CS 241: Systems Programming Lecture 3. More Shell

Fall 2023 Prof. Stephen Checkoway

⟨command⟩ ⟨options⟩ ⟨arguments⟩

```
⟨command⟩ ⟨options⟩ ⟨arguments⟩
```

command is the name of a command or a path to a program

```
⟨command⟩ ⟨options⟩ ⟨arguments⟩
```

- command is the name of a command or a path to a program
- options are directives to the command to control its behavior

```
⟨command⟩ ⟨options⟩ ⟨arguments⟩
```

- command is the name of a command or a path to a program
- \options \are directives to the command to control its behavior
 - Short options are a hyphen and a letter: -h

```
⟨command⟩ ⟨options⟩ ⟨arguments⟩
```

- command is the name of a command or a path to a program
- \options \are directives to the command to control its behavior
 - Short options are a hyphen and a letter: -h
 - Long options are (usually) two hyphens and multiple letters: --help

```
⟨command⟩ ⟨options⟩ ⟨arguments⟩
```

- command is the name of a command or a path to a program
- \options \are directives to the command to control its behavior
 - Short options are a hyphen and a letter: -h
 - Long options are (usually) two hyphens and multiple letters: --help
 - Multiple short options can be combined -a -b -c is the same as -abc

```
⟨command⟩ ⟨options⟩ ⟨arguments⟩
```

- command is the name of a command or a path to a program
- \options \are directives to the command to control its behavior
 - Short options are a hyphen and a letter: -h
 - Long options are (usually) two hyphens and multiple letters: --help
 - Multiple short options can be combined -a -b -c is the same as -abc
 - Options can take arguments: -o file.txt or --output=file.txt

```
⟨command⟩ ⟨options⟩ ⟨arguments⟩
```

- command is the name of a command or a path to a program
- options are directives to the command to control its behavior
 - Short options are a hyphen and a letter: -h
 - Long options are (usually) two hyphens and multiple letters: --help
 - Multiple short options can be combined -a -b -c is the same as -abc
 - Options can take arguments: -o file.txt or --output=file.txt
- \arguments\) are the things the command acts on

```
⟨command⟩ ⟨options⟩ ⟨arguments⟩
```

- command is the name of a command or a path to a program
- options are directives to the command to control its behavior
 - Short options are a hyphen and a letter: -h
 - Long options are (usually) two hyphens and multiple letters: --help
 - Multiple short options can be combined -a -b -c is the same as -abc
 - Options can take arguments: -o file.txt or --output=file.txt
- \arguments\) are the things the command acts on
 - Often file paths or server names or URLs

