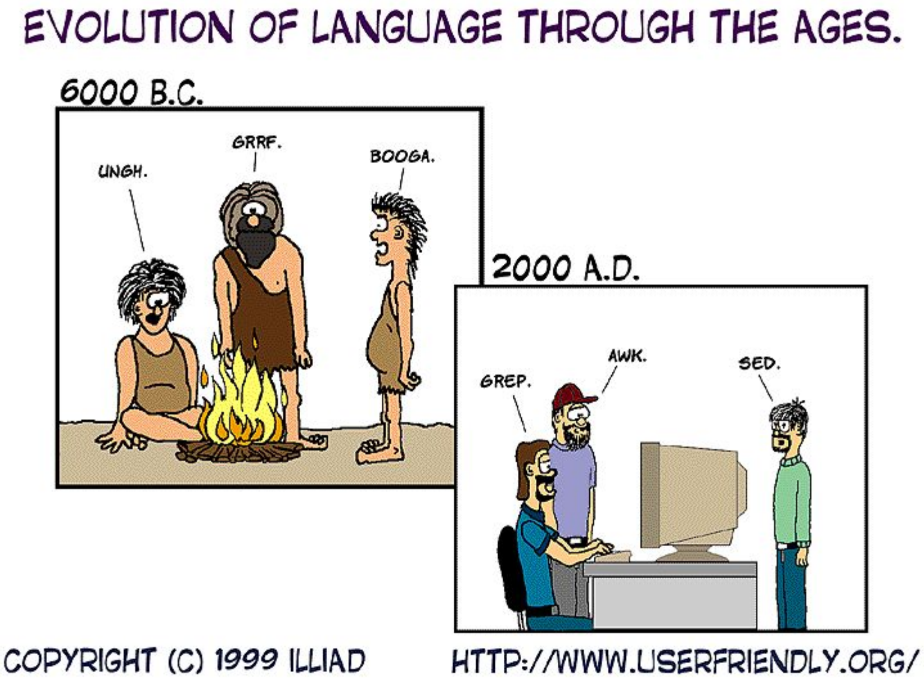
Sed and awk

Kameswari Chebrolu


EVOLUTION OF LANGUAGE THROUGH THE AGES.
6000 B.C.
UNGH.
GRRF.
BOOGA.
2000 A.D.
GREP.
AWK.
SED.
COPYRIGHT (C) 1999 ILLIAD
HTTP://WWW.USERFRIENDLY.ORG/

Sed/Awk

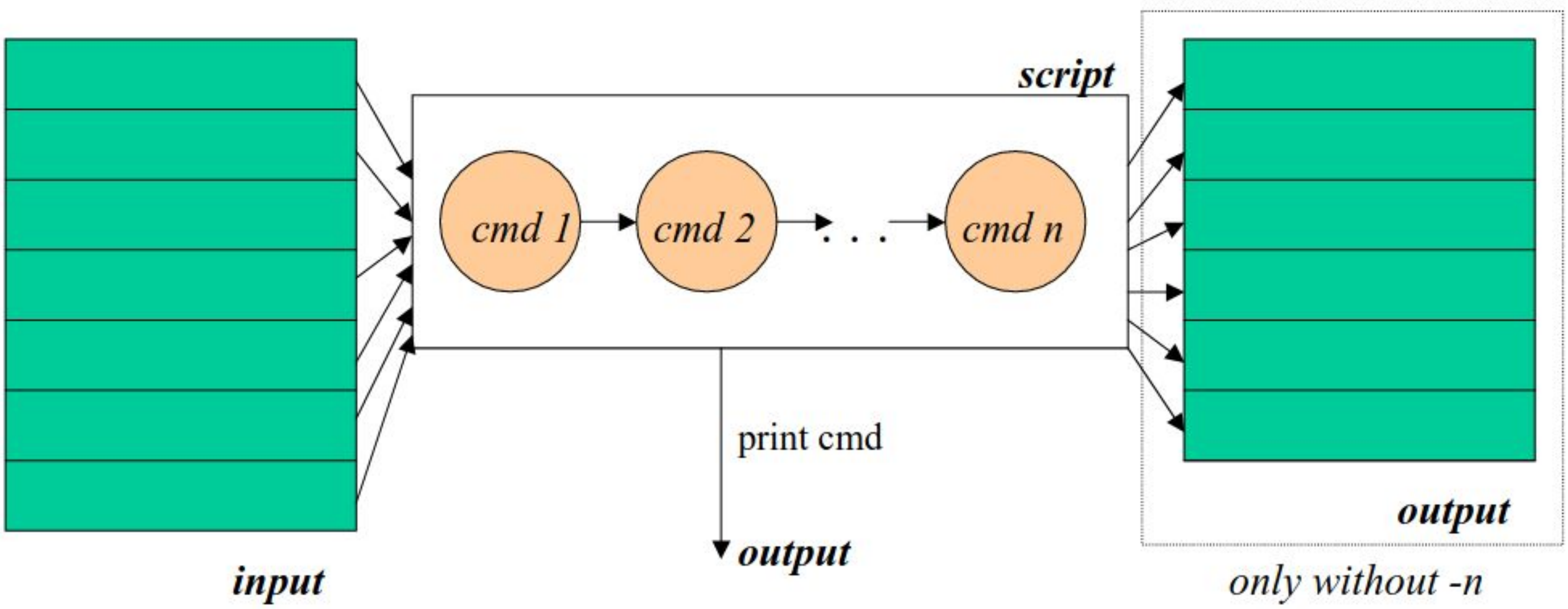
- Powerful text processing utilities
- sed: a non-interactive text editor
- awk: a field-based pattern processing language with a C-style syntax
- Both use
  - regular expressions
  - read input from stdin/files, and output to stdout

