CS 241: Systems Programming
Lecture 3. More Shell

Spring 2024
Prof. Stephen Checkoway

Anatomy of a single command

⟨`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

```
  ▾ tar(1)  -zcf  archive.tar.gz  --verbose  dir/file1  file2
```

The GNU version of the tar archiving utility

-z, --gzip, --gunzip --ungzip

-c, --create
    create a new archive

-f, --file <u>ARCHIVE</u>
    use archive file or device ARCHIVE

-v, --verbose
    verbosely list files processed

tar [-] A --catenate --concatenate | c --create | d --diff --compare | --delete | r --append | t --list |
    --test-label | u --update | x --extract --get [<u>options</u>] [<u>pathname</u> <u>...</u>]
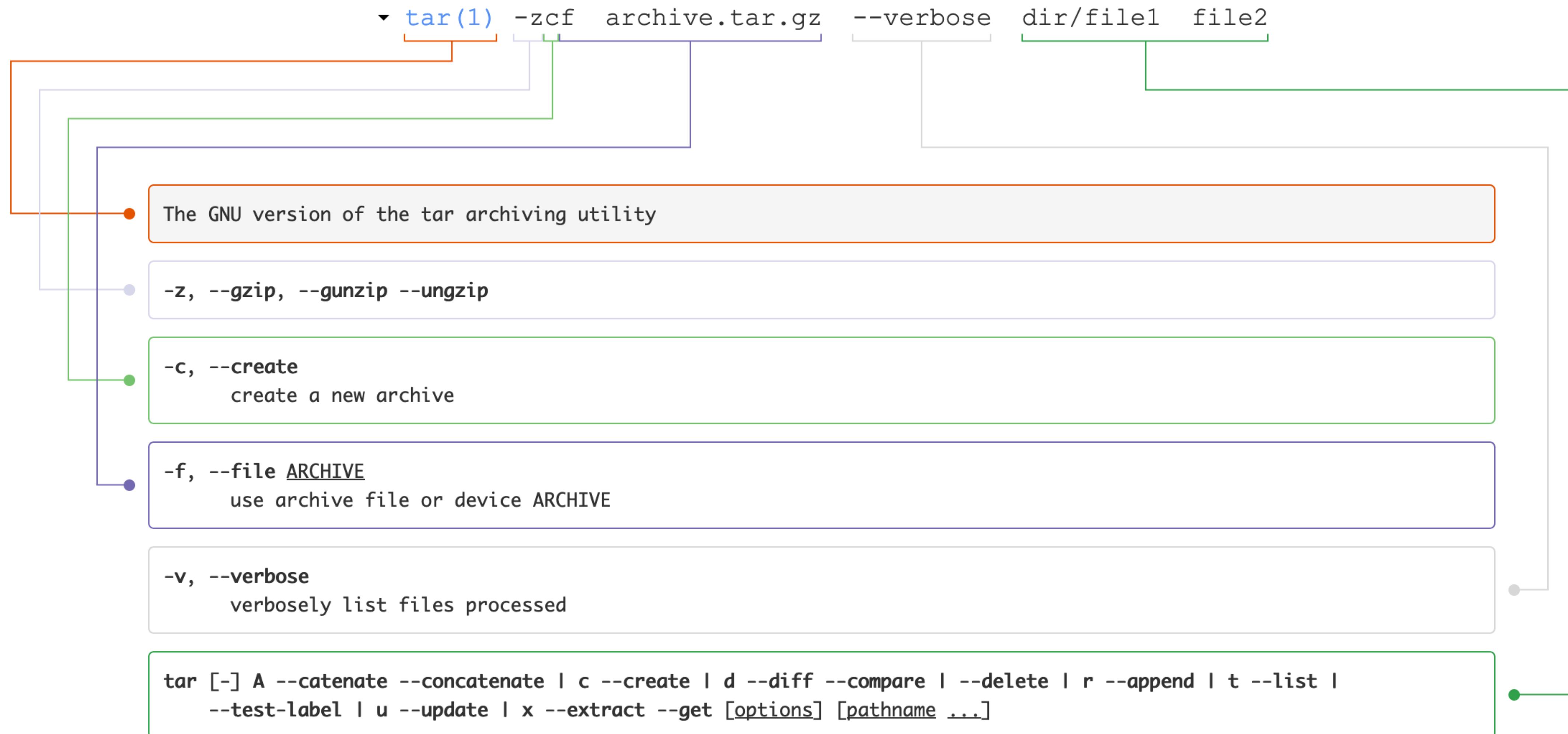
Click to go to explainshell.com

