Lecture 4

Regular Expressions

grep and **sed**

Previously

- Basic UNIX Commands
  - Files: rm, cp, mv, ls, ln
  - Processes
- Unix Filters
  - **cat**, **head**, **tail**, tee, **wc**
  - **cut**, paste
  - **find**
  - **comm**, diff, cmp
  - **sort**, **uniq**
  - tr

Subtleties of commands

- Executing commands with find
- Specification of columns in cut
- Specification of columns in sort
- Methods of input
  - Standard in
  - File name arguments
  - Special "-" filename

Today

- Regular Expressions
  - Allow you to search for text in files
  - **grep** command
- Stream *manipulation*:
  - **sed**

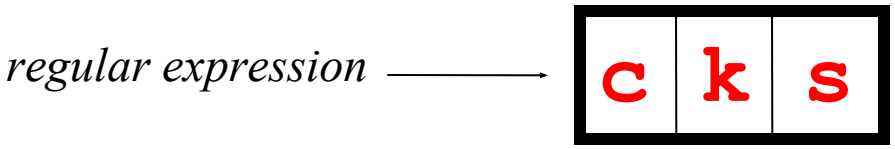
Regular Expressions

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.




```
Open Source rocks.
```

match

```
Open Source sucks.
```

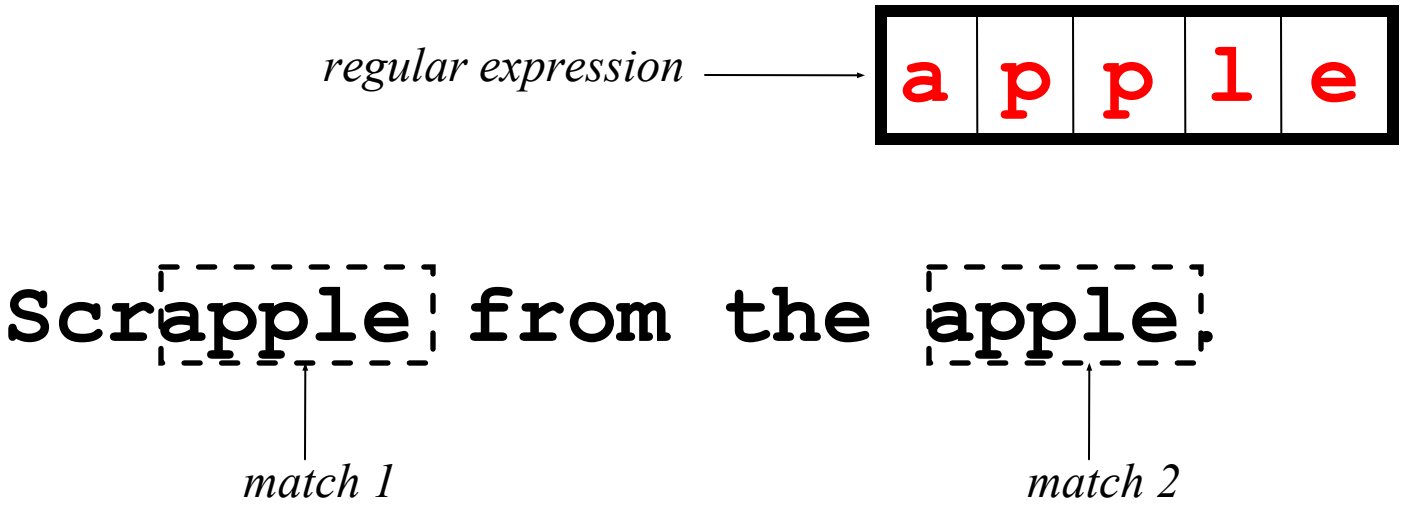
match

```
Open Source is okay.
```

no match

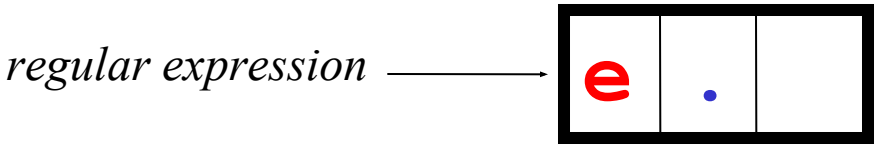
Regular Expressions

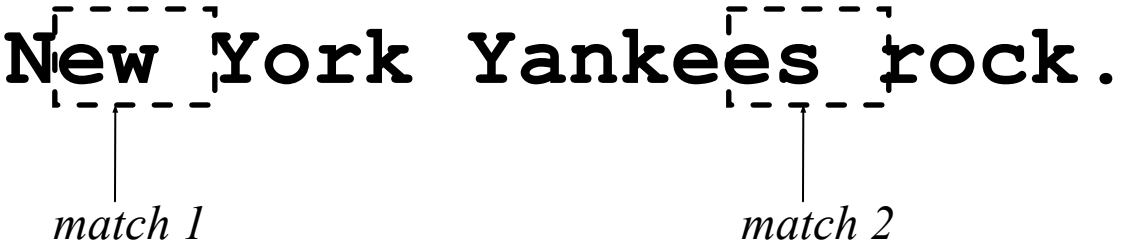
- A regular expression can match a string in more than one place.



Regular Expressions

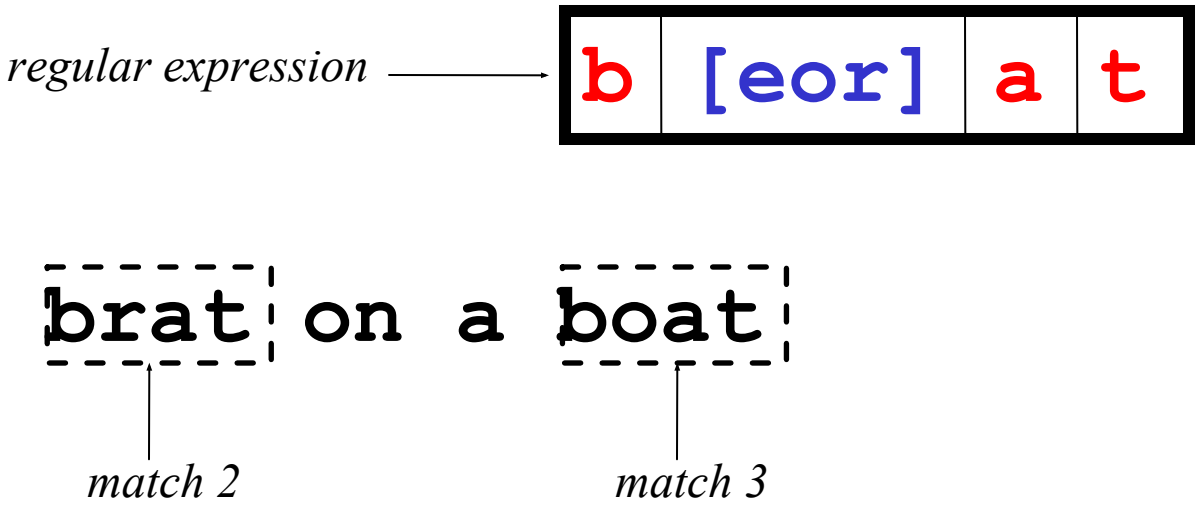
The `.` regular expression can be used to match any character.





