Lecture 4

Regular Expressions:

grep, **sed** and **awk**

Previously

- Basic UNIX Commands
  - Files: rm, cp, mv, ls
  - Processes: ps, kill
- Unix Filters
  - cat, head, tail, tee, wc
  - cut, paste
  - find
  - sort, uniq

Today

- Regular Expressions
  - Allow you to search for text in files
  - **grep** command
- Stream manipulation:
  - **sed**
  - **awk?**
- But first, one command we didn’t cover last time…

tr: TRanslate Characters

- Copies standard input to standard output with substitution or deletion of selected characters
- Syntax: *tr [-cds] [string1] [string2]*
  - `-d` delete all input characters contained in *string1*
  - `-c` complements the characters in *string1* with respect to the entire ASCII character set
  - `-s` squeeze all strings of repeated output characters that are in *string2* to single characters

tr (continued)

- *tr* reads from standard input.
  - Any character that does not match a character in *string1* is passed to *standard output* unchanged
  - Any character that does match a character in *string1* is translated into the corresponding character in *string2* and then passed to *standard output*
- Examples
  - *tr s z* replaces all instances of *s* with z
  - *tr so zx* replaces all instances of *s* with *z* and *o* with *x*
  - *tr a-z A-Z* replaces all lower case characters with upper case characters
  - *tr –d a-c* deletes all a-c characters

tr uses

- Change delimiter

```
tr ‘|’ ‘:’
```

- Rewrite numbers

```
tr ,. .,
```

- Import DOS files

```
tr –d ’\r’ < dos_file
```

- Find ASCII in a binary file

```
tr –cd ’\n[a-zA-Z0-9 ]’ < binary_file
```

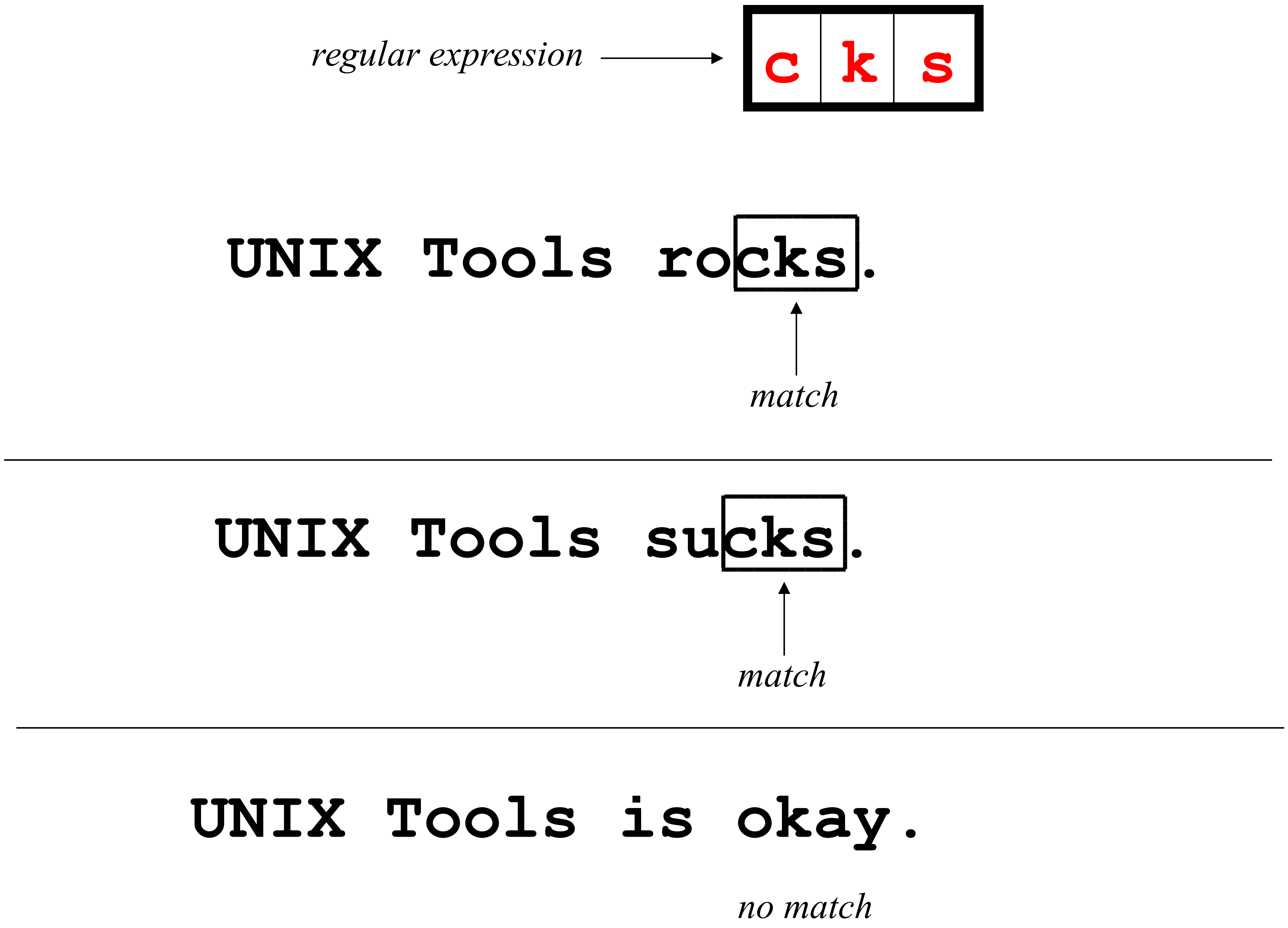
Regular Expressions

What Is a Regular Expression?

- A regular expression (*regex*) describes a set of possible input strings.
- *Regular expressions* descend from a fundamental concept in Computer Science called *finite automata* theory
- *Regular expressions* are endemic to Unix
  - **vi**, **ed**, **sed**, and **emacs**
  - **awk**, **tcl**, **perl** and **Python**
  - **grep**, **egrep**, **fgrep**
  - **compilers**

Regular Expressions

- The simplest regular expressions are a string of literal characters to match.
- The string ***matches*** the regular expression if it contains the substring.



regular expression

c k s

UNIX Tools rocks.

match

UNIX Tools sucks.

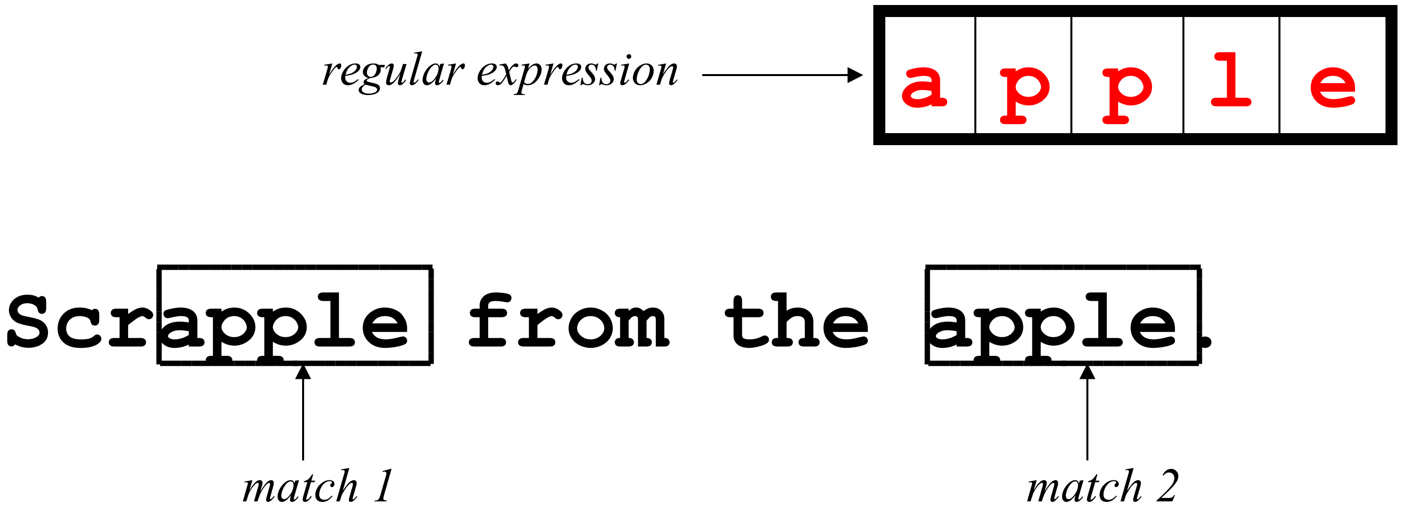
match

UNIX Tools is okay.

