEGORIES OF PATTERNS



PATTERN TYPES

- Match expressions
  - regular expression enclosed by ‘/’s (as in sed)
    - matches an occurrence in entire input record
    - example:

      ```
      awk -F '/special/ {print}'
      ```

      Matches a record with the text `special` anywhere
  - explicit pattern-matching expressions ~ (match), !~ (not match)
    - Matches fields
    - example:

      ```
      awk -F, '($1~/19D[A-Z0-9]+/){print NR, $0}' Students.csv
      ```

      print records whose first field match the regexp `19D[A-Z0-9]+`
    - `/special/{print}` is the same as `($0~/special/)`

Pattern types

- Expressions made up of:
  - Arithmetic operators: `+`, `-`, `*`, `/`, `%` (modulus), `^` (exponential)
  - Relational operators: `<`, `<=`, `==`, `!=`, `>`, `>=`
  - Boolean operators: `&&`, `||`, `!` (not)

- Examples:

```
awk '$3 * $4 > 500 {print $0}' file
awk '($2 > 5) && ($2 <= 15){print $0}' file
awk '$3 == 100 || $4 > 50' file
```

Range Patterns

Matches ranges of consecutive input lines (much like sed)

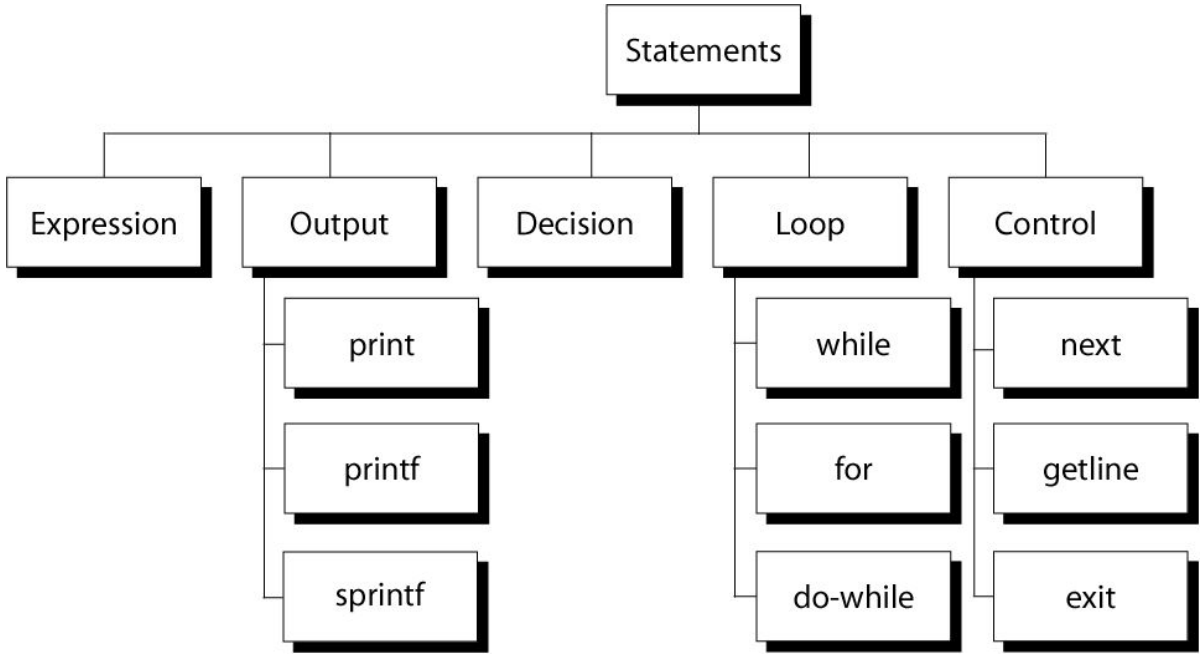
- Syntax:

```
pattern1, pattern2 {action}
```

  - pattern can be any simple pattern
  - **pattern1** turns action on
  - **pattern2** turns action off
- Example:

```
awk /190050002/,/190020010/{print} Students.csv
```

AWK ACTIONS


Statements
Expression
Output
Decision
Loop
Control
print
printf
sprintf
while
for
do-while
next
getline
exit

AWK EXPRESSIONS

- Made up of variables, constants and operators.
  a. Variables can be user variables, field variables and awk built-in variables.
  b. constants are numeric and string constants, associative arrays.
- Example 1: Feed the program input from `wc -c *`

```
awk '
 BEGIN {
      lines=0; total=0; }
 {
      lines++; total+=$1; }
 END {
      print lines " lines read"; print "total is ", total;
      if (lines > 0 ) {
     print "average is ", total/lines;
      } else {
     print "average is 0"; }}'
```

VARIABLES AND THEIR SCOPES

- How to pass a shell variable to a awk script
  a. $1, $2 etc lose their shell variable identities. They become field variables.
  b. The scope of a user defined variable does not extend to an awk script. The `-v` switch has to be used
  c. The shaded part introduces an awk variable `v` in the shell, and passes to it the value of the shell variable `v`. The variable names need not be the same.

```
printf $1
printf "\n"
v=$1
awk  '
BEGIN {
    print "Printing v"; print v}
/190260036/ {
    print $1; print "\n"}'
Students.csv
```

```
printf $1
printf "\n"
v=$1
awk -v v="$v" '
BEGIN {
    print "Printing v"; print v}
/190260036/ {
    print $1; print "\n"}'
Students.csv
```

ASSOCIATIVE ARRAYS

- Does not have predefined indices. Indices get defined by use.

```
awk -F, '
#($9~/[A-Z][A-Z]/)
{
    state[$9]=state[$9]+$10
}
END {
    for (i in state) {
        print state[i], i;
    }
}
```

- `$9` is state code (`UP`,`OR` etc) and `$10` is the number of deaths. state is an array from state code to integers because of usage.

OUTPUT STATEMENTS

- `print`: print easy and simple output
- `printf`: print formatted (similar to C printf)
- `sprintf`: format string (similar to C sprintf)

Examples:

```
awk -F, '{print $1,$2 | "sort"}' Students.csv
awk -F, '{print $1,$2 | "sort -k 2"}' Students.csv
awk '{printf("z is %5.3f \n", z)}'
awk '{printf("The character is %c \n", x)}'
awk '{ text = sprintf("1: %d - 2: %d", $1, $2)
       print text}
```

STATEMENTS

- if statements
- for statement

```
for (i = 1; i <= NR; i++)
{
        total += $i
        count++
}
```

- while-do,do-while
- break,continue

```
for (x = 0; x < 20; x++) {
   if ( array[x] > 100) continue
   printf "%d ", x
   if ( array[x] < 0 ) break
 }
```

EXAMPLE

```
BEGIN {
      FS= ","; total = 0
      print "      Covid Data    Date: 24th July"
      print "     State     Dist.                Cases"
      print "    ================================="
}
{
      printf("%6s       %-25s%d\n", $9, $7, $10) | "sort -k 1"
      state[$9] += $10; total += $10
}
END {
      print "      ========================================"
      print "      State                                Total"
      print "      ========================================"
      for (i in state)
      {
        printf("      %-15s                        %d\n", i, state[i])
      }
      print "      ========================================"
      print "      Total Cases (India):                  " total}
```