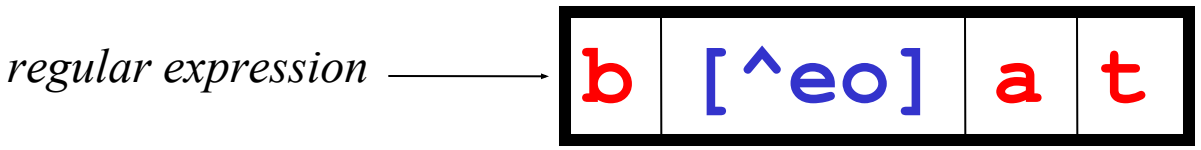
Character Classes

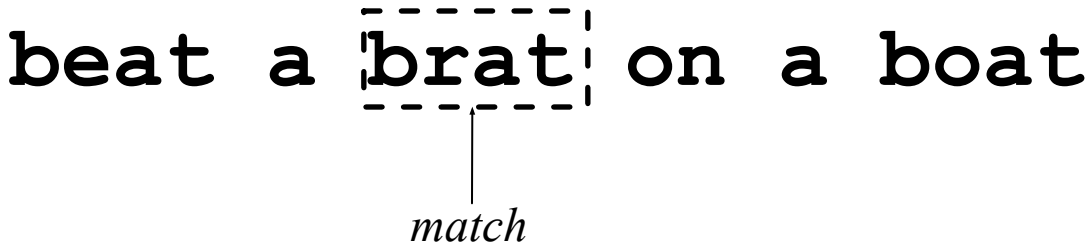
Character classes `[]` can be used to match any specific set of characters.



Negated Character Classes

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





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

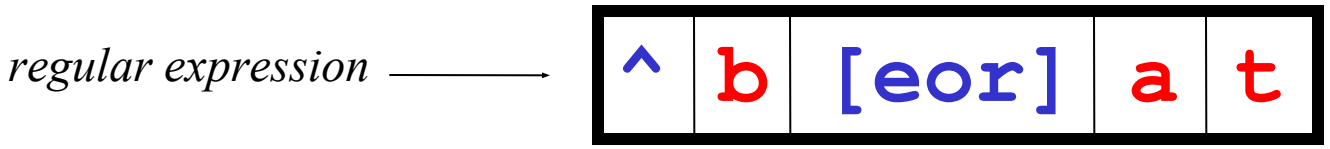
- Commonly used character classes can be referred to by name (*alpha*, *lower*, *upper*, *alnum*, *digit*, *punct*, *cntrl*)
- Syntax `[:`*name*`:]`

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

- Important for portability across languages

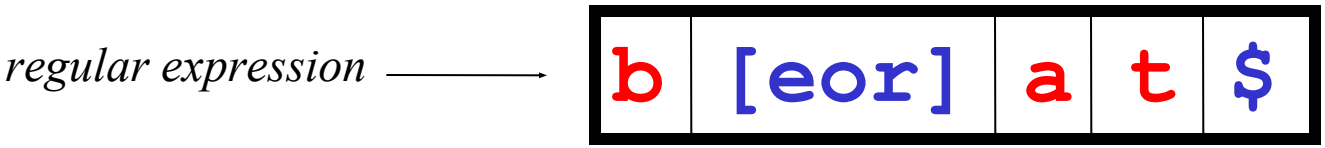
Anchors

- 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



beat a brat on a boat

match



beat a brat on a boat

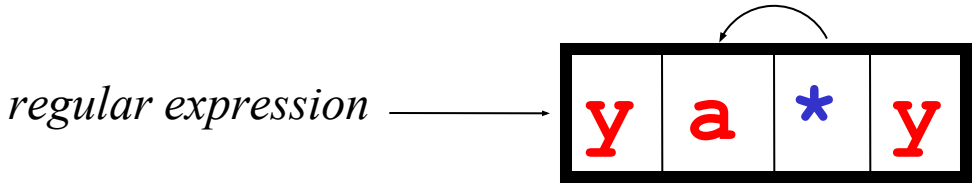
match

---

^word$ ^$

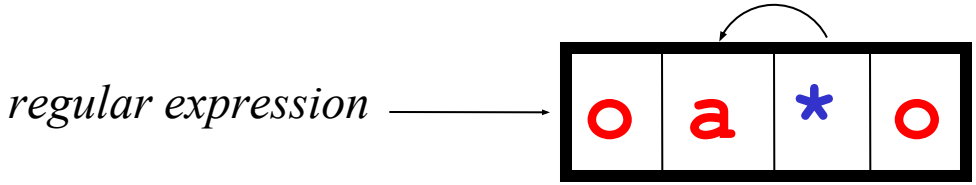
Repetition

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




```
I got mail, yaaaaaaaaaaay!
```