no match

Regular Expressions

- A regular expression can match a string in more than one place.



regular expression

`a p p l e`

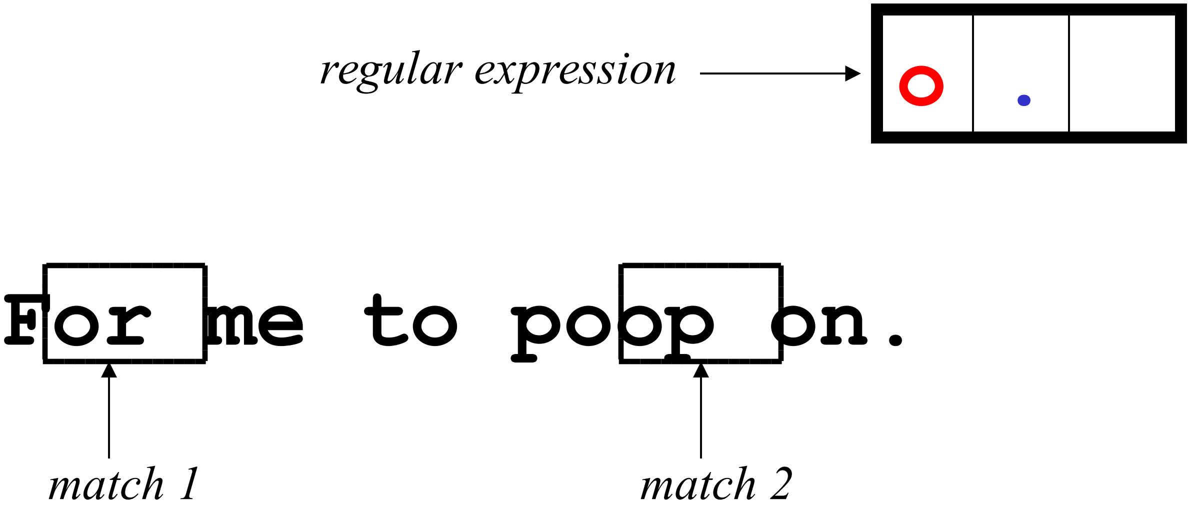
`Scrapple from the apple.`

match 1

match 2

Regular Expressions

- The . regular expression can be used to match any character.



regular expression

o .

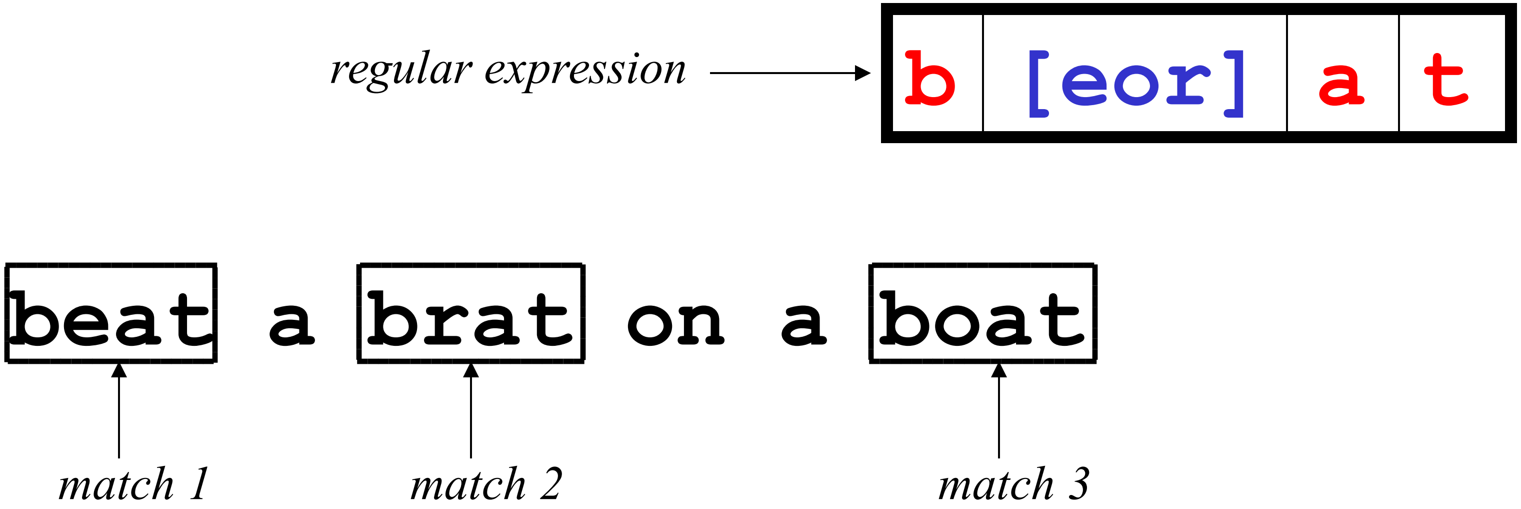
`For me to poop on.`

match 1

match 2

Character Classes

- Character classes `[]` can be used to match any specific set of characters.



