CSCE 3600
Principles of Systems Programming

Regular Expressions: sed and gawk

University of North Texas

CSE

The sed Stream Editor

The sed Stream Editor

- sed is a non-interactive, line-oriented stream editor that processes one line at a time
  - Useful in text processing and especially performing in-place substitution
  - sed can make global substitutions of matched regex patterns with specific text
- Example
  - How change all occurrences of word "the" or "The" to uppercase "THE" in file called file1?

```
sed -r "s/(The|the)/THE/g" file1
```

3

The sed Stream Editor

- Usage:

```
sed -r "s/REGEX/TEXT/g" filename
```

  - Substitutes (replaces) occurrence(s) of REGEX with the given TEXT
  - If filename is omitted, reads from standard input
  - sed has other uses, but most can be emulated with substitutions
  - Resulting output to terminal
    - If wanted permanent changes, need redirect output to new file or make changes in-place using –i option
- Example
  - Replaces all occurrences of 143 with 390 in file2.txt

```
sed -r "s/143/390/g" file2.txt
```

4

The sed Stream Editor

- sed is line-oriented; processes input a line at a time
  - -r option makes *regexes* work better
    - Recognizes ( ) , [ ] , * , + the right way, etc.
  - s for *substitute*
  - g flag after last / asks for a *global match* (replace all)
- Special characters must be escaped to match them literally

```
sed -r "s/http:\/\//https:\/\//g" urls.txt
```

- sed can use delimiters besides / to make more readable

```
sed -r "s#http://#https://#g" urls.txt
```

- Example

```
sed -r "s/([A-Za-z]+), ([A-Za-z]+)/\2 \1/g" names.txt
```

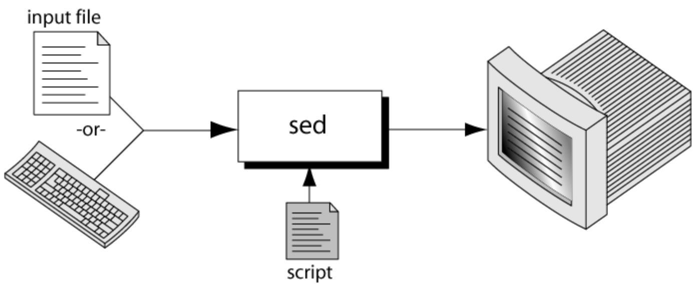
5

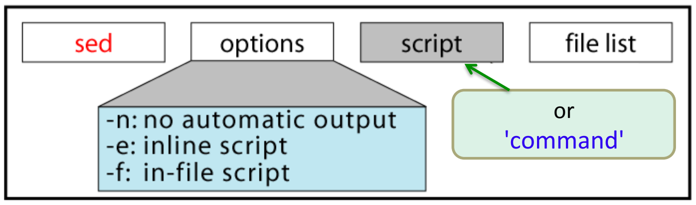
sed Usage

- Edit files too large for interactive editing
- Edit any size files where editing sequence is too complicated to type in interactive mode
- Perform "multiple global" editing functions efficiently in one pass through the input
- Edit multiples files automatically
- Good tool for writing conversion programs

6

The sed Command





sed Syntax

```
sed [-n] [-e] ['command'] [file…]
sed [-n] [-f script] [file…]
```

- Options

–n	only print lines specified with print command (or 'p' flag of substitute ('s') command)
–f script	next argument is filename containing editing commands If first line of script is "#n", acts as if -n had been specified
–e command	next argument is an editing command rather than filename, useful if multiple commands are specified

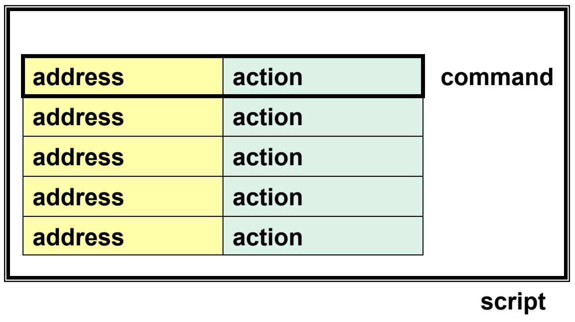
How Does sed Work?