match



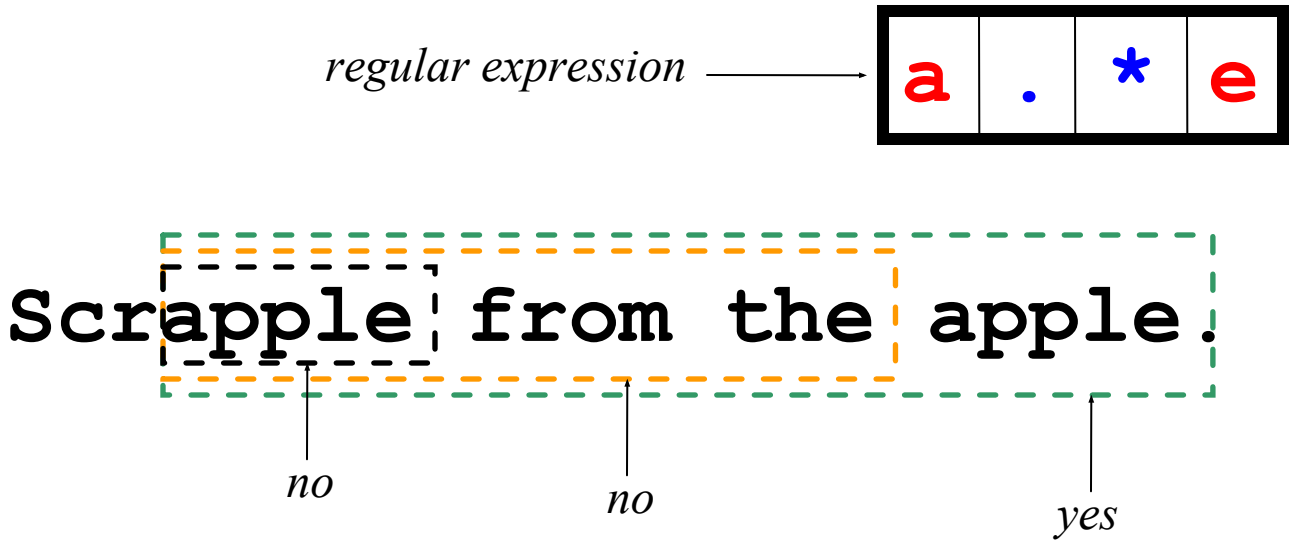

```
kill -9 O.S. Tools!
```

match

`.*`

Match length

A match will be the longest string that satisfies the regular expression.



Repetition Ranges

- Ranges can also be specified
  - `{ }` 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 `*` or `{ }` applies to more than just the previous character, use `( )` notation
- Subexpressions 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`
  - Used when you are looking for text in a document
- 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

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 (we will cover shortly)

Syntax

- Regular expression concepts we have seen so far are common to **grep** and **egrep**.
- But grep and egrep have slightly different syntax:
  - **grep**: *BRE*s
  - **egrep**: *ERE*s
- 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, some characters are special to **grep** (eg `*` and `.`)
- To get literal characters, we *escape* the character with a `\` (backslash)
- Suppose we want to search for the literal 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
  - `(F|Ph)at` will match ‘Fat’ or ‘Phat’
  - `^(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
  - `n(ie|ei)ther` then matches “niether” or “neither”, not “nie” or “either” as would happen without the parenthesis - `nie|either`

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
- Looks for *n*th subexpression
- 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]>`