regular expression

`b [eor] a t`

`beat a brat on a boat`

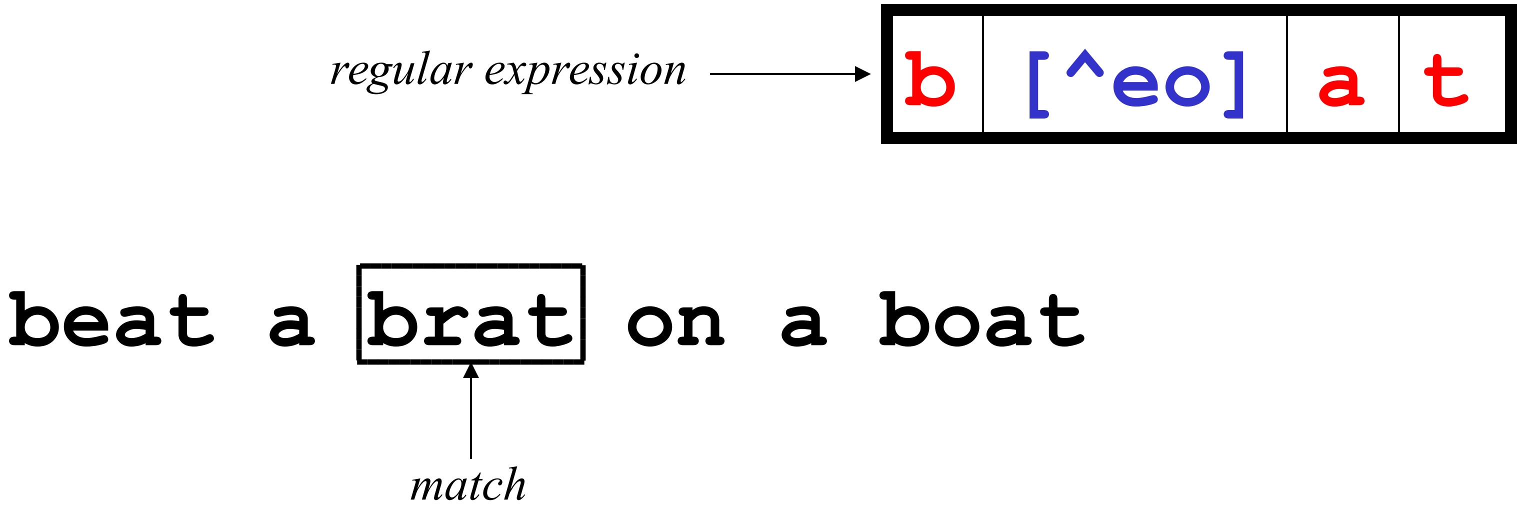
match 1

match 2

match 3

Negated Character Classes

- Character classes can be negated with the `[^]` syntax.



regular expression → `b [^eo] a t`

`beat a brat on a boat`

match

More About Character Classes

  - `[aeiou]` will match any of the characters `a`, `e`, `i`, `o`, or `u`
  - `[kK]orn` will match `korn` or `Korn`

- Ranges can also be specified in character classes
  - `[1-9]` is the same as `[123456789]`
  - `[abcde]` is equivalent to `[a-e]`
  - You can also combine multiple ranges
    - `[abcde123456789]` is equivalent to `[a-e1-9]`
  - Note that the `-` character has a special meaning in a character class ***but only*** if it is used within a range, `[-123]` would match the characters `-`, `1`, `2`, or `3`

Named Character Classes

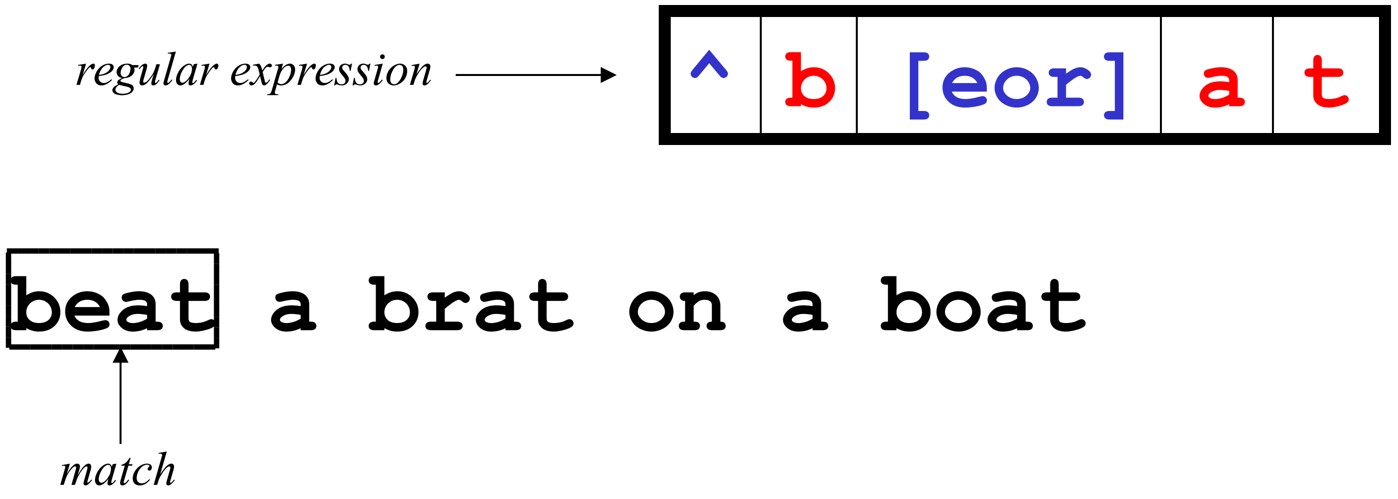
- Commonly used character classes can be referred to by name (*alpha*, *lower*, *upper*, *alnum*, *digit*, *punct*, *cntrl*)
- Syntax **[:*name*:]**

`[a-zA-Z]`	`[[:alpha:]]`
`[a-zA-Z0-9]`	`[[:alnum:]]`
`[45a-z]`	`[45[:lower:]]`

- Important for portability across languages

Anchors

- Anchors are used to match at the beginning or end of a line (or both).
- `^` means beginning of the line
- `$` means end of the line