⟨command⟩ ⟨options⟩ ⟨arguments⟩

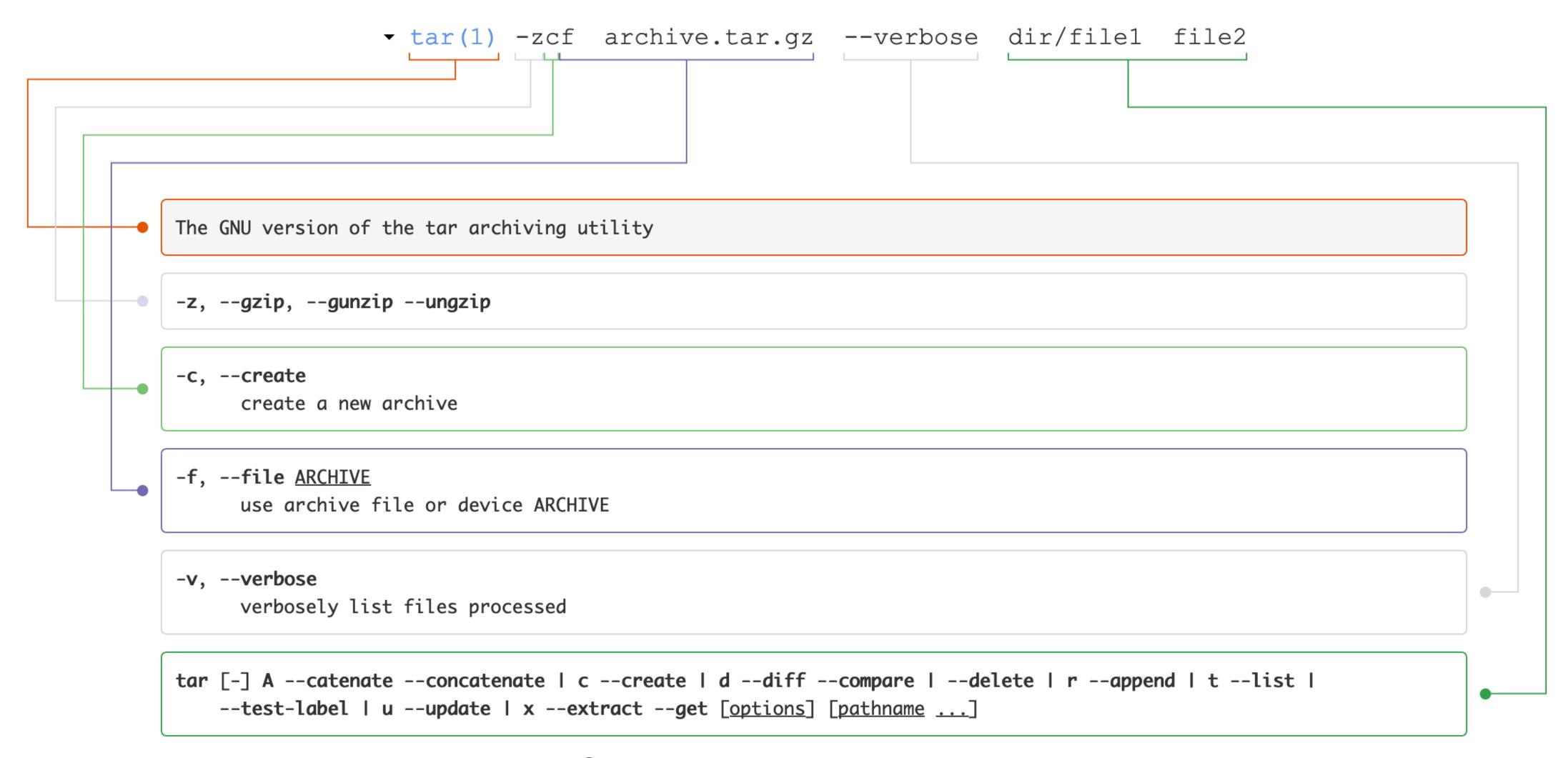
- command is the name of a command or a path to a program
- options are directives to the command to control its behavior
 - Short options are a hyphen and a letter: -h
 - Long options are (usually) two hyphens and multiple letters: --help
 - Multiple short options can be combined -a -b -c is the same as -abc
 - Options can take arguments: -o file.txt or --output=file.txt
- \arguments\) are the things the command acts on
 - Often file paths or server names or URLs
 - When no arguments are given (or a single –), many commands read stdin

```
⟨command⟩ ⟨options⟩ ⟨arguments⟩
```

- command is the name of a command or a path to a program
- \options \are directives to the command to control its behavior
 - Short options are a hyphen and a letter: -h
 - Long options are (usually) two hyphens and multiple letters: --help
 - Multiple short options can be combined -a -b -c is the same as -abc
 - Options can take arguments: -o file.txt or --output=file.txt
- \arguments\) are the things the command acts on
 - Often file paths or server names or URLs
 - When no arguments are given (or a single –), many commands read stdin

Example: tar -zcf archive.tar.gz --verbose dir/file1 file2

Example meaning



Shell builtins

- Functionality built into bash (all listed in the manual)
- E.g., cd, alias, echo, pwd

Shell builtins

- Functionality built into bash (all listed in the manual)
- ► E.g., cd, alias, echo, pwd

Shell functions

User-defined functions (we'll get to these later)

Shell builtins

- Functionality built into bash (all listed in the manual)
- ► E.g., cd, alias, echo, pwd

Shell functions

User-defined functions (we'll get to these later)

Aliases

► E.g., alias ls='ls --color=auto'