3

Shell commands

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`

Performing repetitive tasks

Setup: You have a project directory containing some source code, some data, and a README.

```
project
├── README.md
├── a.csv
├── b.csv
├── c.csv
├── d.csv
└── process.py
```

➡️

```
project
├── README.md
├── data
│   ├── a.csv
│   ├── b.csv
│   ├── c.csv
│   └── d.csv
└── process.py
```

You want to reorganize the project by moving all of the CSV files into a new data directory

A suboptimal approach

We can create the new data directory easily

```
$ cd project
$ mkdir data
```

Moving all of the files is really repetitive

```
$ mv a.csv data
$ mv b.csv data
$ mv c.csv data
$ mv d.csv data
```

Even this is repetitive: `$ mv a.csv b.csv c.csv d.csv data`

A better approach: globbing

```
$ mv *.csv data
```

The *.csv isn't handled by the mv command, but by bash itself

*.csv will be replaced with a (space separated) list of files that in the directory that end with .csv

In other words, that command becomes the following before it is executed

```
$ mv a.csv b.csv c.csv d.csv data
```

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]`

Example

```
ex
├── a-1.bin
├── a-1.txt
├── a-2.bin
├── a-2.txt
├── a-3.bin
├── a-3.txt
├── b-1.bin
├── b-1.txt
├── b-2.bin
├── b-2.txt
├── b-3.bin
├── b-3.txt
└── README
```

Example

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


ex
├── a-1.bin
├── a-1.txt
├── a-2.bin
├── a-2.txt
├── a-3.bin
├── a-3.txt
├── b-1.bin
├── b-1.txt
├── b-2.bin
├── b-2.txt
├── b-3.bin
├── b-3.txt
└── README

Example

```
$ 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
```

Example

```
$ 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.*
```

Example

```
$ 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
```

```
ex
├── a-1.bin
├── a-1.txt
├── a-2.bin
├── a-2.txt
├── a-3.bin
├── a-3.txt
├── b-1.bin
├── b-1.txt
├── b-2.bin
├── b-2.txt
├── b-3.bin
├── b-3.txt
└── README
```