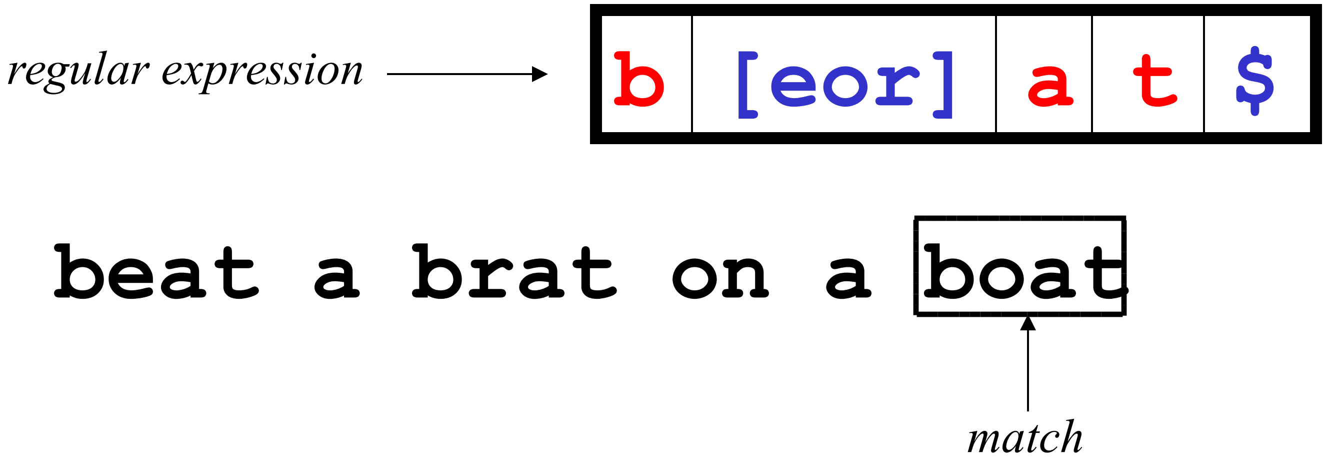


regular expression

^	b	[eor]	a	t

beat a brat on a boat

match



regular expression

b	[eor]	a	t	$

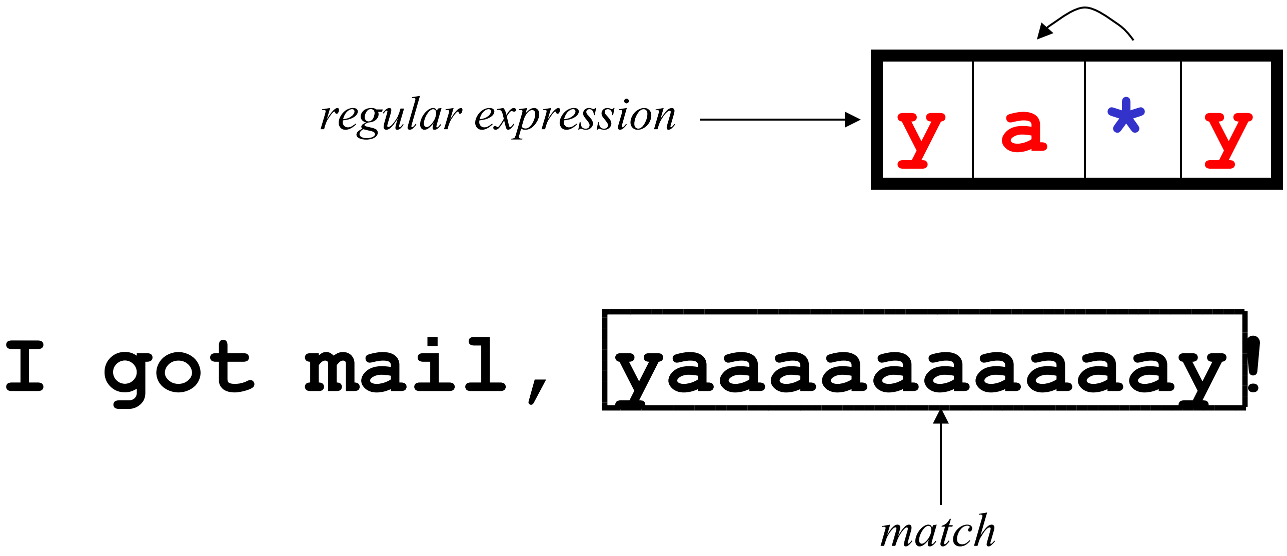
beat a brat on a boat

match

`^word$`   `^$`

Repetition

- The * is used to define **zero or more** occurrences of the *single* regular expression preceding it.

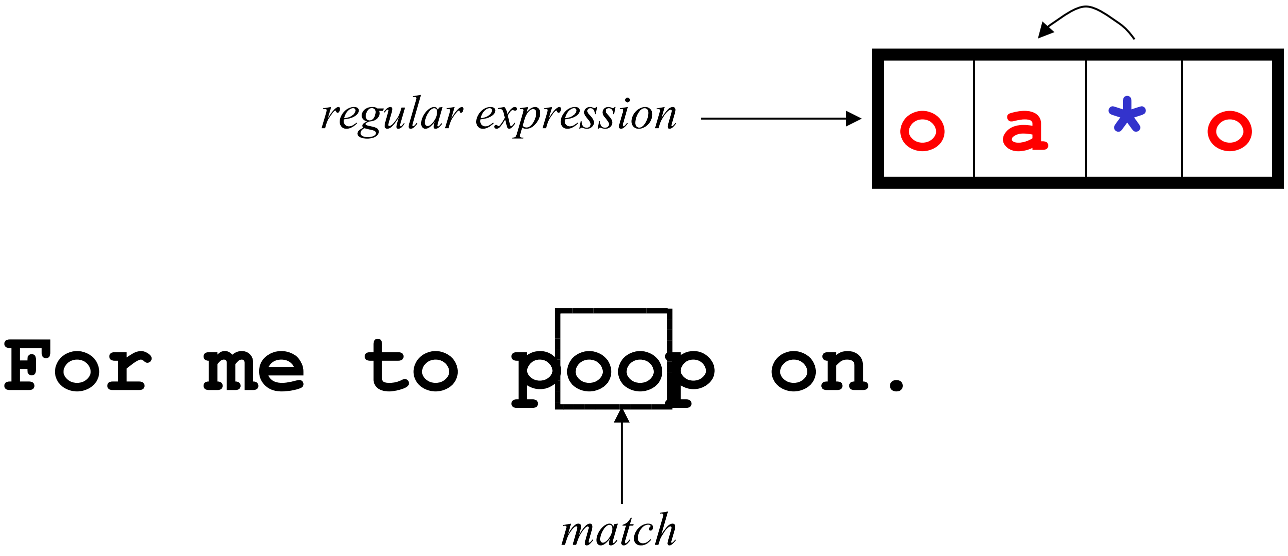


regular expression

`y a * y`

`I got mail, yaaaaaaaaaay!`

match



regular expression

`o a * o`

`For me to poop on.`

match

`.*`

Repetition Ranges

- Ranges can also be specified
  - `{`*n*`,`*m*`}` notation can specify a range of repetitions for the immediately preceding regex
  - `{`*n*`}` means exactly *n* occurrences
  - `{`*n*`,}` means at least *n* occurrences
  - `{`*n*`,`*m*`}` means at least *n* occurrences but no more than *m* occurrences
- Example:
  - `.{0,}` same as `.*`
  - `a{2,}` same as `aaa*`

Subexpressions

- If you want to group part of an expression so that `*` applies to more than just the previous character, use `( )` notation
- Subexpresssions are treated like a single character
  - `a*` matches 0 or more occurrences of `a`
  - `abc*` matches `ab`, `abc`, `abcc`, `abccc`, …
  - `(abc)*` matches `abc`, `abcabc`, `abcabcabc`, …
  - `(abc){2,3}` matches `abcabc` or `abcabcabc`

grep

- grep comes from the **ed** (Unix text editor) search command “**g**lobal **r**egular **e**xpression **p**rint” or g/re/p
- This was such a useful command that it was written as a standalone utility
- There are two other variants, ***egrep*** and ***fgrep*** that comprise the *grep* family
- *grep* is the answer to the moments where you know you want the file that contains a specific phrase but you can’t remember its name

Family Differences

- **grep** - uses regular expressions for pattern matching
- **fgrep** - file grep, does not use regular expressions, only matches fixed strings but can get search strings from a file
- **egrep** - extended grep, uses a more powerful set of regular expressions but does not support backreferencing, generally the fastest member of the grep family
- **agrep** – approximate grep; not standard

Syntax

- Regular expression concepts we have seen so far are common to **grep** and **egrep**.
- grep and egrep have different syntax
  - **grep**: BREs
  - **egrep**: EREs
- Major syntax differences:
  - **grep**: `\(` and `\)`, `\{` and `\}`
  - **egrep**: `(` and `)`, `{` and `}`

Protecting Regex Metacharacters

- Since many of the special characters used in regexs also have special meaning to the shell, it’s a good idea to get in the habit of single quoting your regexs
  - This will protect any special characters from being operated on by the shell
  - If you habitually do it, you won’t have to worry about when it is necessary

Escaping Special Characters

- Even though we are single quoting our regexs so the shell won’t interpret the special characters, sometimes we still want to use an operator as itself
- To do this, we “escape” the character with a \ (backslash)
- Suppose we want to search for the character sequence ‘a*b*’
  - Unless we do something special, this will match zero or more ‘a’s followed by zero or more ‘b’s, *not what we want*
  - **‘a*b*’** will fix this - now the asterisks are treated as regular characters

Egrep: Alternation

- Regex also provides an alternation character `|` for matching one or another subexpression
  - `(T|Fl)an` will match ‘Tan’ or ‘Flan’
  - `^(From|Subject):` will match the From and Subject lines of a typical email message
    - It matches a beginning of line followed by either the characters ‘From’ or ‘Subject’ followed by a ‘:’
- Subexpressions are used to limit the scope of the alternation
  - `At(ten|nine)tion` then matches “Attention” or “Atnination”, not “Atten” or “ninetion” as would happen without the parenthesis - `Atten|ninetion`

Egrep: Repetition Shorthands

- The `*` (star) has already been seen to specify zero or more occurrences of the immediately preceding character
- `+` (plus) means “one or more”
  - `abc+d` will match ‘abcd’, ‘abccd’, or ‘abccccccd’ but will not match ‘abd’
  - Equivalent to `{1,}`