Shell builtins

- Functionality built into bash (all listed in the manual)
- ► E.g., cd, alias, echo, pwd

Shell functions

User-defined functions (we'll get to these later)

Aliases

► E.g., alias ls='ls --color=auto'

Programs stored on the file system

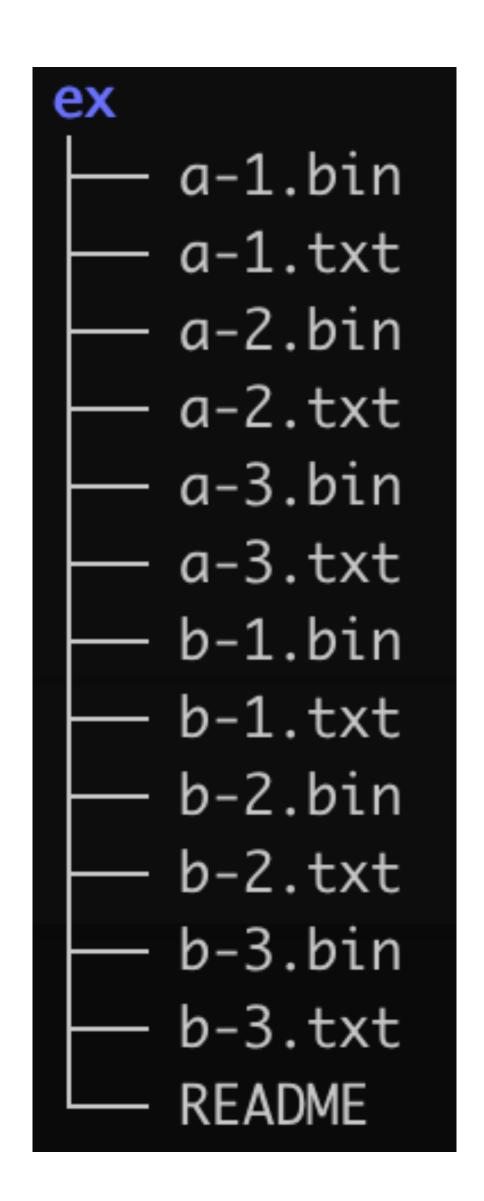
- bin, /usr/bin, /usr/local/bin, /sbin, /usr/sbin
- ► E.g., ssh, cat, ls, rm

Pathname expansion/globbing

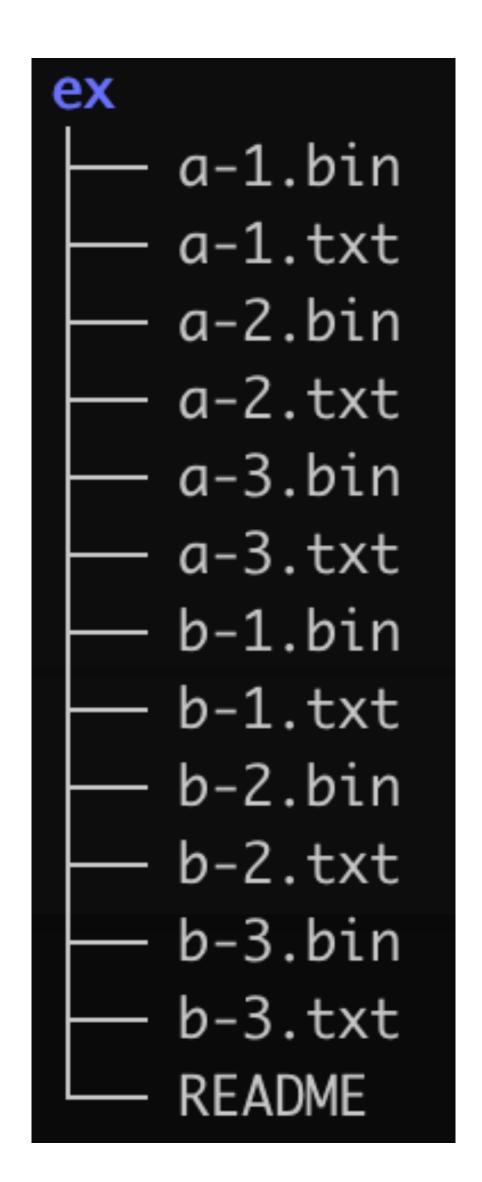
Bash performs pathname expansion via pattern matching (a.k.a. globbing) on each unquoted word containing a wild card

```
Wild cards: *,?,[
```