- sed reads line of input
  - Line of input is copied into a temporary buffer called pattern space
  - Editing commands are applied
    - Subsequent commands are applied to line in the pattern space, not the original input line
    - Once finished, line is sent to output (unless –n option was used)
  - Line is removed from pattern space
- sed reads next line of input, until end of file
- Note that input file is unchanged!

9

sed Scripts

- A script is nothing more than a file of commands
- Each command consists of an address and an action, where the address can be a regular expression or line number



10

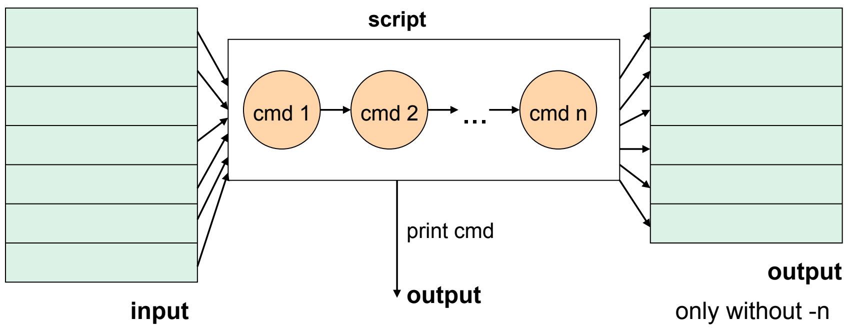
sed Scripts

- As each line of the input file is read, sed reads the first command of the script and checks the address against the current input line:
  - If there is a match, command executed
  - If there is no match, command ignored
  - sed then repeats this action for every command in script file
- When it has reached the end of the script, sed outputs the current line (pattern space) unless the -n option has been set

11

Flow of Control

- sed then reads the next line in the input file and restarts from the beginning of the script file
- All commands in the script file are compared to, and potentially act on, all lines in the input file



12

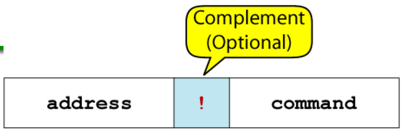
sed Commands

- sed commands have the general form

  `[address[, address]][!]command [arguments]`
- sed copies each input line into a pattern space
  - If address of the command matches line in pattern space, command is applied to that line
  - If command has no address, it is applied to each line as it enters pattern space
  - If a command changes the line in pattern space, subsequent commands operate on the modified line
- When all commands have been read, the line in pattern space is written to standard output and a new line is read into pattern space

13

Addressing



- Address determines which lines in the input file are to be processed by the command(s)
  - Either a line number or a pattern, enclosed in slashes / … /
  - If no address is specified, then command is applied to each input line
- Most commands will accept two addresses
  - If only one address is given, command operates only on that line
  - If two comma separated addresses are given, then command operates on a range of lines between the first and second address, inclusively
- The ! operator can be used to negate an address
  - Command applied to all lines that do NOT match address

14

Commands

- Command is a single letter
- Example:
  - Deletion: d

  `[address1][,address2]d`
- Delete the addressed line(s) from the pattern space
  - Line(s) not passed to standard output
- A new line of input is read and editing resumes with the first command of the script

Delete Address-Command Examples

`d`	deletes all lines
`6d`	deletes line 6
`/^$/d`	deletes all blank lines
`1,10d`	deletes lines 1 through 10
`1,/^$/d`	deletes from line 1 through the first blank line
`/^$/,$d`	deletes from first blank line through last line of file
`/^$/,10d`	deletes from first blank line through line 10
`/^ya*y/,/[0-9]$/d`	deletes from first line that begins with yay, yaay, yaaay, etc. through first line that ends with a digit

Delete Command (D) Example

- Remove Part-time data from "tuition.data" file

```
cat tuition.data
Part-time         1003.99
Two-thirds-time   1506.49
Full-time         2012.29
```

Input data

```
sed -e '/^Part-time/d' tuition.data
Two-thirds-time   1506.49
Full-time         2012.29
```