Egrep: Repetition Shorthands cont

- The ‘`?`’ (question mark) specifies an optional character, the single character that immediately precedes it
  - `July?` will match ‘Jul’ or ‘July’
  - Equivalent to `{0,1}`
  - Also equivalent to `(Jul|July)`
- The `*`, `?`, and `+` are known as *quantifiers* because they specify the quantity of a match
- Quantifiers can also be used with subexpressions
  - `(a*c)+` will match ‘c’, ‘ac’, ‘aac’ or ‘aacaacac’ but will not match ‘a’ or a blank line

Grep: Backreferences

- Sometimes it is handy to be able to refer to a match that was made earlier in a regex
- This is done using *backreferences*
  - **`\`*n*** is the backreference specifier, where *n* is a number
- For example, to find if the first word of a line is the same as the last:
  - **`^\([[:alpha:]]\{1,\}\).*\1$`**
  - The **`\([[:alpha:]]\{1,\}\)`** matches 1 or more letters

Practical Regex Examples

- Variable names in C
  - **`[a-zA-Z_][a-zA-Z_0-9]*`**
- Dollar amount with optional cents
  - **`\$[0-9]+(\.[0-9][0-9])?`**
- Time of day
  - **`(1[012]|[1-9]):[0-5][0-9] (am|pm)`**
- HTML headers <h1> <H1> <h2> …
  - **`<[hH][1-4]>`**

grep Family

- Syntax

  grep [-hilnv] [-e expression] [filename]

  egrep [-hilnv] [-e expression] [-f filename] [expression] [filename]

  fgrep [-hilnxv] [-e string] [-f filename] [string] [filename]

  - **-h** Do not display filenames
  - **-i** Ignore case
  - **-l** List only filenames containing matching lines
  - **-n** Precede each matching line with its line number
  - **-v** Negate matches
  - **-x** Match whole line only (*fgrep* only)
  - **-e** *expression* Specify expression as option
  - **-f** *filename* Take the regular expression (egrep) or a list of strings (fgrep) from *filename*

grep Examples

  - `grep 'men' GrepMe`
  - `grep 'fo*' GrepMe`
  - `egrep 'fo+' GrepMe`
  - `egrep -n '[Tt]he' GrepMe`
  - `fgrep 'The' GrepMe`
  - `egrep 'NC+[0-9]*A?' GrepMe`
  - `fgrep -f expfile GrepMe`
- Find all lines with signed numbers

```
$ egrep ’[-+][0-9]+\.?[0-9]*’ *.c
bsearch. c: return -1;
compile. c: strchr("+1-2*3", t-> op)[1] - ’0’, dst,
convert. c: Print integers in a given base 2-16 (default 10)
convert. c: sscanf( argv[ i+1], "% d", &base);
strcmp. c: return -1;
strcmp. c: return +1;
```

- **egrep** has its limits: For example, it cannot match all lines that contain a number divisible by 7.

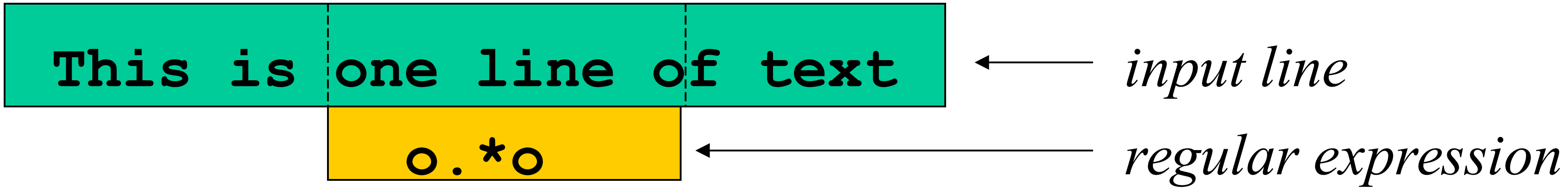
Fun with the Dictionary

- `/usr/dict/words` contains about 25,000 words
  - `egrep hh /usr/dict/words`
    - beachhead
    - highhanded
    - withheld
    - withhold
- **egrep** as a simple spelling checker: Specify plausible alternatives you know

  ```
  egrep "n(ie|ei)ther" /usr/dict/words
  neither
  ```
- How many words have 3 a’s one letter apart?
  - `egrep a.a.a /usr/dict/words | wc –l`
    - 54
  - `egrep u.u.u /usr/dict/words`
    - cumulus

Other Notes

- Use **`/dev/null`** as an extra file name
  - Will print the name of the file that matched
    - **`grep test bigfile`**
      - **`This is a test.`**
    - **`grep test /dev/null bigfile`**
      - **`bigfile:This is a test.`**
- Return code of grep is useful
  - **`grep fred filename > /dev/null && rm filename`**



This is one line of text

o.*o

input line

regular expression

x xyz	Ordinary characters match themselves (NEWLINES and metacharacters excluded) Ordinary strings match themselves	*fgrep, grep, egrep*
\m	Matches literal character *m*	*grep, egrep*
^	Start of line	
$	End of line	
.	Any single character	
[xy^$x]	Any of x, y, ^, $, or z	
[^xy^$z]	Any one character other than x, y, ^, $, or z	
[a-z]	Any single character in given range	
r*	zero or more occurrences of regex r	
r1r2	Matches r1 followed by r2	
\(r\)	Tagged regular expression, matches r	*grep*
\n	Set to what matched the *n*th tagged expression (n = 1-9)	
\{n,m\}	Repetition	
r+	One or more occurrences of r	*egrep*
r?	Zero or one occurrences of r	
r1|r2	Either r1 or r2	
(r1|r2)r3	Either r1r3 or r2r3	
(r1|r2)*	Zero or more occurrences of r1|r2, e.g., r1, r1r1, r2r1, r1r1r2r1,…)	
{n,m}	Repetition	