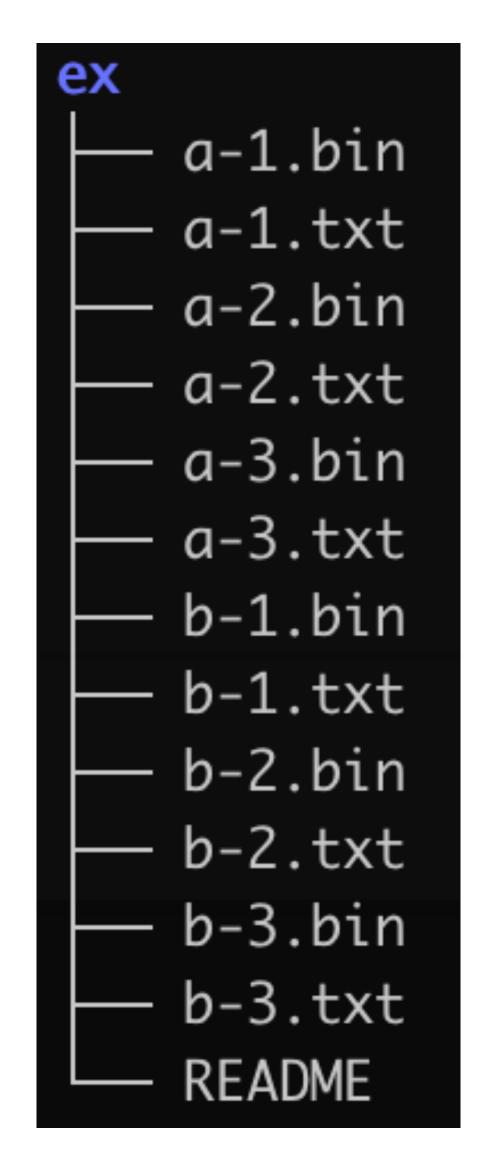
- * matches zero or more characters
- ? matches any one character
- [...] matches any single character between the brackets, e.g., [atz]
- [!...] or [^...] matches any character not between the brackets
- [x-y] matches any character in the range, e.g., [a-f]



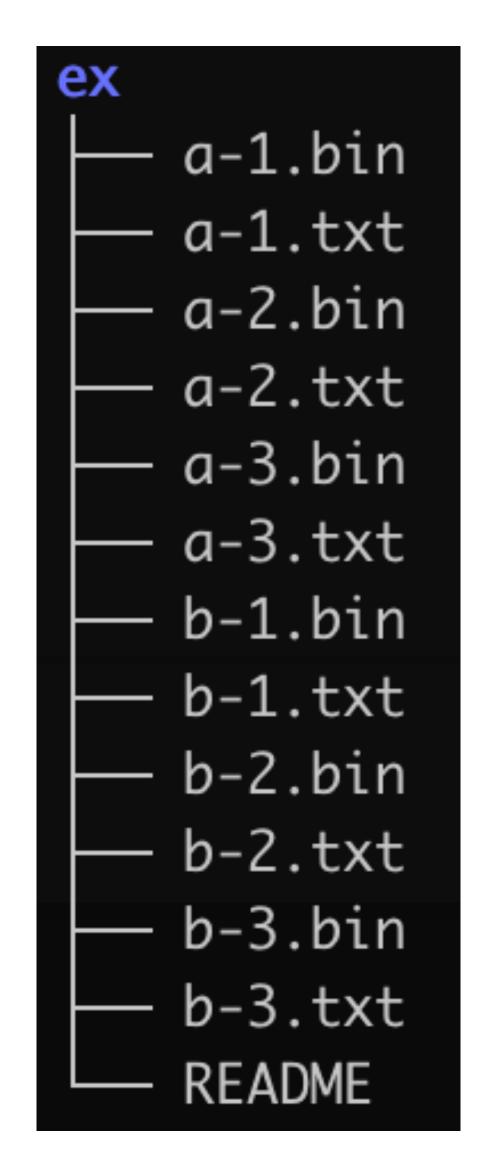
```
$ ls ex/*.txt
```



```
$ ls ex/*.txt
ex/a-1.txt ex/a-2.txt ex/a-3.txt ex/b-1.txt
ex/b-2.txt ex/b-3.txt
```