Output after applying delete command

17

Multiple Commands

- Braces { } used to apply multiple commands to an address

```
[address][,address]{
  command1
  command2
  command3
}
```

- The opening brace { must be the last character on a line
- The closing brace } must be on a line by itself
  - No spaces following the braces
- Alternatively, use ";" after each command:

```
[address][,address]{command1; command2; command3; }
```

- Or:

```
'[address][,address]command1; command2; command3'
```

18

sed Commands

- sed contains many editing commands, though only a few are mentioned here

s	substitute
a	append
i	insert
c	change
d	delete
p	print
r	read
w	write
y	transform
=	display line number
N	append next line to current one
q	quit

19

Print

- Print command (p) used to force pattern space to be output, useful if -n option has been specified
- Syntax:

  `[address1[,address2]]p`

  – Note: if -n or #n option has not been specified, p will cause the line to be output twice!
- Examples:

`1,5p`	will display lines 1 through 5
`/^$/,$p`	will display lines from first blank line through last line of file

20

Substitute

- Syntax:

  `[address(es)]s/pattern/replacement/[flags]`

  - pattern : search pattern
  - replacement : replacement string for pattern
  - flags : optionally any of the following

`n`	a number from 1 to 512 indicating which occurrence of pattern should be replaced
`g`	global, replace all occurrences of pattern in pattern space
`p`	print contents of pattern space

21

Substitute Examples

`s/Puff Daddy/P. Diddy/`

- Substitute P. Diddy for the first occurrence of Puff Daddy in pattern space

`s/Four/Five/2`

- Substitutes Five for the second occurrence of Four in the pattern space (i.e., each line)

`s/paper/plastic/p`

- Substitutes plastic for the first occurrence of paper and outputs (prints) pattern space

22

Replacement Patterns

- Substitute can use several special characters in the replacement string

`&`	replaced by entire string matched in regular expression for pattern
`\n`	replaced by nth substring (or sub-expression) previously specified using "\(" and "\)"
`\`	used to escape the ampersand (&) and the backslash (\)

23

Replacement Pattern Examples

"the UNIX operating system …"

```
s/.NI./wonderful &/
```

--> "the wonderful UNIX operating system …"

"unix is fun"

```
sed 's/\([[:alpha:]]\)\([^ \n]*\)/\2\1ay/g'
```

--> "nixuay siay unfay"

```
cat file3
```

first:second
one:two

```
sed 's/\(.*\):\(.*\)/\2:\1/' file3
```

second:first
two:one

24

Append, Insert, and Change

- Syntax for these commands is little strange because they must be specified on multiple lines
- Append
  ```
  [address]a\
  text
  ```
- Insert
  ```
  [address]i\
  text
  ```
- Change
  ```
  [address(es)]c\
  text
  ```

Append (a) and Insert (i) for single lines only, not range

25

Append Command (A) Example

```
cat tuition.append.sed
a \
---------------------------
```