The 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;
```

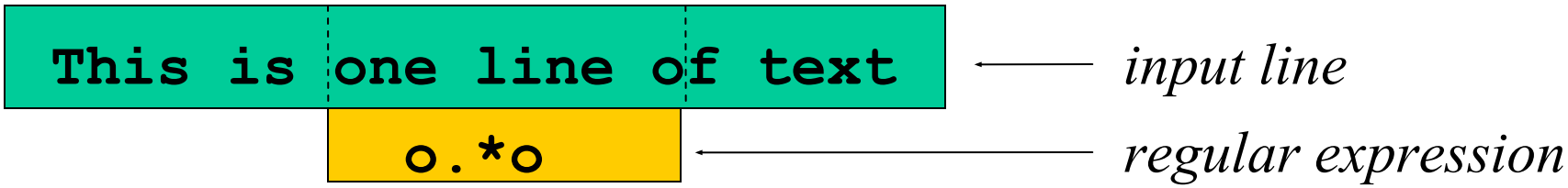
- **egrep** has its limits: For example, it cannot match all lines that contain a number divisible by 7.

Fun with the Dictionary

- **`/usr/share/dict/words`** contains about 25,000 words
  - **`egrep hh /usr/share/dict/words`**
    - beachhead
    - highhanded
    - withheld
    - withhold
- **egrep** as a simple spelling checker: Specify plausible alternatives you know
  - **`egrep "n(ie|ei)ther" /usr/share/dict/words`**
  - **`neither`**
- How many words have 3 a's one letter apart?
  - **`egrep a.a.a /usr/share/dict/words | wc -l`**
    - 54
  - **`egrep u.u.u /usr/share/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`**



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

Quick Reference

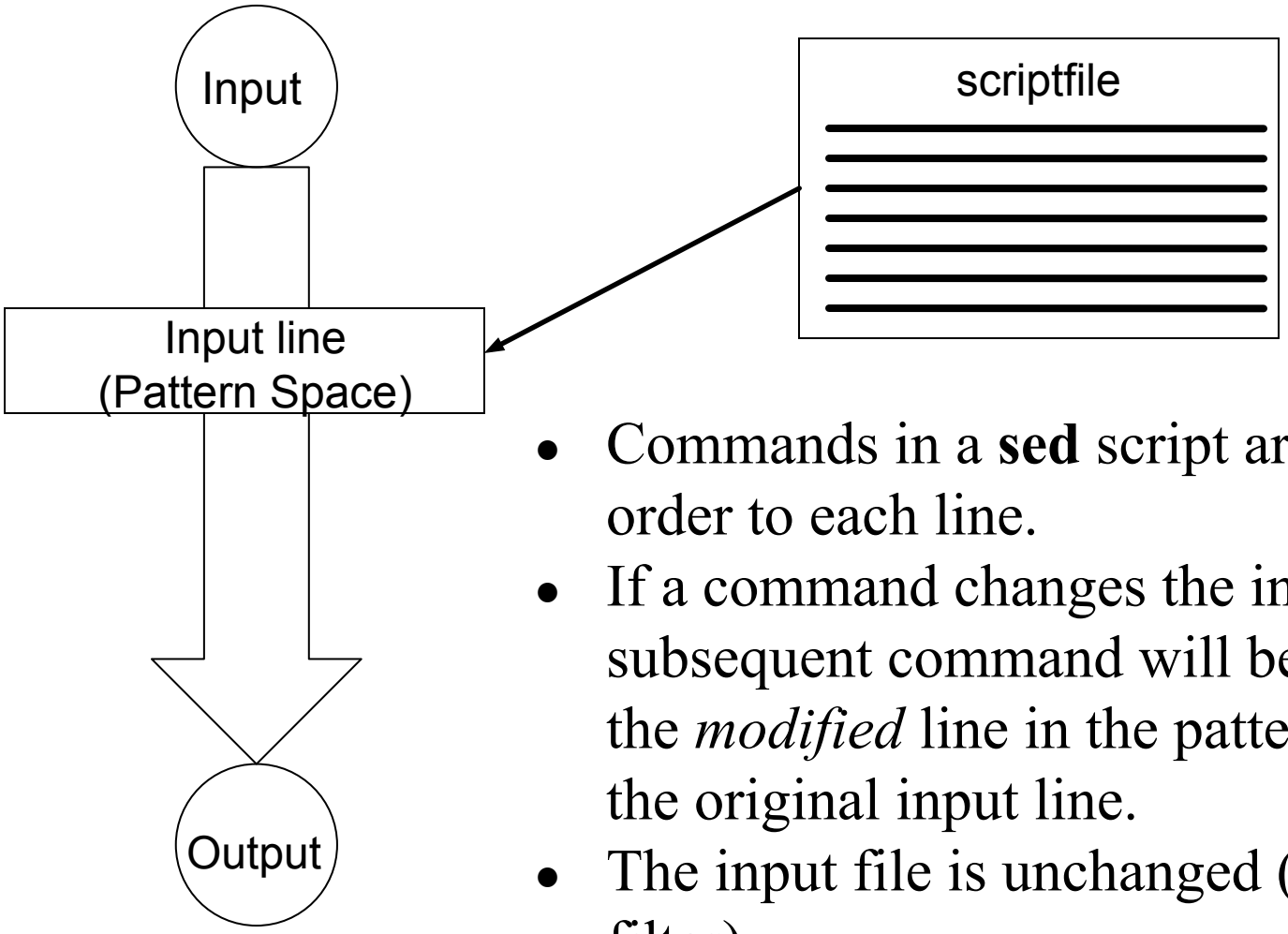
GNU Extensions (non-POSIX)

- Backreferences in EREs
- ERE functionality with BRE syntax (**`\+`**, **`\?`**)
- Anchors for word boundaries
  - **`\b`** is word boundary, `\B` is not
  - **`\<`** is start of word, **`\>`** is end of word
- Character classes
  - Shorthand for **`[[:alnum:]]`** is **`\w`**