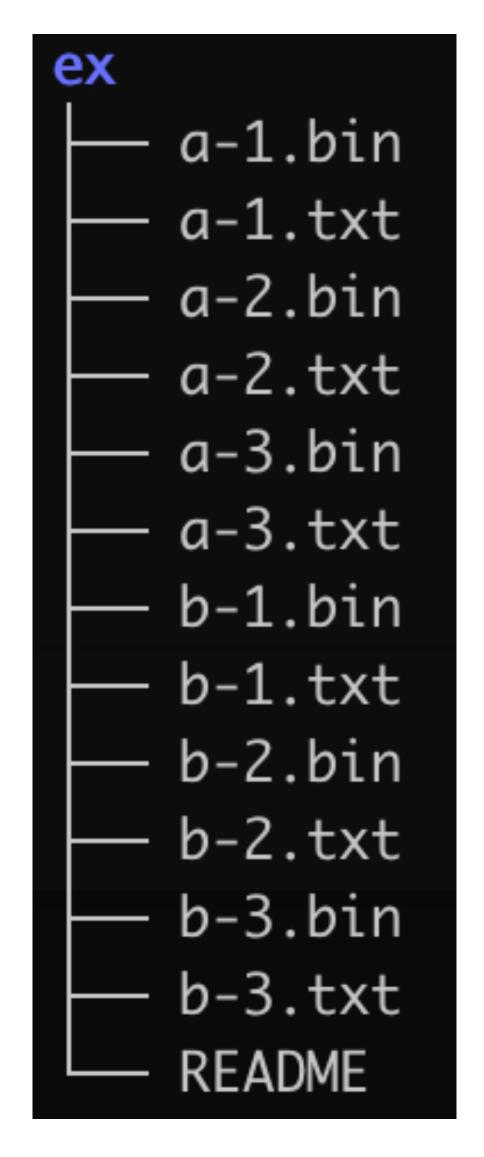


```
$ ls ex/*.txt
ex/a-1.txt ex/a-2.txt ex/a-3.txt ex/b-1.txt
ex/b-2.txt ex/b-3.txt
$ ls ex/?-3.*
```



```
$ ls ex/*.txt
ex/a-1.txt ex/a-2.txt ex/a-3.txt ex/b-1.txt
ex/b-2.txt ex/b-3.txt

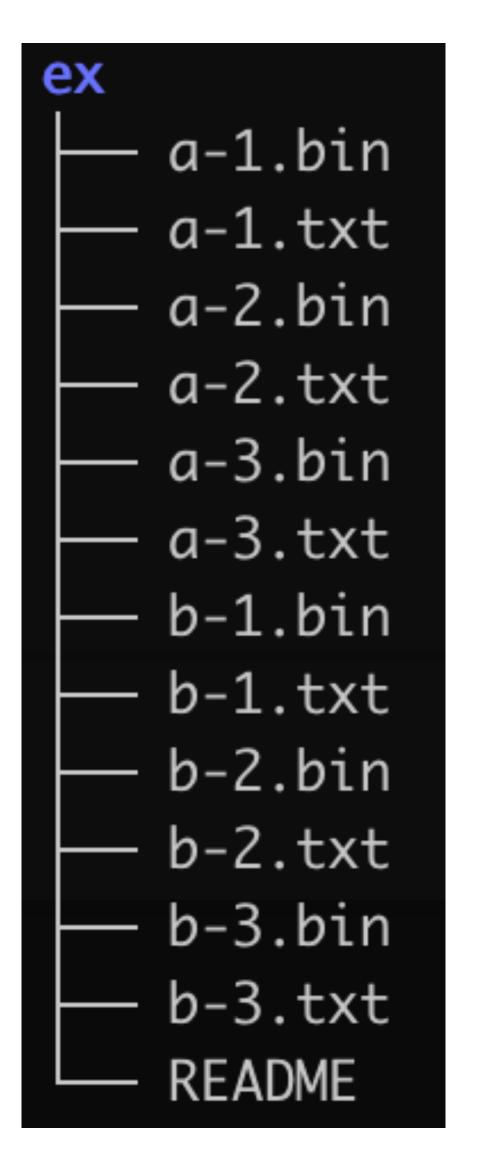