sed script to append dashed line after each input line

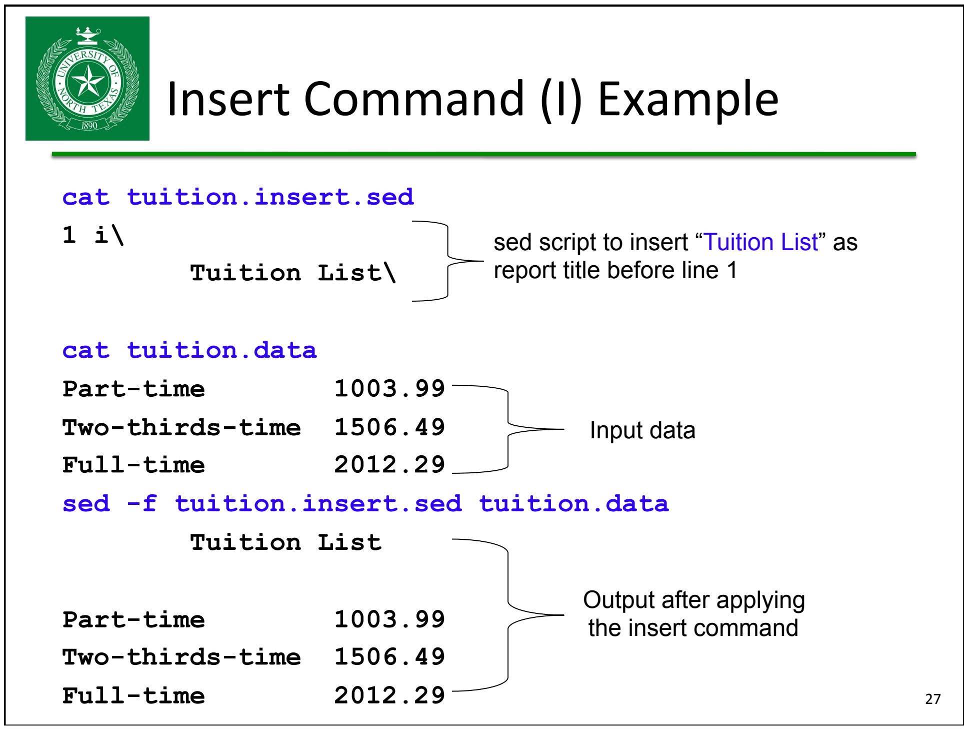
```
cat tuition.data
Part-time         1003.99
Two-thirds-time   1506.49
Full-time         2012.29
```

Input data

```
sed -f tuition.append.sed tuition.data
Part-time         1003.99
---------------------------
Two-thirds-time   1506.49
---------------------------
Full-time         2012.29
---------------------------
```

Output after applying the append command

26



Insert Command (I) Example

```
cat tuition.insert.sed
1 i\
        Tuition List\
```

sed script to insert “Tuition List” as report title before line 1

```
cat tuition.data
Part-time         1003.99
Two-thirds-time   1506.49
Full-time         2012.29
```

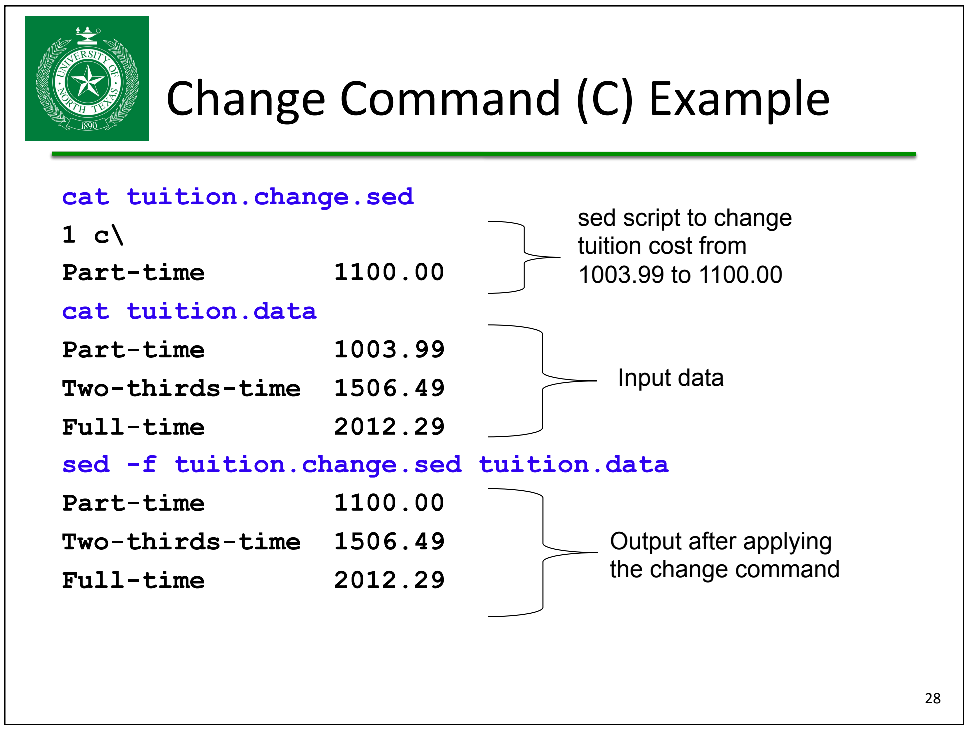
Input data

```
sed -f tuition.insert.sed tuition.data
        Tuition List

Part-time         1003.99
Two-thirds-time   1506.49
Full-time         2012.29
```