Quick Reference

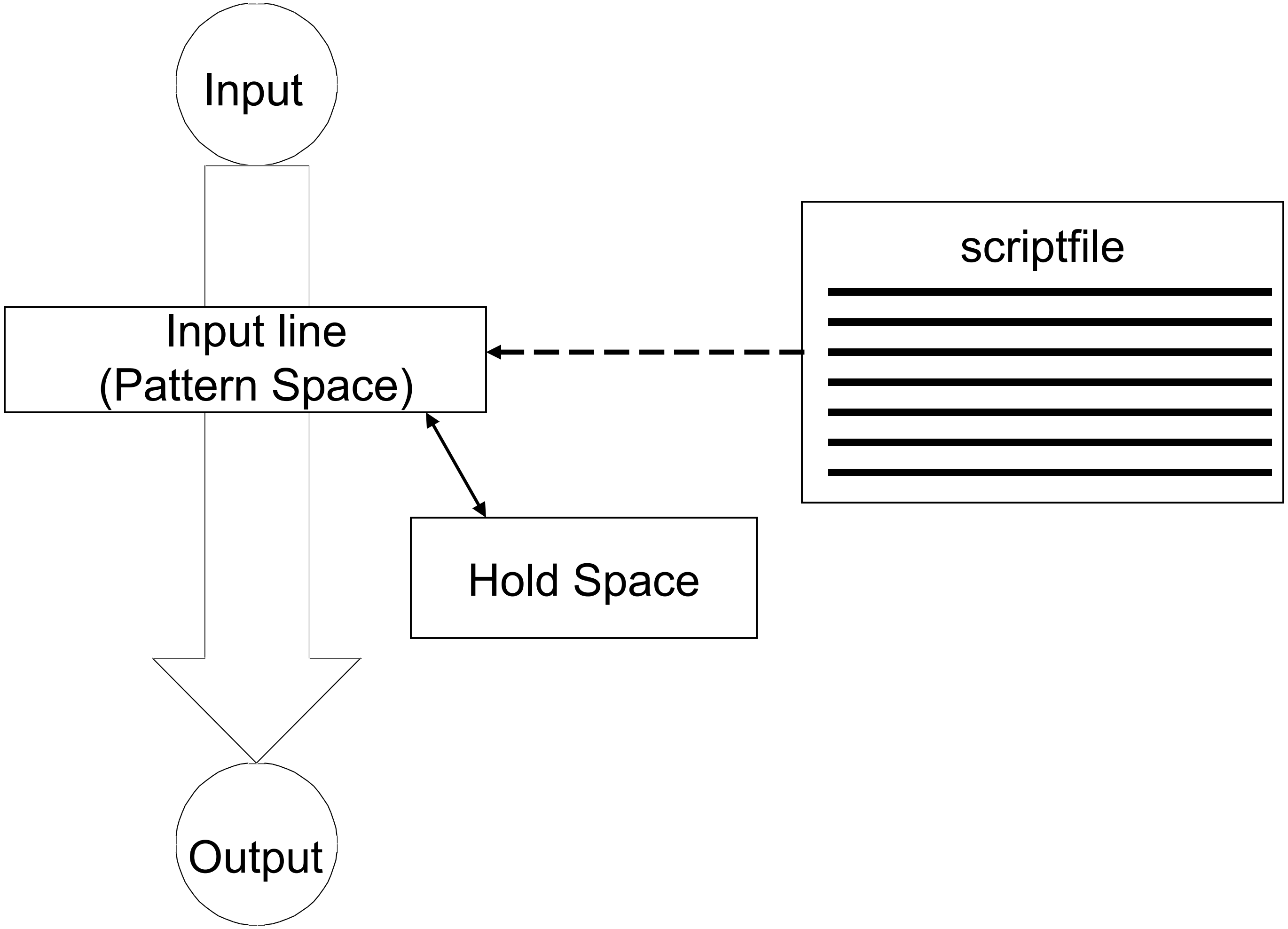
Sed: Stream-oriented, Non-Interactive, Text Editor

- Look for patterns one line at a time, like **grep**
- *Change* lines of the file
- Non-interactive text editor
  - Editing commands come in as *script*
  - There is an interactive editor *ed* which accepts the same commands
- A Unix filter
  - Superset of previously mentioned tools

Conceptual overview

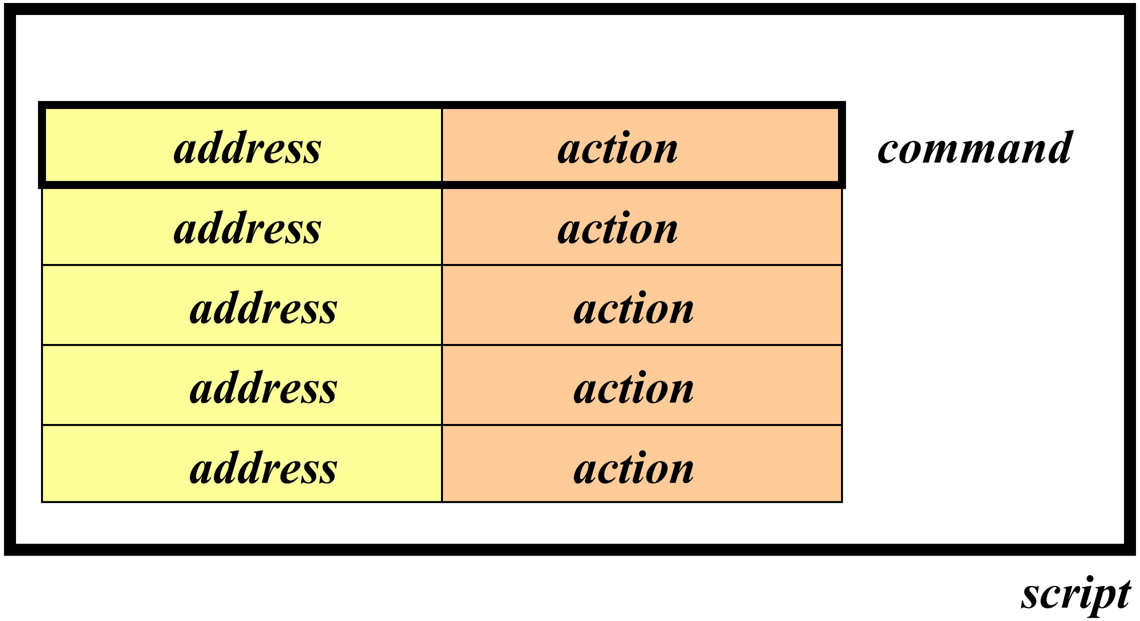
- All editing commands in a **sed** script are applied in order to each input line.
- If a command changes the input, subsequent command address will be applied to the current (modified) line in the pattern space, not the original input line.
- The original input file is unchanged (sed is a filter), and the results are sent to standard output (but can be redirected to a file).

Sed Architecture



Scripts

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



address	***action***
address	***action***
address	***action***
address	***action***
address	***action***

command

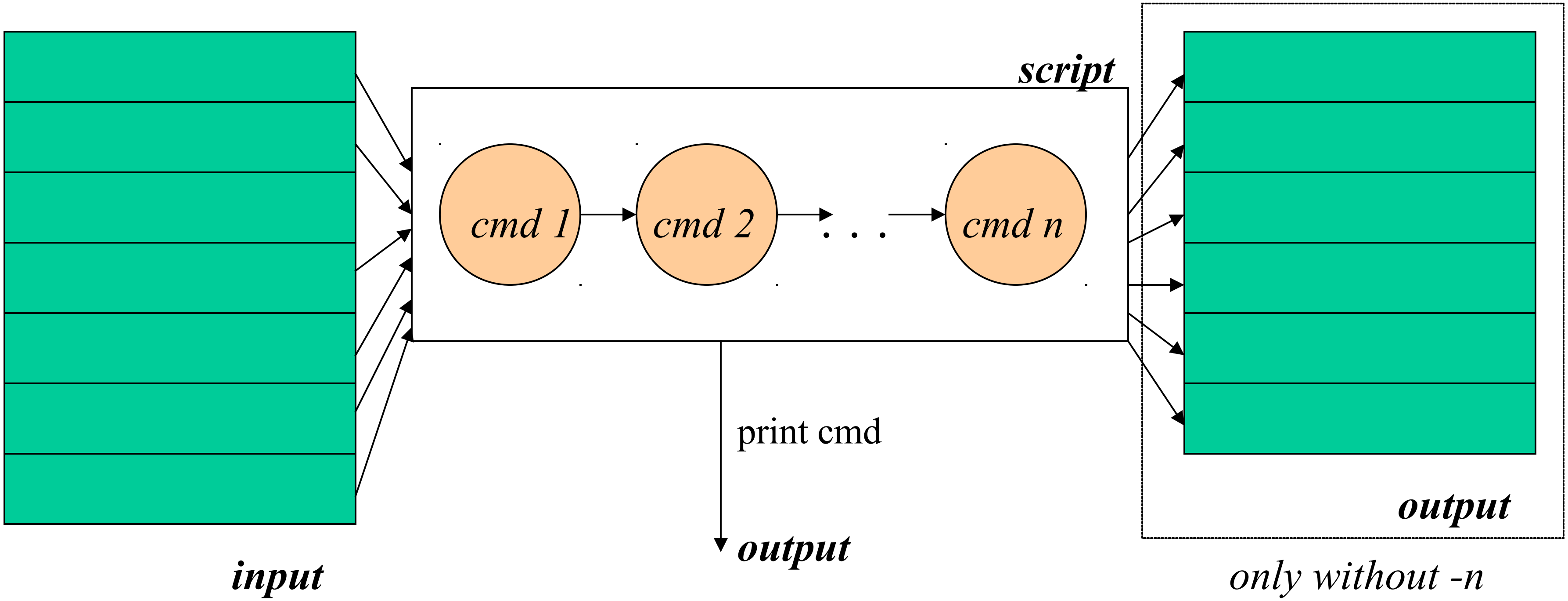
script

Scripts (continued)