$ ls ex/?-3.*
ex/a-3.bin ex/a-3.txt ex/b-3.bin ex/b-3.txt
```



```
$ ls ex/*.txt
ex/a-1.txt ex/a-2.txt ex/a-3.txt ex/b-1.txt
ex/b-2.txt ex/b-3.txt

$ ls ex/?-3.*
ex/a-3.bin ex/a-3.txt ex/b-3.bin ex/b-3.txt

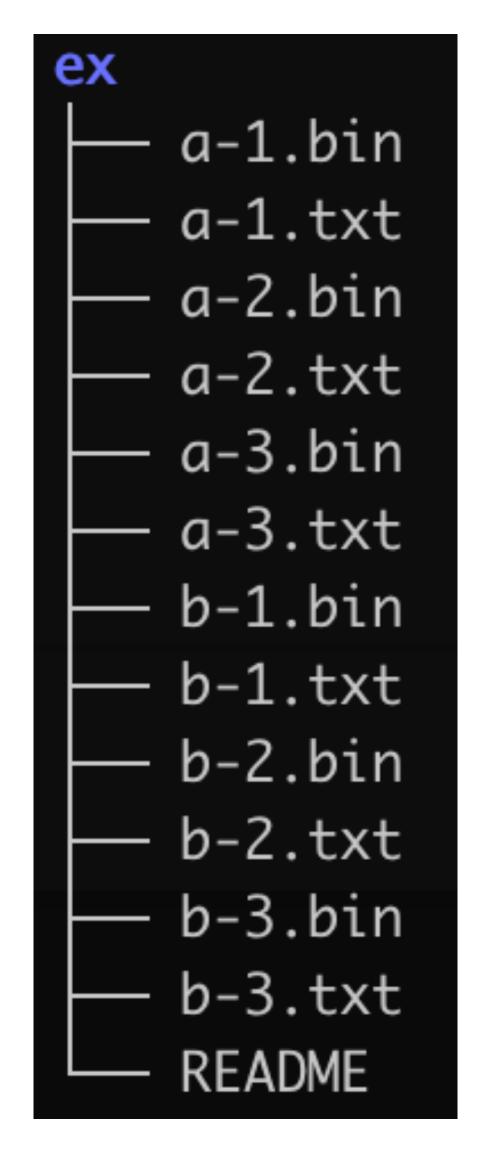
$ ls ex/[^acd]-[0-9].b*in
```