Sed (stream editor)

sed: Stream-oriented, Non-Interactive, Text Editor

- Stream: Look one line at a time
- Non-interactive: editing commands come in as script
- Text editor: change lines of a file
  - Sed is more a filter
  - Original input file is unchanged (unless with -i option)
  - Results sent to standard output (can be redirected to a file)

Sed Control Flow



Example

- Replace 'hello' with 'hi' in a file file.txt

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

```
sed -f commands.sed file.txt
```

- A script is a file made of commands
- Commands also specify
  - regular expression to match a pattern (and/or)
  - an address range (line nos in a file)
- Single quotes (' ') are used to delimit the command being executed

- Commands are applied in order to each input line
  - sed reads the first command and checks address/pattern against the current input line
    - match, command is executed
    - no match, command is ignored
  - sed then repeats this action for every command in the script file
    - Note: If a command changes the input, subsequent command will be applied to the modified line
  - Reached end of the script? output the (modified?) line unless “-n” option is set

Delete Command

- Syntax:

  sed ‘ADDRESSd’ filename

  sed ‘/PATTERN/d’ filename
- Examples:
  - ‘6d’ (delete line 6)
  - '1, 5d’ (delete lines 1 to 5)
  - '/^$/d' (remove empty lines)

Substitute Command

- Syntax:

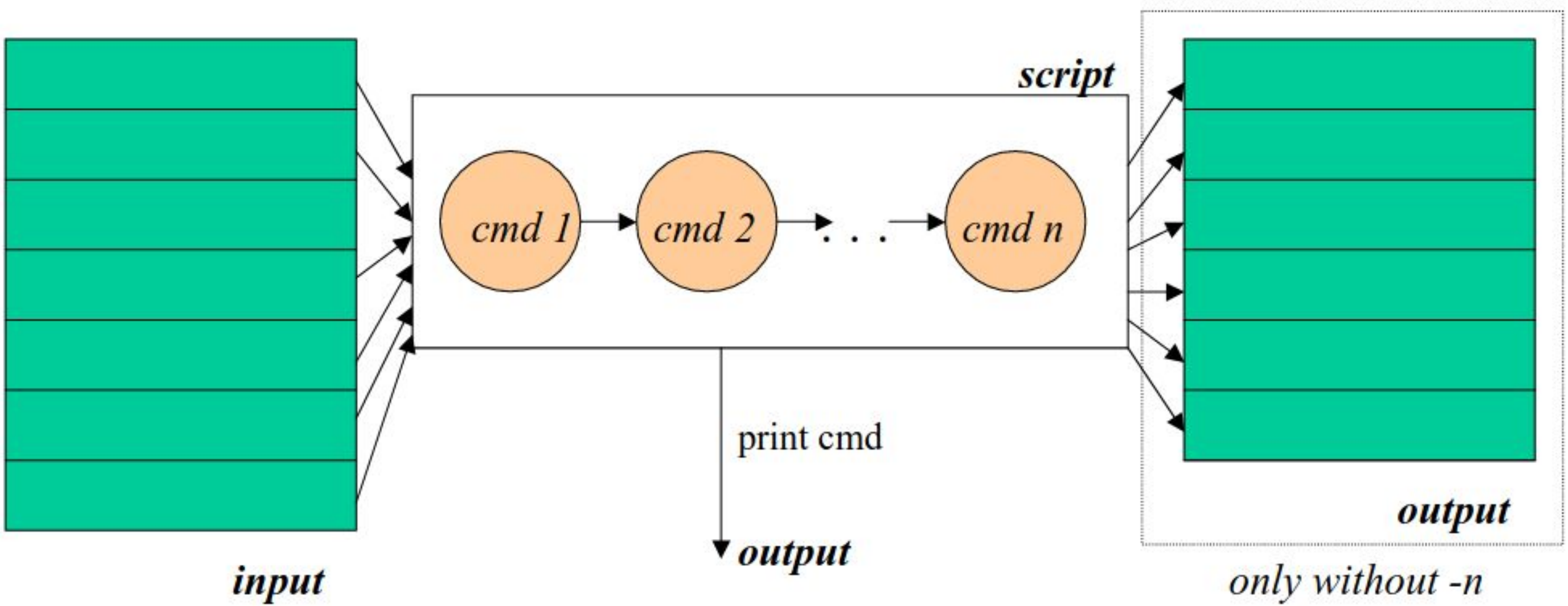
  `sed ‘address(es) s/pattern/replacement/[flags]’ filename`

  - Flags:
    - a number indicating the occurence
    - g: global, replace all occurrences of pattern in pattern space
    - p: print contents
    - I: case insensitive
  - E.g.
    - `sed 's/wolf/fox/' bigfile`
    - `sed '3 s/wolf/fox/' bigfile`
    - `sed 's/wolf/fox/3gI' bigfile`
    - `echo "Welcome To The Course CS104" | sed 's/\(\b[A-Z]\)/\(\1\)/g'`

Print Command

- Syntax:
  sed 'ADDRESSp’ filename
  sed '/PATTERN/p' filename
- Used to print the matched pattern
  - Often used with the -n option
  - Without -n option, sed automatically prints each line after applying editing commands
  - With -n option, sed will only print output when explicitly instructed using the p command
- Syntax: [address/pattern]p
  - sed '1p' bigfile
  - sed -n '1p' bigfile
  - sed -n '/scream/p' bigfile

Sed Control Flow



Append/Insert/Replace

- Append: [address/pattern]a file
  - Append places text after the current line in pattern space
  - sed '2a tomato' fruits
- Insert
  - Insert places text before the current line in pattern space
  - sed '2i tomato' fruits
- Replace
  - Replaces
  - sed '2c aam' fruits
  - sed '/mango/c aam ' fruits