Output after applying the insert command

27



Change Command (C) Example

```
cat tuition.change.sed
1 c\
Part-time         1100.00
```

sed script to change tuition cost from 1003.99 to 1100.00

```
cat tuition.data
Part-time         1003.99
Two-thirds-time   1506.49
Full-time         2012.29
```

Input data

```
sed -f tuition.change.sed tuition.data
Part-time         1100.00
Two-thirds-time   1506.49
Full-time         2012.29
```

Output after applying the change command

28

Complement (!) Operator

- If an address is followed by exclamation point (!), associated command is applied to all lines that don't match address or address range
- Examples:

“The brown cow" --> "The brown horse"
"The black cow" --> "The black cow"

`/black/!s/cow/horse/`

substitute horse for cow on all lines except those that contained black

`1,5!d` delete all lines except 1 through 5

- Print lines that do not contain “obsolete”

```
sed -e '/obsolete/!p' input-file
```

29

Read and Write File Commands

- Syntax: `r filename`
  - Queue contents of filename to be read and inserted into output stream at end of current cycle, or when next input line is read
    - If filename cannot be read, treated as if were an empty file, without any error indication
- Syntax: `w filename`
  - Write the pattern space to filename
  - The filename will be created (or truncated) before the first input line is read
  - All w commands which refer to the same filename are output through the same FILE stream

30

Read and Write File Commands

```
cat tmp
one two three
one three five
two four six
sed 'r tmp'
My first line of input          ---> not read until the first line is taken from the input
My first line of input
one two three
one three five
two four six
My next line
My next line^D
sed 'w tmp1'
hello 1
hello 1
hello 2
hello 2
hello 3
hello 3^D
cat tmp1
hello 1
hello 2
hello 3
```

31

Line Number

- Line number command (=) writes the current line number before each matched/output line
- Examples:

```
sed -e '/Two-thirds-time/=' tuition.data
sed -e '/^[0-9][0-9]/=' inventory
```

```
sed '=' tmp1          sed -n '=' tmp1
1                     1
hello1                2
2                     3
hello2
3
hello3
```

32

Transform

- Transform command (y) operates like tr, doing a one-to-one or character-to-character replacement
  - Accepts zero, one or two addresses

```
[address[,address]]y/abc/xyz/
```

  - Every a within the specified address(es) is transformed to an x, b to y and c to z
- Examples

```
y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
```

  - Changes all lower case characters on addressed line to upper case

```
sed -e '1,10y/abcd/wxyz/' datafile
```

  - Must have same number of characters

33

Quit

- Syntax: `[addr]q`
  - Quit (exit sed) when addr is encountered
- It takes at most a single line address
  - Once a line matching the address is reached, script will be terminated
  - Can be used to save time when you only want to process some portion of the beginning of a file
- Example
  - To print the first 100 lines of a file (like head)

```
sed '100q' filename
```

  - sed will, by default, send the first 100 lines of filename to standard output and then quit processing

34

CSE

The gawk Programming Language

The gawk Programming Language

- A scripting language used for manipulating data and generating reports
  - Geared towards working with delimited fields on a line-by-line basis
- Summary of gawk operations
  - Scans a file line-by-line
  - Splits each input line into fields
  - Compares each input line and fields to the specified pattern
  - Performs the requested action(s) on lines matching the specified pattern

Structure of a gawk Program

- A gawk program consists of:
  - An optional BEGIN segment
    - For processing to execute prior to reading input
  - Pattern – Action pairs
    - Processing for input data
    - For each pattern matched, the corresponding action is taken
  - An optional END segment
    - Processing after end of input data

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

37

Running a gawk Program

- There are several ways to run a gawk program

  `gawk 'program' input_file(s)`
  - Program and input files are provided as command-line arguments

  `gawk 'program'`
  - Program is a command-line argument
  - Input is taken from standard input