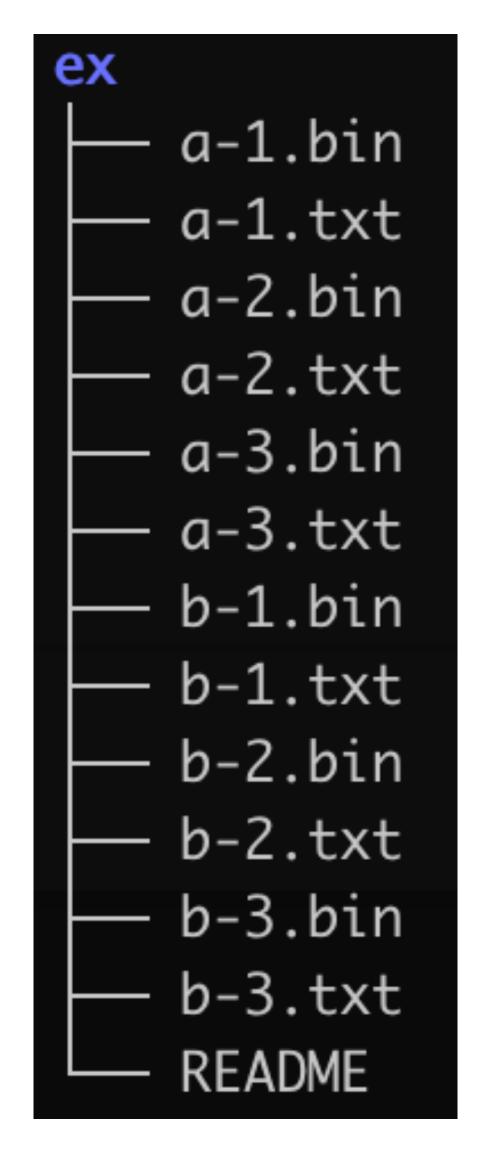
```
$ ls ex/*.txt
ex/a-1.txt ex/a-2.txt ex/a-3.txt ex/b-1.txt
ex/b-2.txt ex/b-3.txt

$ ls ex/?-3.*
ex/a-3.bin ex/a-3.txt ex/b-3.bin ex/b-3.txt

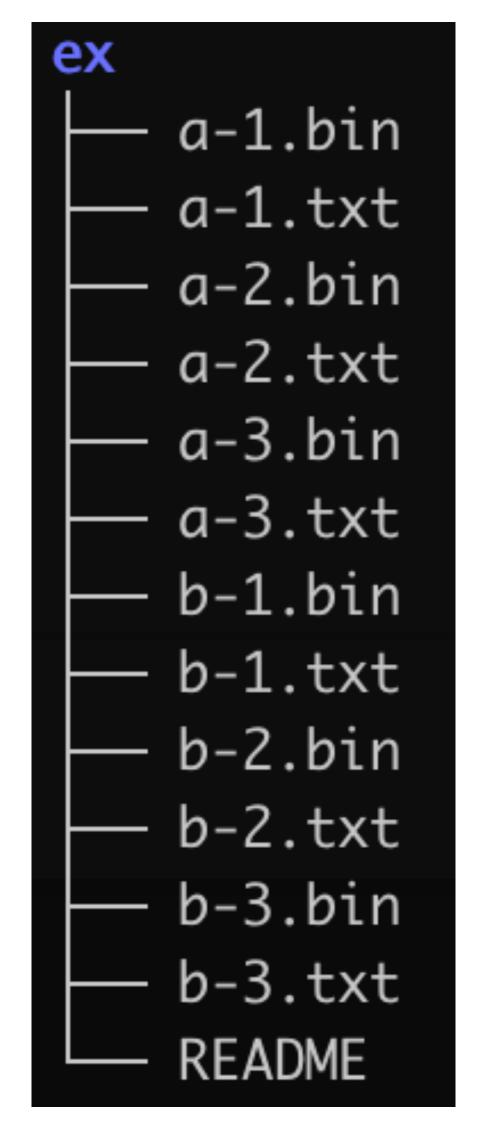
$ ls ex/[^acd]-[0-9].b*in
ex/b-1.bin ex/b-2.bin ex/b-3.bin
```



```
$ ls ex/*.txt
ex/a-1.txt ex/a-2.txt ex/a-3.txt ex/b-1.txt
ex/b-2.txt ex/b-3.txt
$ ls ex/?-3.*
ex/a-3.bin ex/a-3.txt ex/b-3.bin ex/b-3.txt
$ ls ex/[^acd]-[0-9].b*in
ex/b-1.bin ex/b-2.bin ex/b-3.bin
 ls "ex/*"
```



```
$ ls ex/*.txt
ex/a-1.txt ex/a-2.txt ex/a-3.txt ex/b-1.txt
ex/b-2.txt ex/b-3.txt
$ ls ex/?-3.*
ex/a-3.bin ex/a-3.txt ex/b-3.bin ex/b-3.txt
$ ls ex/[^acd]-[0-9].b*in
ex/b-1.bin ex/b-2.bin ex/b-3.bin
 ls "ex/*"
ls: cannot access 'ex/*': No such file or
directory
```