- 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, the command is executed
  - If there is no match, the command is ignored
  - *sed* then repeats this action for every command in the 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

Sed 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



sed Commands

- sed commands have the general form
  - *[address[, address]][!]command [arguments]*
- *sed* copies each input line into a *pattern space*
  - If the address of the command matches the line in the *pattern space*, the command is applied to that line
  - If the 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*

Addressing

- An address can be either a line number or a pattern, enclosed in slashes (/*pattern*/)
- A pattern is described using *regular expressions* (BREs, as in **grep**)
- If no pattern is specified, the command will be applied to **all** lines of the input file
- To refer to the last line: **$**

Addressing (continued)

- Most commands will accept two addresses
  - If only one address is given, the command operates only on that line
  - If two comma separated addresses are given, then the command operates on a range of lines between the first and second address, inclusively
- The `!` operator can be used to negate an address, ie; *address!command* causes *command* to be applied to all lines that do ***not*** match *address*

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.

Address and Command Examples

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

Multiple Commands

- Braces **{}** can be used to apply multiple commands to an address

```
[/pattern/[,/pattern/]]{
command1
command2
command3
}
```

- Strange syntax:
  - The *opening brace* must be the last character on a line
  - The *closing brace* must be on a line by itself
  - Make sure there are no spaces following the braces

Sed Commands

- Although sed contains many editing commands, we are only going to cover the following subset:
  - **`s`** - substitute
  - **`a`** - append
  - **`i`** - insert
  - **`c`** - change
  - **`d`** - delete
  - **`p`** - print
  - **`r`** - read
  - **`w`** - write
  - **`y`** - transform
  - **`q`** - quit

sed Syntax

- Syntax: *sed [-n] [-e] [‘command’] [file…]*

  sed [-n] [-f scriptfile] [file…]

  - ***-n*** - only print lines specified with the print command (or the ‘p’ flag of the substitute (‘s’) command)
  - ***-f*** *scriptfile* - next argument is a filename containing editing commands
  - ***-e*** *command* - the next argument is an editing command rather than a filename, useful if multiple commands are specified
  - If the first line of a scriptfile is “***#n***”, sed acts as though ***-n*** had been specified

Print

- The Print command (**`p`**) can be used to force the pattern space to be output, useful if the ***-n*** option has been specified
- Syntax: **`[address1[,address2]]p`**
- Note: if the ***-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 the lines from the first blank line through the last line of the file

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

Substitute Examples

- **`s/Puff Daddy/P. Diddy/`**
  - Substitute P. Diddy for the first occurrence of Puff Daddy in *pattern space*
- **`s/Tom/Dick/2`**
  - Substitutes Dick for the second occurrence of Tom in the *pattern space*
- **`s/wood/plastic/p`**
  - Substitutes plastic for the first occurrence of wood and outputs (prints) *pattern space*

Replacement Patterns

- Substitute can use several special characters in the *replacement* string
  - **&** - replaced by the entire string matched in the regular expression for pattern
  - **\***n* - replaced by the *n*th substring (or subexpression) previously specified using “\\(“ and “\\)”
  - \ - used to escape the ampersand (&) and the backslash (\\)

Replacement Pattern Examples

```
"the UNIX operating system …"
s/.NI./wonderful &/
"the wonderful UNIX operating system …"
```

```
cat test1
first:second
one:two
sed 's/\(.*\):\(.*\)/\2:\1/' test1
second:first
two:one
```

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

- Pig Latin ("unix is fun" -> "nixuay siay unfay")

Append, Insert, and Change

- Syntax for these commands is a little strange because they **must** be specified on multiple lines
- **append** *[address]a*
  text
- **insert** *[address]i*
  text
- **change** *[address(es)]c*
  text
- append/insert for single lines only, not range

Append and Insert

- Append places *text* after the current line in pattern space
- Insert places *text* before the current line in pattern space
  - Each of these commands requires a \ following it.
    text must begin on the next line.
  - If text begins with whitespace, sed will discard it
    unless you start the line with a \
- Example:

```
/<Insert Text Here>/i\
Line 1 of inserted text\
\        Line 2 of inserted text
```

would leave the following in the pattern space

```
Line 1 of inserted text
             Line 2 of inserted text
<Insert Text Here>
```

Change

- Unlike Insert and Append, Change can be applied to either a single line address or a range of addresses
- When applied to a range, the entire range is replaced by text specified with change, not each line
  - *Exception*: If the Change command is executed with other commands enclosed in **{ }** that act on a range of lines, **each line** will be replaced with *text*
- No subsequent editing allowed

Change Examples

- Remove mail headers, ie; the address specifies a range of lines beginning with a line that begins with From until the first blank line.
  - The first example replaces all lines with a single occurrence of <Mail Header Removed>.
  - The second example replaces each line with <Mail Header Removed>

```
/^From /,/^$/c\
  <Mail Headers Removed>


/^From /,/^$/{
   s/^From //p
   c\
   <Mail Header Removed>
   }
```

Using `!`

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

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

  `/black/!s/cow/horse/` would substitute “horse” for “cow” on all lines except those that contained “black”

“The brown cow” -> “The brown horse”

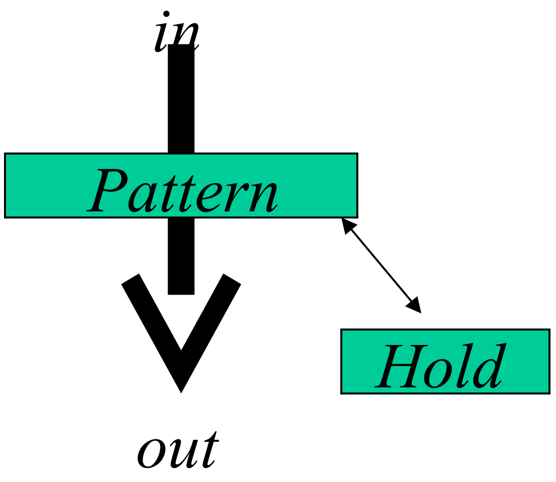
“The black cow” -> “The black cow”

Transform

- The Transform command (**y**) operates like **tr**, it does a one-to-one or character-to-character replacement
- Transform accepts zero, one or two addresses
- **`[address[,address]]y/abc/xyz/`**
  - every *a* within the specified address(es) is transformed to an *x*. The same is true for *b* to *y* and *c* to *z*
  - **`y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/`** changes **all** lower case characters on the addressed line to upper case
  - If you only want to transform specific characters (or a word) in the line, it is much more difficult and requires use of the *hold space*

Pattern and Hold spaces

- **Pattern space**: Workspace or temporary buffer where a single line of input is held while the editing commands are applied
- **Hold space**: Secondary temporary buffer for temporary storage only



```
h, H, g, G
```

Quit

- Quit causes **sed** to stop reading new input lines and stop sending them to standard output
- It takes at most a single line address
  - Once a line matching the address is reached, the script will be terminated
  - This 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*) use:
  - `sed '100q' filename`
  - sed will, by default, send the first 100 lines of *filename* to standard output and then quit processing

Sed Advantages

- Regular expressions
- Fast
- Concise

Sed Drawbacks

- Hard to remember text from one line to another
- Not possible to go backward in the file
- No way to do forward references like `/..../+1`
- No facilities to manipulate numbers
- Cumbersome syntax

Awk

Programmable Filters

Why is it called AWK?