Example

```
$ 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
├── a-1.bin
├── a-1.txt
├── a-2.bin
├── a-2.txt
├── a-3.bin
├── a-3.txt
├── b-1.bin
├── b-1.txt
├── b-2.bin
├── b-2.txt
├── b-3.bin
├── b-3.txt
└── README
```
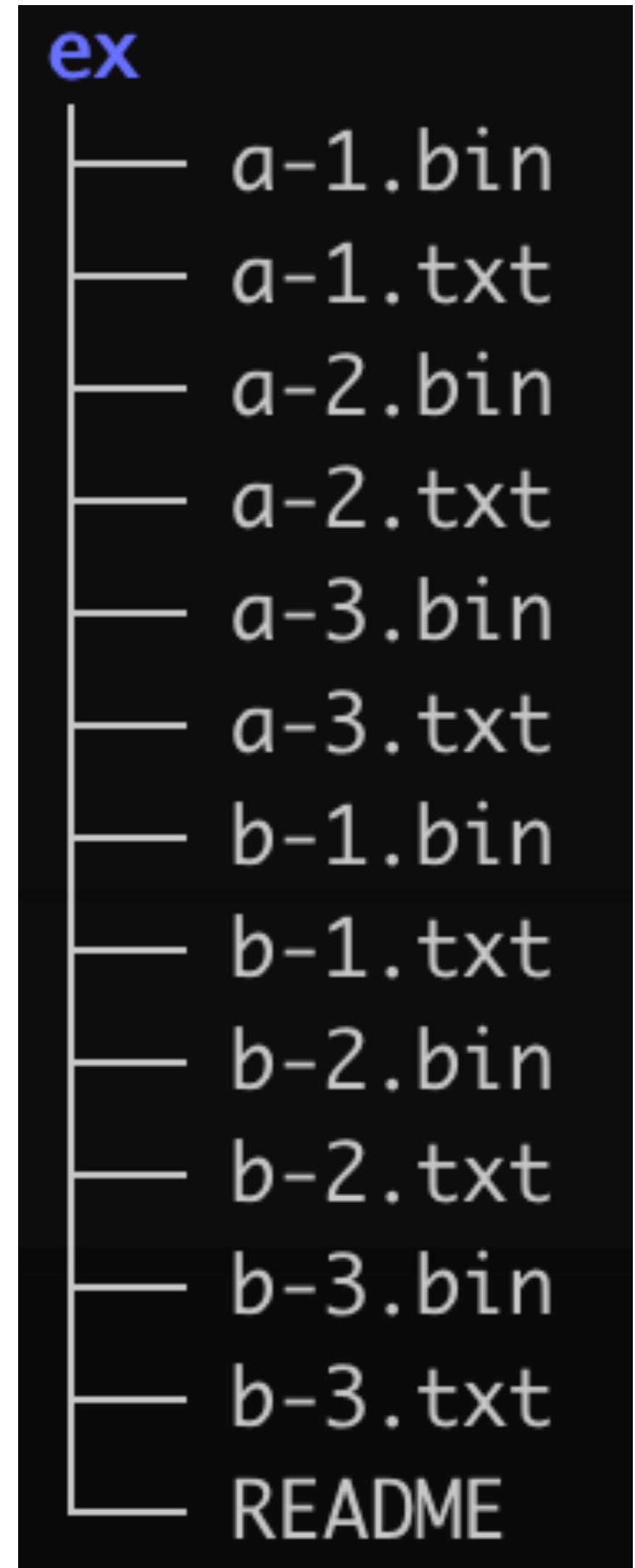
Example

```
$ 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
```