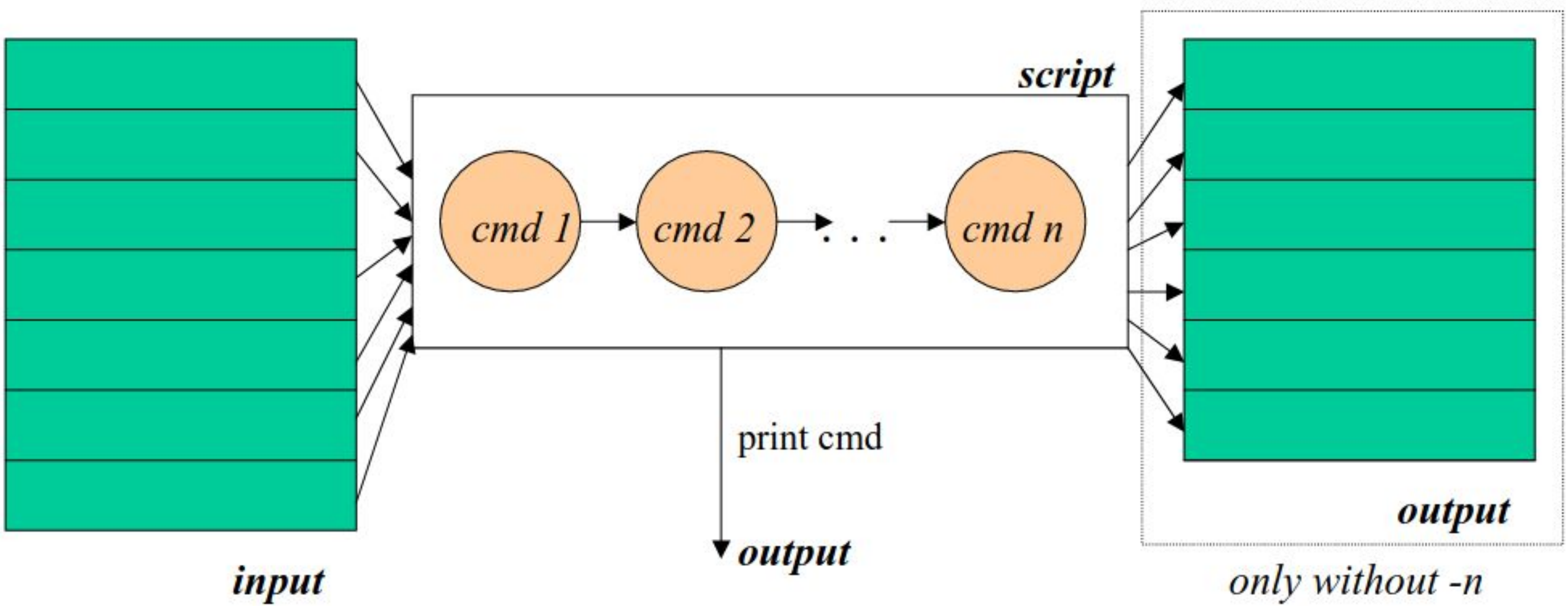
quit

- Quit causes sed to stop reading new input lines
  - Once a line matches the pattern/address, the script terminates
  - Can help save time when you want to process just some portion at beginning of file
  - sed '5q' fruits (print first 5 lines and quits)

Multiple Commands

- Separate instructions with a semicolon
  - `sed 's/mango/aam/; s/banana/kela/;' fruits`
- Precede each instruction by -e
  - `sed -e 's/mango/aam/' -e 's/banana/kela/' fruits`
- Order is important!
  - `echo "please fix the light bulb!" | sed 's/light/tube/g; s/bulb/light/g'`
  - `echo "please fix the light bulb!" | sed 's/bulb/light/g; s/light/tube/g'`

Sed Control Flow



Backward References

- Can reference previously captured groups that match some regular expression pattern
  - Helps reuse parts of the pattern that you've already matched
- Represented using \1, \2, and so on
  - Number corresponds to the order of the captured group
  - \1 refers to the first captured group, \2 refers to the second captured group, and so forth
- You can define a captured group using parentheses ( ) in the regular expression

- Example: \([A-Z]\)\1
  - ([A-Z]) is a group that captures any uppercase letter
  - \1 matches whatever was captured
  - echo "AA BB CC" | sed 's/\([A-Z]\)\1/XX/g' will result in XX XX XX
- & is used to reference the entire matched pattern
  - Example:
    - echo "I have 5 apples and 3 bananas." | sed 's/[0-9]\+/[&]/g'
    - Outputs: I have [5] apples and [3] bananas.

In Place Editing

- -i option: modify files directly instead of just printing the output to the terminal
  - No need for redirection (>)
  - sed -i 's/hi/hello/' file1.txt

Script

- Not practical to enter many commands on the command line
- Create a script file that contains instruction and use -f option
  - sed -f script-file file
- Note: can use sed in a bash script also

Drawbacks

- Not possible to go backward in the file
- No way to do forward references
- No facilities to manipulate numbers
- Cumbersome syntax

References

- https://www.gnu.org/software/sed/manual/sed.html
- https://linuxhint.com/50_sed_command_examples/