Which command copies all Rust source files (those whose names end in .rs) from the directory a/b to the directory /tmp?

C. \$ cp a/b/*.rs /tmp

Typical Unix tool behavior

- \$ program
 - reads from stdin, writes to stdout
- \$ program file1 file2 file3
 - runs 'program' on the 3 files, write to stdout
- \$ program -
 - For programs that require filenames, might read from stdin

Every running program has (by default) 3 open "files" referred to by their file descriptor number

Input comes from stdin (file descriptor 0)

- input() # Python: Read a line
- System.in.read(var) // Java: Read bytes and store in var array
- \$ IFS= read -r var # Read a line and store in var variable

Normal output goes to stdout (file descriptor 1)

- print(var) # Python
- System.out.println(var) // Java
- > \$ echo "\${var}" # Bash

Normal output goes to stdout (file descriptor 1)

- print(var) # Python
- System.out.println(var) // Java
- \$ echo "\${var}" # Bash

Error messages traditionally go to stderr (file descriptor 2)

- print(var, file=sys.stderr) # Python
- System.err.println(var) // Java
- \$ echo "\${var}" >&2 # Bash

Redirection

>file — redirect standard output (stdout) to file with truncation

```
>file — redirect standard output (stdout) to file with truncation
```

>>file — redirect stdout to file, but append

```
>file — redirect standard output (stdout) to file with truncation>>file — redirect stdout to file, but append
```

<file — redirect input (stdin) to come from file</pre>

```
>file — redirect standard output (stdout) to file with truncation
>>file — redirect stdout to file, but append
<file - redirect input (stdin) to come from file

    connect stdout from left to stdin on right

  $ Is wc
2>file — redirect standard error (stderr) to file with truncation
```

```
>file — redirect standard output (stdout) to file with truncation
>>file — redirect stdout to file, but append
<file - redirect input (stdin) to come from file

    connect stdout from left to stdin on right

  $ Is wc
2>file — redirect standard error (stderr) to file with truncation

    redirect stderr to stdout

2>&1
```

```
$ echo 'Hi!' >output.txt
```

```
$ echo 'Hi!' >output.txt
$ cat <input.txt</pre>
```

```
$ echo 'Hi!' >output.txt
$ cat <input.txt
$ sort <input.txt >output.txt
```

```
$ echo 'Hi!' >output.txt
$ cat <input.txt
$ sort <input.txt >output.txt
$ ps -ax | grep bash
```

```
$ echo 'Hi!' >output.txt
$ cat <input.txt
$ sort <input.txt >output.txt
$ ps -ax | grep bash
$ grep hello file | sort | uniq -c
```

```
$ echo 'Hi!' >output.txt
$ cat <input.txt
$ sort <input.txt >output.txt
$ ps -ax | grep bash
$ grep hello file | sort | uniq -c
$ echo Hello | cut -c 1-4 >>result.txt
```

```
$ echo 'Hi!' >output.txt
$ cat <input.txt
$ sort <input.txt >output.txt
$ ps -ax grep bash
$ grep hello file | sort | uniq -c
$ echo Hello | cut -c 1-4 >>result.txt
$ ./process <input | tail -n 4 >output
```

(Almost) everything is a file

Files on the file system

Network sockets (for communicating with remote computers, e.g., web browsers, ssh, mail clients etc.)

Terminal I/O

A bunch of special files

- /dev/null Writes are ignored, reads return end-of-file (EOF)
- /dev/zero Writes are ignored, reads return arbitrarily many 0 bytes
- /dev/urandom Reads return arbitrarily many (pseudo) random bytes

Given that /dev/null ignores all data written to it, how can we run the program foo and redirect stderr so no error messages appear in our terminal but we continue to see normal output on stdout?

- A. \$ foo >/dev/null
- B. \$ foo 1>/dev/null
- C. \$ foo 2>/dev/null
- D.\$ foo | /dev/null
- E.\$ foo &2>/dev/null

Some programs read all of their input on stdin before terminating. If foo is such a program, how can we run foo such that it has no input at all? (foo is just an example, not a real program.)

- A. \$ foo </dev/null
- B. \$ foo </dev/zero
- C. \$ foo </dev/urandom</pre>
- D. \$ foo </dev/eof</pre>
- E.\$ echo foo