  `gawk -f program_file input_files`
  - Program is read from a file

38

Patterns and Actions

- Search a set of files for patterns
- Perform specified actions upon lines or fields that contain instances of patterns
- Does not alter input files
- Process one input line at a time
- This is similar to sed

39

Pattern-Action Structure

- Every program statement has to have a pattern or an action or both
- Default pattern is to match all lines
- Default action is to print current record
- Patterns are simply listed; actions are enclosed in { }
  - Some actions can be similar to C code
- gawk scans a sequence of input lines, or records, one by one, searching for lines that match the pattern
  - Meaning of match depends on the pattern

40

Patterns

- Selector that determines whether action is to be executed
- Pattern can be:
  - Special token BEGIN or END
  - Regular expression (enclosed with / /)
  - Relational or string match expression
  - ! negates the match
  - Arbitrary combination of the above using && and/or ||
    - `/UNT/` matches if the string “UNT” is in the record
    - `x > 0` matches if the condition is true
    - `/UNT/ && (name == "UNIX Tools")`

41

BEGIN and END Patterns

- BEGIN and END provide a way to gain control before and after processing, for initialization, and wrap-up

BEGIN
- Actions are performed before first input line is read

END
- Actions are done after the last input line has been processed.

42

Actions

- Action may include a list of one or more C like statements, as well as arithmetic and string expressions and assignments and multiple output streams
- Action performed on every line that matches pattern
  - If pattern not provided, action performed on every input line
  - If action not provided, all matching lines sent to standard output
- Since patterns and actions are optional, actions must be enclosed in braces to distinguish them from pattern

43

Introductory Example

```
ls | gawk '
  BEGIN { print "List of html files:" }
  /\.html$/ { print }
  END { print "There you go!" }
  '
```

```
List of html files:
index.html
as1.html
as2.html
There you go!
```

44

Variables

- gawk scripts can define and use variables

```
BEGIN { sum = 0 }
{ sum++ }
END { print sum }
```

- Some variables are predefined

Basic gawk Terminology

- gawk supports two types of buffers
  - Field
    - A unit of data in a line separated from other fields by the field separator
  - Record
    - A collection of fields in a line (file made up of records)
- Default field separator is whitespace
- Namespace for fields in current record: $1, $2, etc.
  - The $0 variable contains the entire record (i.e., line)
- Example
  - Given line of input: "This class is fun!"
  - $1 = "This", $2 = "class", etc.

Records

- Default record separator is newline
  - By default, gawk processes its input one line at a time
- Could be any other *regular expression*
- RS: record separator
  - Can be changed in BEGIN action
- NR is the variable whose value is the number of the current record.

Fields

- Each input line is split into fields
  - FS: field separator
    - Default is whitespace (1 or more spaces or tabs)
  - **`gawk -Fc`** option sets FS to the character c
    - Can also be changed in BEGIN
  - $0 is the entire line
  - $1 is the first field, $2 is the second field, ….
- Only fields begin with $, variables do not

Some gawk System Variables

- gawk supports number of system variables

– FS	Field separator (default = space)
– 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
– ARGC/ARGV	Get arguments from command line

Simple Output from gawk

- Printing every line
  - If action has no pattern, action is performed to all input lines
    - `{ print }` prints all input lines to standard out
    - `{ print $0 }` will do the same thing
- Printing certain fields
  - Multiple items can be printed on the same output line with a single print statement
  - `{ print $1, $3 }`
  - Expressions separated by a comma are, by default, separated by a single space when printed (OFS)

More Output from gawk

- NF, the Number of Fields
  - Any valid expression can be used after a $ to indicate the contents of a particular field
  - One built-in expression is NF, or Number of Fields
  - `{ print NF, $1, $NF }` will print number of fields, first field, and last field in the current record
  - `{ print $(NF-2) }` prints the third to last field
- Computing and printing
  - You can also do computations on the field values and include the results in your output
  - `{ print $1, $2 * $3 }`

51

More Output from gawk

- Printing line numbers
  - The built-in variable NR can be used to print line numbers
  - `{ print NR, $0 }` prints each line prefixed with its line number
- Putting text in the output
  - You can also add other text to the output besides what is in the current record
  - `{ print "total pay for", $1, "is", $2 * $3 }`
  - Note that the inserted text needs to be surrounded by double quotes