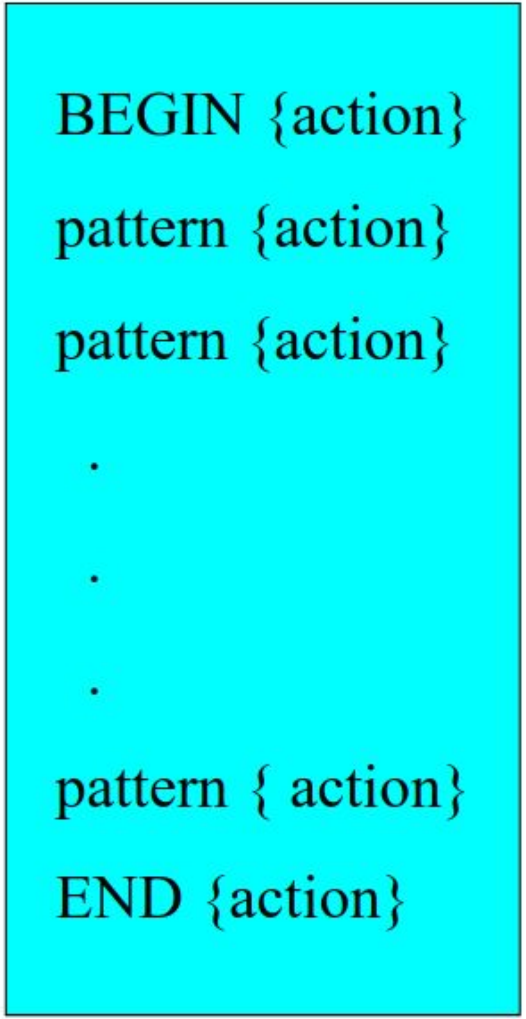
awk

- awk named after inventors: Alfred V. Aho, Peter J. Weinberger, and Brian W. Kernighan.
- Like sed, stream-oriented and interprets a script of editing commands
- Unlike sed
  - Supports a programming language modeled on C Language
    - expressions, conditional statements, loops etc
  - awk processes fields while sed only processes lines
- nawk (new awk) is the new standard for awk
  - Designed to facilitate large awk programs
  - gawk is a free nawk clone from GNU

Structure

An awk program consists of:

- An optional BEGIN segment
  - To execute prior to reading input
- Pattern - action pairs
  - Processing input data
  - Action enforced in { }
- An optional END segment
  - To execute after end of input data



```
BEGIN {action}
pattern {action}
pattern {action}
  .
  .
  .
pattern { action}
END {action}
```

Simple example

```
awk '{print}' file1.txt file2.txt
```

(Prints all lines of the file)

Note: AWK can process multiple files sequentially

Another Example

```
ls | awk '
BEGIN { print "List of jpg files:" }
/\.jpg$/ { print }
END { print "All done!" }
'
```

print

- print statement is used to output data
- Can be used to print expressions, variables, fields, or the entire line from the input
- Syntax: print expression
  - expression: Can be a variable, field, string, or any arithmetic expression
  - If no expression is provided, AWK prints the entire current record (line)

print vs printf

- Syntax: print item1, item2, ...
  - Best for Simple field-based printing
- Syntax: printf "format_string", item1, item2, …
  - Custom formatting like padding, alignment, decimals

Records

Awk views each input line as a record
Default record separator is newline

Each word on that line is delimited by spaces or tabs or comma etc, as a field
$0 represents the entire input line
$1, $2, ... refer to the individual fields on the input line
Awk splits the input record before the script is applied

Built in Variables

NR: Number of records (lines) processed so far

NF: Number of fields in the current record

FS: Input field separator (default is whitespace)

OFS: Output field separator (default is space)

RS: Input record separator (default is newline \n)

ORS: Output record separator (default is newline \n)

```bash
#!/bin/bash

#Awk inside bash. To run, do below
#chmod +x 02-variables.sh
#./02-variables.sh

# Create a sample CSV file
cat <<EOF > sample.csv
Amit,30,Engineer
Priya,25,Doctor
Raj,40,Teacher
Neha,35,Architect
EOF

echo "Original CSV File:"
cat sample.csv
echo ""
```

```
# AWK script to process the file
awk '
BEGIN {
   FS = ",";  # Input field separator is a comma
   OFS = "\t"; # Output field separator is a tab
   RS = "\n";  # Input record separator (default is newline)
   ORS = "\n"; # Output record separator (default is newline)

   print "Processing CSV file...\n";
}

{
   print "NR=" NR, "NF=" NF, "Record:", $0;
   print "Formatted Output:", $1, $2, $3;
}
' sample.csv

# Cleanup
rm sample.csv
```