Aho

Weinberger

Kernighan

Awk Introduction

- **awk**'s purpose: A general purpose programmable filter that handles text (strings) as easily as numbers
  - This makes **awk** one of the most powerful of the Unix utilities
- **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
- **awk** gets it’s input from
  - files
  - redirection and pipes
  - directly from standard input

AWK Highlights

- A programming language for handling common data manipulation tasks with only a few lines of code
- **awk** is a *pattern-action* language, like **sed**
- The language looks a little like *C* but automatically handles input, field splitting, initialization, and memory management
  - Built-in string and number data types
  - No variable type declarations
- **awk** is a great prototyping language
  - Start with a few lines and keep adding until it does what you want

Awk Features over Sed

- Convenient numeric processing
- Variables and control flow in the actions
- Convenient way of accessing fields within lines
- Flexible printing
- Built-in arithmetic and string functions
- C-like syntax

Structure of an AWK Program

- An **awk** 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}
```

Running an AWK Program

- There are several ways to run an Awk program
  - ***awk 'program' input_file(s)***
    - program and input files are provided as command-line arguments
  - ***awk 'program'***
    - program is a command-line argument; input is taken from standard input (yes, awk is a filter!)
  - ***awk -f program_file input_files***
    - program is read from a file

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**

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 `{ }`
- **awk** 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

Patterns

- Selector that determines whether *action* is to be executed
- *pattern* can be:
  - the special token **BEGIN** or **END**
  - regular expressions (enclosed with //)
  - arithmetic relation operators
  - string-valued expressions
  - arbitrary combination of the above
    - **`/NYU/`** matches if the string “NYU” is in the record
    - **`x > 0`** matches if the condition is true
    - **`/NYU/ && (name == "UNIX Tools")`**

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 the first input line is read.
  - **END**: actions are done after the last input line has been processed.

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* is performed on every line that matches *pattern*.
  - If *pattern* is not provided, *action* is performed on every input line
  - If *action* is not provided, all matching lines are sent to standard output.
- Since *patterns* and *actions* are optional, *actions* must be enclosed in braces to distinguish them from *pattern*.

An Example

```
ls | awk '
  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!
```

Variables

- **awk** scripts can define and use variables

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

- Some variables are predefined

Records

- Default record separator is **newline**
  - By default, **awk** processes its input a 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)
  - **`awk -F`***c* 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 are unadorned

Simple Output From AWK

- Printing Every Line
  - If an action has no pattern, the action is performed to all input lines
    - `{ print }` will print 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 output

Output (continued)

- **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 the number of fields, the first field, and the 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 }`**

Output (continued)

- Printing Line Numbers
  - The built-in variable NR can be used to print line numbers
  - **`{ print NR, $0 }`** will print 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

Fancier Output

- Lining Up Fields
  - Like C, Awk 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 **awk**. You have to insert them yourself.

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

Selection

- Awk 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 == "NYU"`
    - `/NYU/`
  - Combinations of Patterns
    - `$2 >= 4 || $3 >= 20`
  - Selection by Line Number
    - `NR >= 10 && NR <= 20`

Arithmetic and variables

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

Computing with AWK

- Counting is easy to do with Awk

```
$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
    }
```

Handling Text

- One major advantage of Awk is its ability to handle strings as easily as many languages handle numbers
- Awk variables can hold strings of characters as well as numbers, and Awk 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 }
```

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 }
```

Built-in Functions

- **awk** contains a number of built-in functions. length is one of them.
- Counting Lines, Words, and Characters using length (a poor man's **wc**)

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

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

Control Flow Statements

- **awk** 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 While

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

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)
}
```

Arrays

- Array elements are not declared
- Array subscripts can have ***any*** value:
  - Numbers
  - Strings! (***associative arrays***)
- Examples
  - **`arr[3]="value"`**
  - **`grade["Korn"]=40.3`**

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]
        }
    }
```

Useful One (or so)-liners

-
  ```
  END { print NR }
  ```

-
  ```
  NR == 10
  ```

-
  ```
  { print $NF }
  ```

-
  ```
    { field = $NF }
    END { print field }
  ```

-
  ```
  NF > 4
  ```

-
  ```
  $NF > 4
  ```

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

More One-liners

-
  ```
  /Jeff/ { nlines = nlines + 1 }
   END   { print nlines }
  ```

-
  ```
  $1 > max { max = $1; maxline = $0 }
   END     { print max, maxline }
  ```

-
  ```
  NF > 0
  ```

-
  ```
  length($0) > 80
  ```

-
  ```
  { print NF, $0}
  ```

-
  ```
  { print $2, $1 }
  ```

-
  ```
  { temp = $1; $1 = $2; $2 = temp; print }
  ```

-
  ```
  { $2 = ""; print }
  ```

Even More One-liners

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

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

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

Awk Variables

- $0, $1, $2, $NF
- NR - Number of records processed
- NF - Number of fields in current record
- FILENAME - name of current input file
- FS - Field separator, space or TAB by default
- OFS - Output field separator, space by default
- ARGC/ARGV - Argument Count, Argument Value array
  - Used to get arguments from the command line

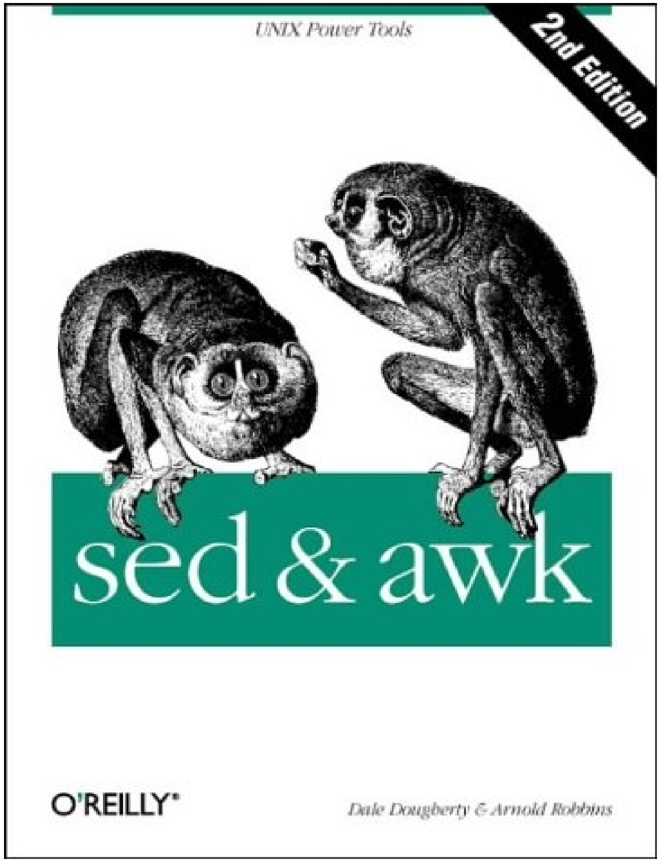
Operators

- = 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
- +, -, /, *, %, ^
- String concatenation

Built-In Functions

- Arithmetic
  - **sin**, **cos**, **atan**, **exp**, **int**, **log**, **rand**, **sqrt**
- String
  - **length**, **substitution**, find substrings, split strings
- Output
  - **print**, **printf**, print and printf to file
- Special
  - **system** - executes a Unix 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

More Information



on the website