52

Formatted Output from gawk

- Lining up fields
  - Like C, gawk has a printf function for producing formatted output
  - printf has the form printf(format, val1, val2, val3, …)

```
{ printf ("total pay for %s is $%.2f\n", $1, $2 * $3) }
```

  - When using printf, formatting is under your control so no automatic spaces or newlines are provided by gawk
  - You have to insert them yourself.

```
{ printf ("%-8s %6.2f\n", $1, $2 * $3 ) }
```

Selection

- gawk patterns are good for selecting specific lines from the input for further processing
  - Selection by comparison

    ```
    $2 >= 5 { print }
    ```
  - Selection by computation

    ```
    $2 * $3 > 50 { printf ("%6.2f for %s\n", $2 * $3, $1) }
    ```
  - Selection by text content

    ```
    $1 == "UNT"
    $2 ~ /UNT/
    ```
  - Combinations of patterns

    ```
    $2 >= 4 || $3 >= 20
    ```
  - Selection by line number

    ```
    NR >= 10 && NR <= 20
    ```

Arithmetic and Variables

- gawk variables take on numeric (floating point) or string values according to context
- User-defined variables are unadorned (i.e., they do not need to be declared)
- By default, user-defined variables are initialized to the null string which has numerical value 0

55

Computing with gawk

- Counting is easy to do with gawk

```
$3 > 15 { emp = emp + 1}
END { print emp, "employees worked
         more than 15 hrs"}
```

- Computing sums and averages is also simple

```
{ pay = pay + $2 * $3 }
END { print NR, "employees"
      print "total pay is", pay
      print "average pay is", pay/NR
    }
```

56

Handling Text

- One major advantage of gawk is its ability to handle strings as easily as many languages handle numbers
- gawk variables can hold strings of characters as well as numbers, and gawk conveniently translates back and forth as needed
- This program finds the employee who is paid the most per hour:

```
# Fields: employee, payrate
$2 > maxrate { maxrate = $2; maxemp = $1 }
END { print "highest hourly rate:",
       maxrate, "for", maxemp }
```

57

String Manipulation

- String Concatenation
  - New strings can be created by combining old ones

```
    { names = names $1 " " }
END { print names }
```

- Printing the Last Input Line
  - Although NR retains its value after the last input line has been read, $0 does not

```
    { last = $0 }
END { print last }
```

58

Built-In Functions

- gawk contains a number of built-in functions: length is one of them
- Counting lines, words, and characters using length (similar to wc)

```
      { nc = nc + length($0) + 1
        nw = nw + NF
      }
  END { print NR, "lines,", nw, "words,", nc,
          "characters" }
```

- substr(s, m, n) produces substring of s that begins at position m and is at most n characters long

Control Flow Statements

- gawk provides several control flow statements for making decisions and writing loops
- If-Then-Else

```
      $2 > 6 { n = n + 1; pay = pay + $2 * $3 }

  END { if (n > 0)
            print n, "employees, total pay is",
              pay, "average pay is", pay/n
        else
            print "no employees are paid more
              than $6/hour"
      }
```

Loop Control

- While

```
# interest1 - compute compound interest
#   input: amount, rate, years
#   output: compound value at end of each year
{  i = 1
   while (i <= $3) {
       printf("\t%.2f\n", $1 * (1 + $2) ^ i)
       i = i + 1
   }
}
```

Do-While Loops

```
do {
      statement1
   }
while (expression)
```

61

for Statements

- For

```
# interest2 - compute compound interest
#   input: amount, rate, years
#   output: compound value at end of each year

{ for (i = 1; i <= $3; i = i + 1)
    printf("\t%.2f\n", $1 * (1 + $2) ^ i)
}
```

62

Arrays

- Array elements are not declared
- Array subscripts can have any value:
  - Numbers
  - Strings (associative arrays)
- Examples

```
arr[3]="value"
grade["Smith"]=40.3
```