User defined Variables

- Need no declaration
  - Take on numeric or string value based on context
  - By default, variables are initialized to the null string which has numerical value 0
- Syntax: variable_name=value
- Example: sum is a user defined variable

```
awk '{ sum = sum + $1 } END { print sum }'
file.txt
```

Arithmetic

- Much better support than bash
- AWK supports a wide range of arithmetic operations:
  - Examples:
    - x = x + 1
    - y = y + $2 * $3
    - Lot of built-in functions: sin, cos, atan, exp, int, log, rand, sqrt etc

```bash
#!/bin/bash

# Arithmetic example using variables
awk 'BEGIN {
   a = 50;
   b = 20;
   # Performing a basic arithmetic operation
   result = a + b * 5 + a / b;
   print "Expression value (a + b * 5 + a / b) = ", result
}'

# Example with trigonometric operations and formatted output
awk 'BEGIN {
   PI = 3.14159265;
   a = 60;
   # Calculating cosine of 60 degrees
   result = cos(a * PI / 180.0);
   printf "The cosine of %f degrees is %f\n", a, result
}'
```

```
# Create a sample file (students.tsv) for the demonstration; tsv is tab separated
cat <<EOF > students.tsv
Anil 22 85
Bobby 23 92
Chandu 21 88
Darshan 25 75
EOF

# Arithmetic operation involving addition and multiplication on a file
awk '{
   # Multiply the second field by 2 and add the first field
   result = $1 + $2 * 2;
   print "Sum of first field + second field * 2 = ", result
}' students.tsv

# Cleanup: Remove the sample file
rm students.tsv
```

Operators

- Arithmetic Operators:

  + : Addition
  - : Subtraction
  * : Multiplication
  / : Division
  % : Modulus

- Relational Operators:

  == : Equal to

  != : Not equal to

  < : Less than

  > : Greater than

  <= : Less than or equal to

  >= : Greater than or equal to

- Logical Operators:

  && : Logical AND

  || : Logical OR

  ! : Logical NOT

- String Operators:

  ~ : Matches regular expression

  !~ : Does not match regular expression

- Assignment Operators:
  = : Assignment
  += : Addition assignment
  -= : Subtraction assignment
  *= : Multiplication assignment
  /= : Division assignment
  %= : Modulus assignment

- Increment/Decrement Operators:
  ++ : Increment
  -- : Decrement

Conditionals

- If
- If else
- If else if

```
if (condition) {
   action-1
   action-2
   .
}


if (condition) {
   action-1
   action-2
   .
} else {
   action-a
   action-b
   .
   .
}
```

```
if (condition) {
   action-1
   action-2
   .
} else if (condition) {
   action-1
   action-2
  .
} else {
   action-1
   action-2
   .
}
```

```
#!/bin/bash

# Create a sample CSV file
cat <<EOF > sample.csv
Amit,30,Engineer
Priya,25,Doctor
Raj,40,Teacher
Neha,35,Architect
EOF

echo "Original CSV File:"
cat sample.csv
echo ""

# AWK script to perform various operations
awk '
BEGIN {
   # Initialize variables
   FS = ",";    # Field separator is a comma
   OFS = "\t";  # Output field separator is a tab
   RS = "\n";   # Input record separator (default is newline)
   ORS = "\n";  # Output record separator (default is newline)

   # Print header
   print "Performing operations on CSV file..."
}
```

```
{
   # String Concatenation
   full_description = $1 " is a " $3;  # Concatenate name and occupation
   print "Description:", full_description;

   # Relational Operation
   if ($2 > 30) {
       print $1 " is older than 30";
   } else {
       print $1 " is 30 or younger";
   }

   # String Matching
   if ($3 ~ /Doctor/) {
       print $1 " is a Doctor";
   } else {
       print $1 " is not a Doctor";
   }

   # Conditional operation (ternary operator)
   experience = ($2 >= 30) ? "Experienced" : "Less Experienced";  # Ternary operation
   print $1 " is " experience;
```

```
# Modulus Operation
   if ($2 % 2 == 0) {
       print $1 ": age is even";
   } else {
       print $1 ": age is odd";
   }

   # Increment/Decrement Operator (Increment age by 1)
   $2++;  # Increment age by 1
   print $1 " new age after increment: ", $2;


   # Logical AND operator
   if ($2 >= 30 && $3 == "Engineer") {
       print $1 " is an Engineer and older than or equal to 30.";
   } else {
       print $1 " does not meet the Engineer and age condition.";
   }
```

```
# Logical OR operator
   if ($2 <= 30 || $3 == "Doctor") {
       print $1 " is either 30 or younger, or a Doctor.";
   } else {
       print $1 " does not meet either condition (age <= 30 or Doctor).";
   }
}
' sample.csv

# Cleanup
rm sample.csv
```