  - Shorthand for **`[[:space:]]`** is **`\s`**
  - **`\W`** and **`\S`** are negation of above

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 a *script*
  - There is an interactive editor *ed* which accepts the same commands
- A Unix filter
  - Superset of previously mentioned tools

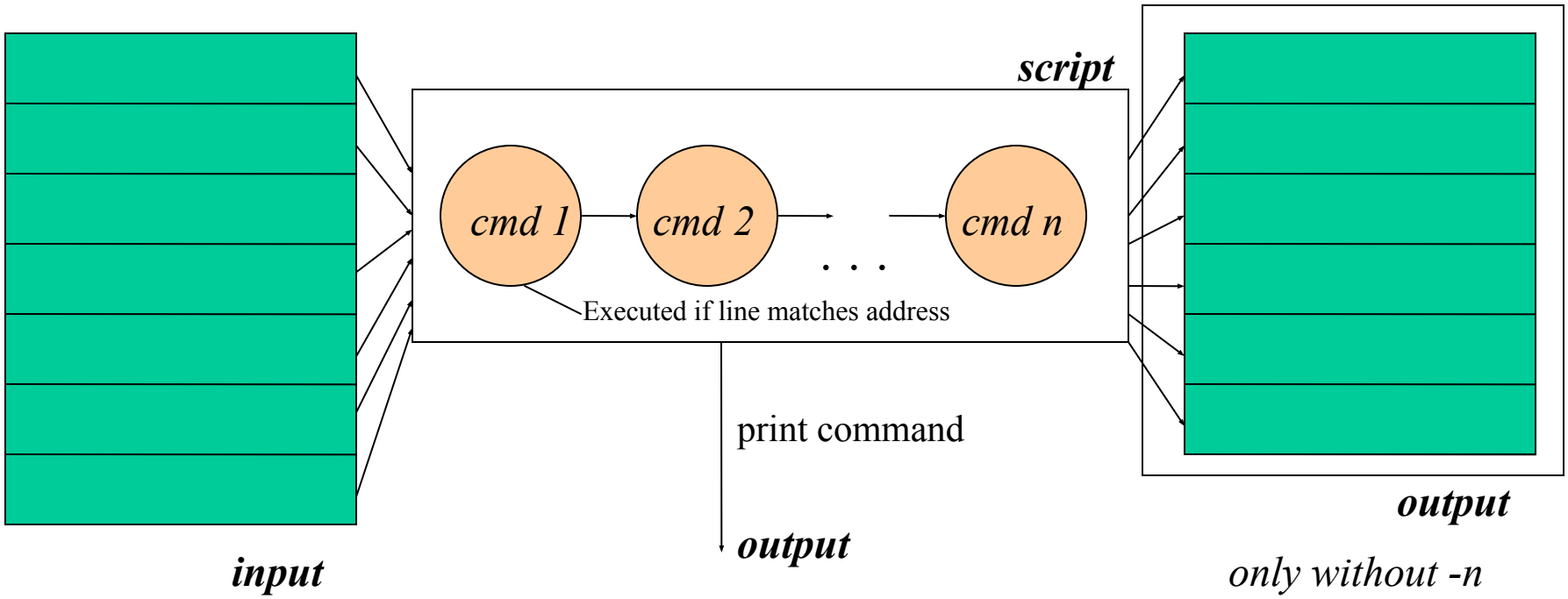
Sed Architecture



- Commands in a **sed** script are applied in order to each line.
- If a command changes the input, subsequent command will be applied to the *modified* line in the pattern space, not the original input line.
- The input file is unchanged (sed is a filter).
- Results are sent to standard output unless redirected.

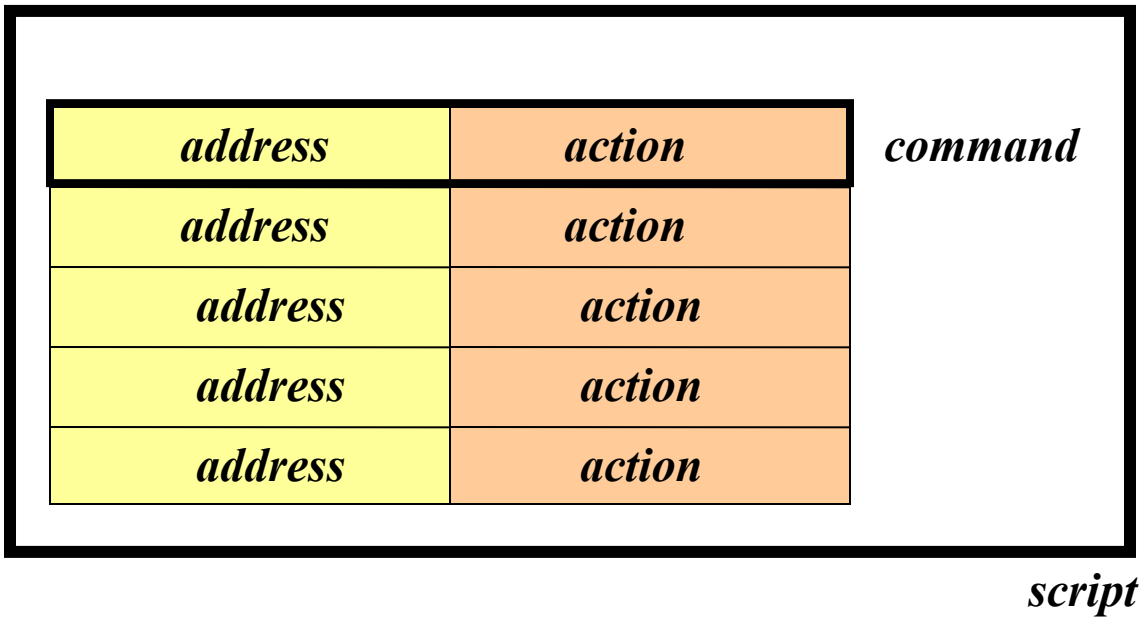
Sed Flow of Control

- *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



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



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 script is “***#n***”, sed acts as though ***-n*** had been specified

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
  - You can mix line numbers and patterns
- 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 will not cover them all. The most widely used commands are:

  - **`s`** - substitute
  - **`a`** - append
  - **`i`** - insert
  - **`c`** - change
  - **`d`** - delete
  - **`p`** - print
  - **`y`** - transform
  - **`q`** - quit

- Among these, **`s`** is most widely used by far!

Print

- The Print command (**`p`**) can be used to force the pattern space to be output, primarily useful if the ***-n*** option has been specified
  - If the ***-n*** option has not been specified, **`p`** will cause the line to be output twice!
- Syntax: **`[address1[,address2]]p`**
- 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 (useful with -n)
- default flag is 1

Substitute Examples

`s/Puff Daddy/P. Diddy/`

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

`s/Jeff/Jeffrey/2`

Substitutes Jeffrey for the second occurrence of Jeff 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/\([[:alpha:]]\)\([^[:space:]]*\)/\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

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 (note: no ranges!)

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

Next Time

awk: Program your own filter!