63

Array Example

```
# reverse - print input in reverse order by line

{ line[NR] = $0 }     # remember each line

END {
      for (i=NR; (i > 0); i=i-1) {
          print line[i]
      }
    }
```

- Use for loop to read associative array

```
for (v in array) { … }
```

  - Assigns to v each subscript of array (unordered)
  - Element is `array[v]`

64

Operators

`=` assignment operator
- Sets a variable equal to a value or string

`==` equality operator
- Returns TRUE is both sides are equal

`!=` inverse equality operator

`&&` logical AND

`||` logical OR

`!` logical NOT

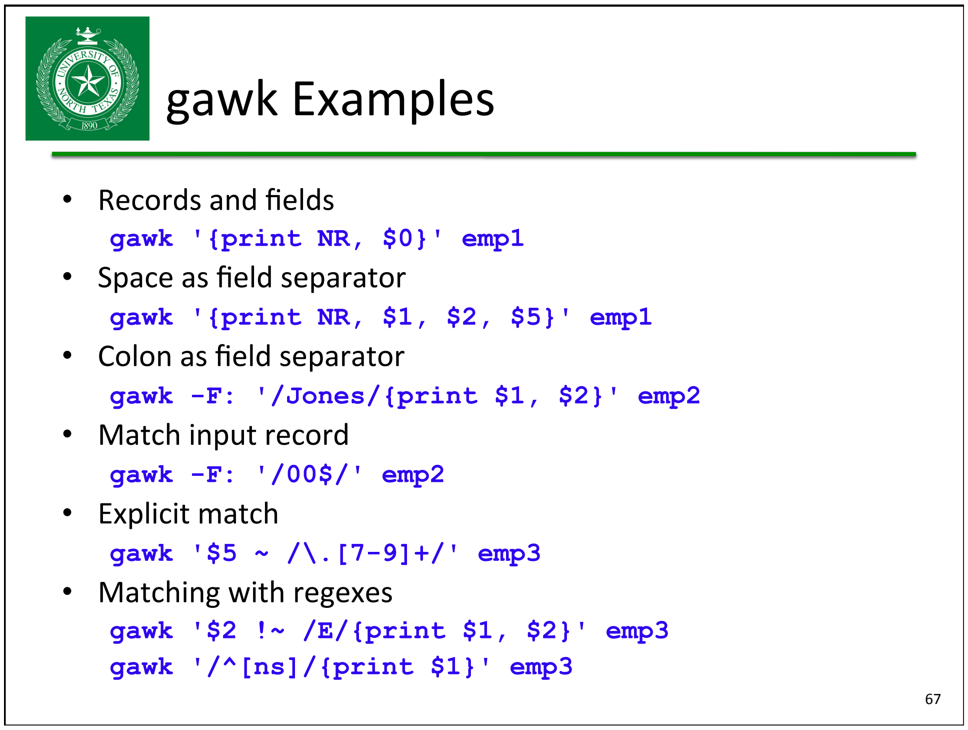
`<`, `>`, `<=`, `>=` relational operators

`+`, `-`, `/`, `*`, `%`, `^`

~ and !~ are used to perform regex comparisons, as in exp ~ /regexp/

Built-In Functions

- Arithmetic
  - sin, cos, atan, exp, int, log, rand, sqrt
- String
  - length, substr, split
- Output
  - print, printf
- Special
  - system - executes a Unix/Linux command
    - `system("clear")` to clear the screen
    - Note double quotes around the Unix command
  - exit - stop reading input and go immediately to the END pattern-action pair if it exists, otherwise exit the script



gawk Examples

- Records and fields
```
gawk '{print NR, $0}' emp1
```
- Space as field separator
```
gawk '{print NR, $1, $2, $5}' emp1
```
- Colon as field separator
```
gawk -F: '/Jones/{print $1, $2}' emp2
```
- Match input record
```
gawk -F: '/00$/' emp2
```
- Explicit match
```
gawk '$5 ~ /\.[7-9]+/' emp3
```
- Matching with regexes
```
gawk '$2 !~ /E/{print $1, $2}' emp3
gawk '/^[ns]/{print $1}' emp3
```

67