Loops

- Loops: for, while, do--while
- Break, continue and exit also possible

```
for (initialization; condition; increment/decrement)
   action



while (condition)
   action



do
   action
while (condition)
```

```
awk '
BEGIN {
   print "Loop Demonstrations:\n";

   # while loop
   i = 1;
   while (i <= 5) {
       print "While Loop Iteration:", i;
       i++;
   }
   print "";

   # do-while loop
   j = 5;
   do {
       print "Do-While Loop Iteration:", j;
       j--;
   } while (j > 0);
   print "";
```

```
  # for loop
    for (k = 1; k <= 5; k++) {
        print "For Loop Iteration:", k;
    }
    print "";

    # break and continue example
    for (m = 1; m <= 5; m++) {
        if (m == 3) {
            print "Skipping iteration", m, "using continue";
            continue;
        }
        if (m == 5) {
            print "Breaking loop at", m;
            break;
        }
        print "For Loop Iteration:", m;
    }
    print "";

    print "End of loop demonstrations.";
}'
```

Arrays

- Supports associative arrays
  - arrays are indexed by strings rather than numbers
  - E.g. arr[2]=6 or grade[ram]=AA
  - No need to declare the size of an array in advance
  - Loop order is not guaranteed (associative arrays are unordered)

- Access elements, use the key inside the array reference, like array[key]
- Initialize an associative array directly by assigning a value to a specific key: array["key"] = value
- Delete a key-value pair using the delete keyword: delete array["key"]
- Loop through the keys of an associative array using the for (key in array) loop structure
- If a key doesn't exist in the array, trying to access it will return a null value (empty string or 0, depending on the context)

```
#!/bin/bash

# AWK script to demonstrate associative arrays
awk '
BEGIN {
   # Declare an associative array with student names as keys and grades as values
   grades["Amit"] = 85;
   grades["Priya"] = 92;
   grades["Raj"] = 78;
   grades["Neha"] = 95;

   # Print the entire array; Notice that the order of printing not same as above
   print "Initial Student Grades:";
   for (name in grades) {
       print "Student:", name, "=> Grade:", grades[name];
   }
```

```
# Update a students grade
   grades["Priya"] = 98;
   print "\nAfter Updating Grade:";

   for (name in grades) {
       print "Student:", name, "=> Grade:", grades[name];
   }

   # Delete a students record
   delete grades["Raj"];
   print "\nAfter Removing Record:";  # Same fix for the apostrophe

   for (name in grades) {
       print "Student:", name, "=> Grade:", grades[name];
   }
```

```
# Array length (manual counting of elements)
   count = 0;
   for (name in grades) {
       count++;
   }

   print "\nTotal Students:", count;
}
'
```

Built in functions

- AWK provides an extensive library of built-in functions
- String Functions
  length(string)
  substr(string, start, length)
  index(string, substring)
  split(string, array, delimiter)
  tolower(string)
  toupper(string)
  sprintf(format, value, ...)

- Mathematical Functions

  sqrt(x)
  int(x)
  sin(x)
  cos(x)
  tan(x)
  exp(x)
  log(x)
  rand()
  srand(seed)

- Input/Output Functions
  print
  printf(format, value, ...)
  getline

- Other Functions

  system(command)

  systime()

  ENVIRON["var"]

```
#!/bin/bash
# AWK script demonstrating various built-in functions
awk '
BEGIN {
   # --- String Functions ---

   # length(string)
   str = "Hello, AWK!"
   print "Length of string:", length(str)

   # substr(string, start, length)
   print "Substring (3rd to 7th):", substr(str, 3, 5)

   # index(string, substring)
   print "Index of AWK:", index(str, "AWK")

   # split(string, array, delimiter)
   split(str, arr, ",")
   print "Split string at , :", arr[1], arr[2]

   # tolower(string)
   print "Lowercase string:", tolower(str)

   # toupper(string)
   print "Uppercase string:", toupper(str)

   # sprintf(format, value, ...)
   # Similar to printf, except returns the formatted string as a value; does not print directly
   # You can store this string in a variable and then print the variable
   num = 123.456
   formatted = sprintf("Formatted number: %.2f", num)
   print formatted
```

```
# --- Mathematical Functions ---

   # sqrt(x)
   num = 25
   print "Square root of", num, "is", sqrt(num)

   # int(x)
   float_num = 12.75
   print "Integer part of", float_num, "is", int(float_num)

   # sin(x), cos(x)
   angle = 45
   print "Sin(45):", sin(angle)
   print "Cos(45):", cos(angle)

   # --- Input/Output Functions ---

   # print
   print "This is printed using the print function"

   # printf(format, value, ...)
   printf "Formatted number with two decimal places: %.2f\n", num
```

```
# getline (reads a line of input from a file, stdin, or from a string)
   print "Reading from standard input:"
   print "Enter your name:"
   getline name < "-";  # read a line from stdin
   print "Hello,", name


   # --- Other Functions ---

   # system(command)
   print "Running system command (ls):"
   system("ls")

   # systime(), time since unix epoch
   print "Current Unix timestamp:", systime()
   #In a user friendly format via the strftime function
   print "Current date and time:", strftime("%Y-%m-%d %H:%M:%S", systime())

   # ENVIRON["var"]
   print "Value of PATH environment variable:", ENVIRON["PATH"]
}
'
```

Flags

-v (Variable assignment)

- Allows to pass variables from the shell environment into the awk program
- Syntax: awk -v var=value 'awk_program'
  - awk -v var="Hello" '{print var, $0}'
    - Passes the string "Hello" as a variable var to the awk script
    - Since no file is mentioned, it will expect stdin
    - var is printed followed by whatever you type

-f (Program file)

- Used to specify a file containing the awk program
  - Allows to keep the awk script in a separate file rather than writing it inline
- Syntax: awk -f script.awk input_file
- E.g.
  - awk -f script.awk fruits
  - ls | awk -f 01-example.awk

-F (Field separator)

- Used to specify a custom field separator (delimiter) for splitting input lines
  - By default, awk uses whitespace (spaces or tabs)
  - Can change this to any delimiter, such as a comma, colon, or tab
- Syntax: awk -F"delimiter" '{print $1, $2}'
  - echo "Jay,25,Engineer" | awk -F, '{print $1, $2}' # prints Jay 25
- Can also accept regular expressions to specify delimiters
  - echo "apple1orange2banana3grape" | awk -F'[0-9]+' '{print $1, $2, $3, $4}'
    - Splits the input string by one or more digits
    - Resulting in the fields "apple", "orange", "banana", and "grape"

```
#!/bin/bash

# Sample input data (comma-separated values)
echo "Amit,28,Engineer" > data.txt
echo "Priya,32,Doctor" >> data.txt
echo "Raj,17,Artist" >> data.txt

# Using the -F flag to set a comma as the field separator and -v to pass a variable
awk -F, -v min_age=25 '{
   if ($2 >= min_age) {
       print $1, "is a major,", $2, "years old, and is a/an", $3;
   }
}' data.txt

#Another example
awk -F, '{Grp[$8]++} END {for(g in Grp) print g, Grp[g]}' students.csv

#Convert command seperated to tab seperated withe extra formatting done by column
awk -F, '{OFS="\t"; $1=$1; print}' students.csv | column -t -s $'\t' > students.tsv
```

sed+awk

- Can combine through pipe command
- E.g. sed 's/,/ /g' students.csv | awk '{print $2}'

References

https://www.tutorialspoint.com/awk/index.htm