Example

```
$ 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/*"
```

```
ex
├── a-1.bin
├── a-1.txt
├── a-2.bin
├── a-2.txt
├── a-3.bin
├── a-3.txt
├── b-1.bin
├── b-1.txt
├── b-2.bin
├── b-2.txt
├── b-3.bin
├── b-3.txt
└── README
```
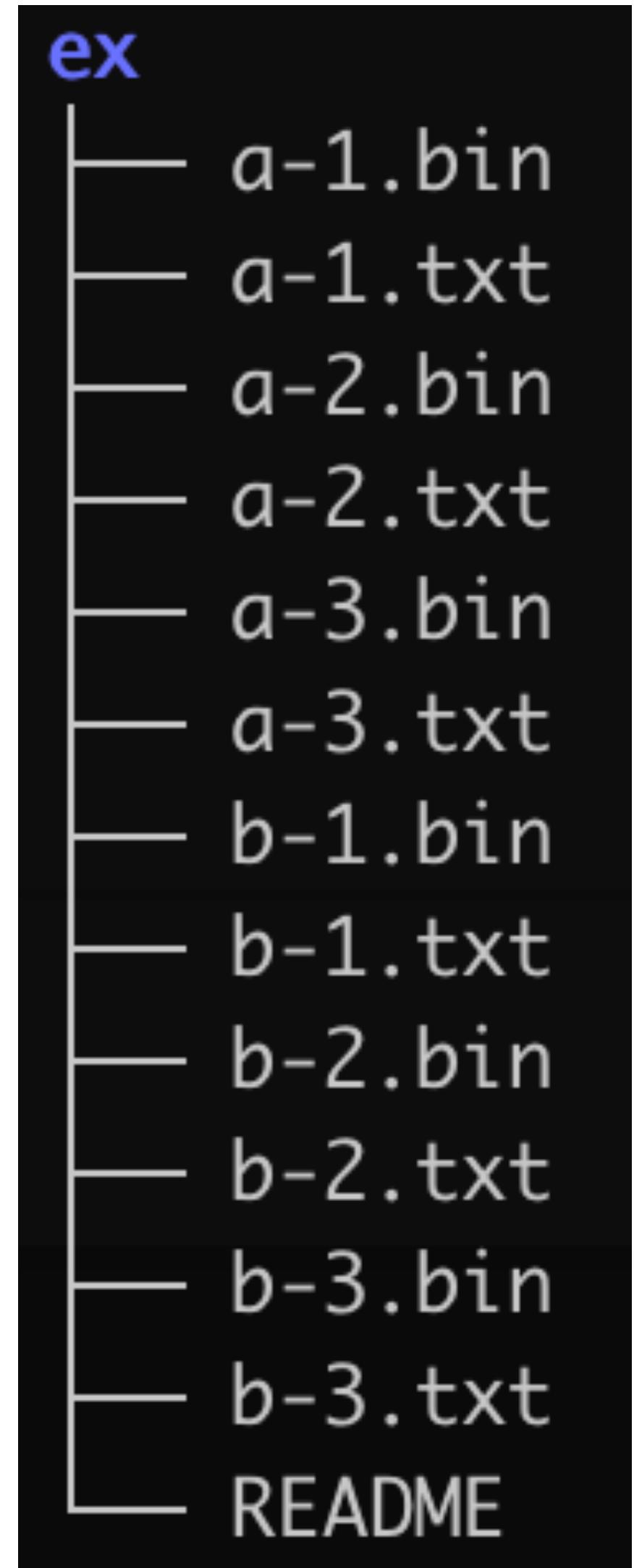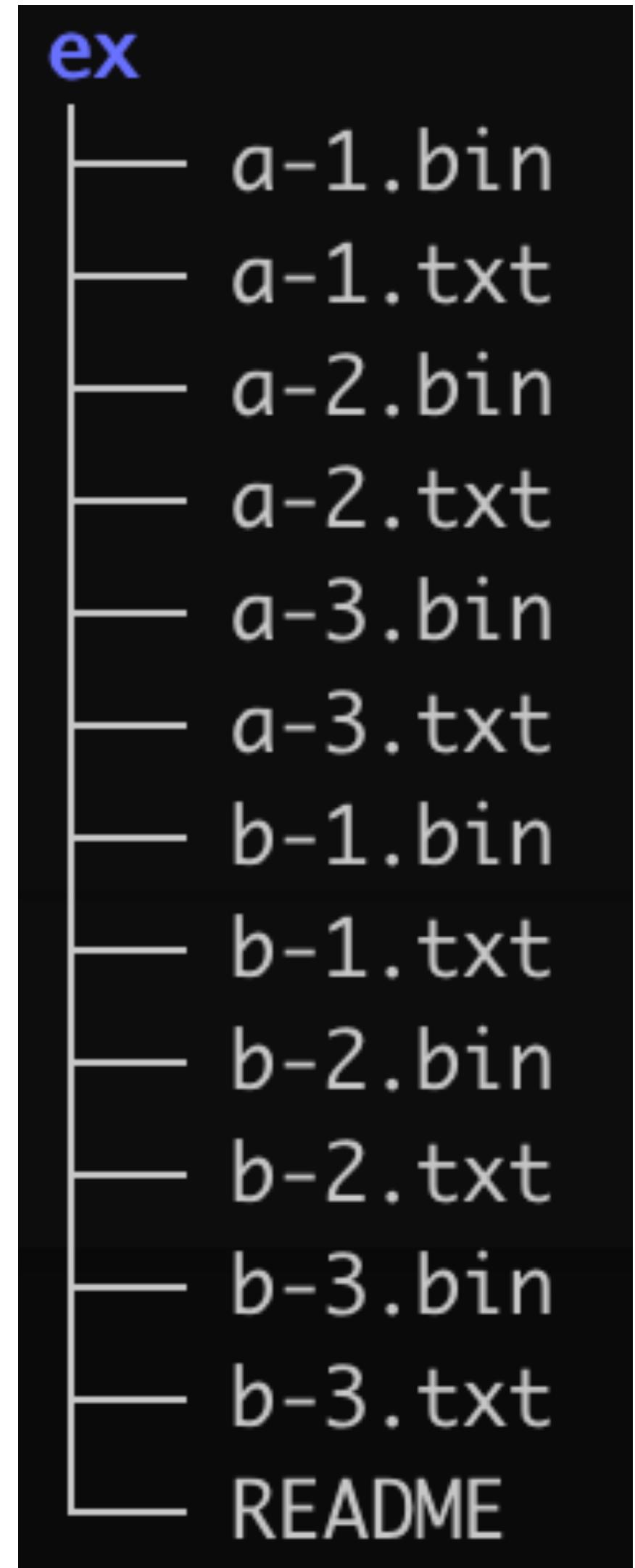
Example

```
$ 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
```
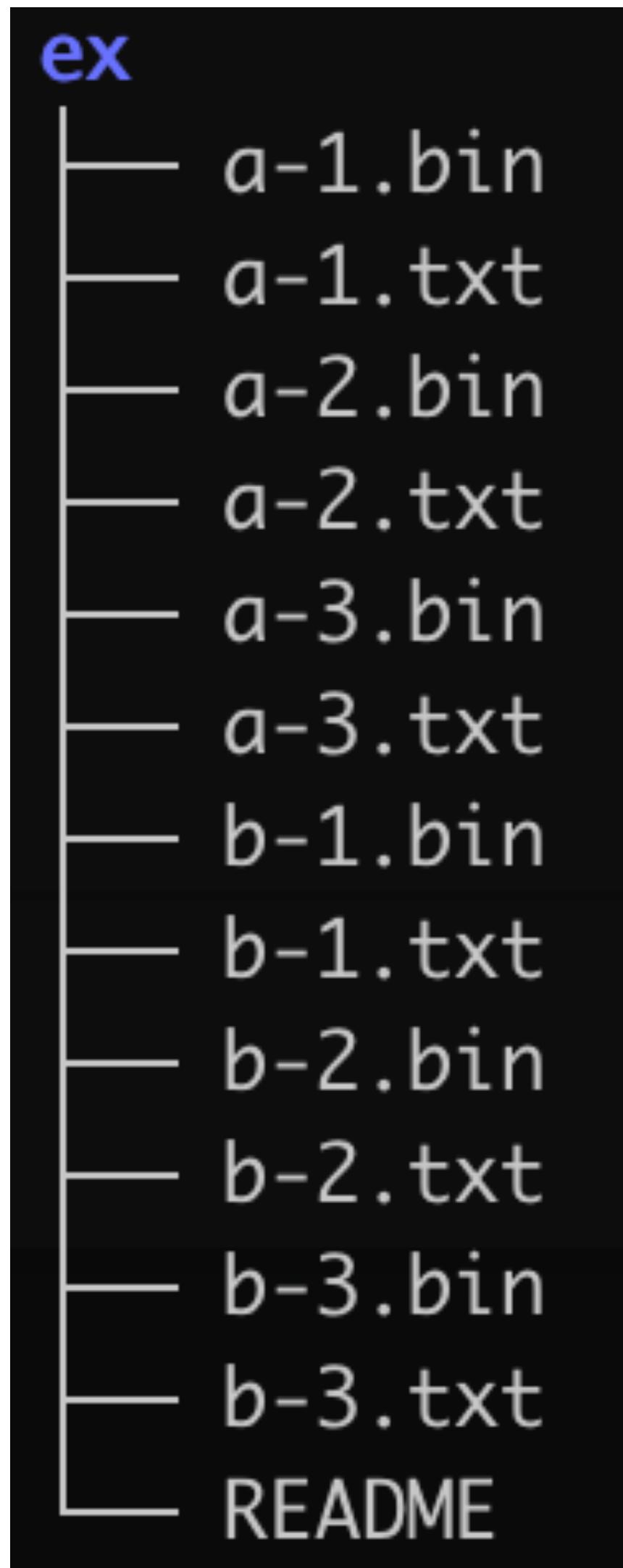
NAME

      cp – copy files and directories

SYNOPSIS

      cp [OPTION]... [–T] SOURCE DEST

      cp [OPTION]... SOURCE... DIRECTORY

      cp [OPTION]... –t DIRECTORY SOURCE...

DESCRIPTION

      Copy SOURCE to DEST, or multiple SOURCE(s) to DIRECTORY.

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

A. `$ cp a/b/[a-z].rs  /tmp`

B. `$ cp a/*/*.rs  /tmp`

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

D. `$ cp a/b/?.rs  /tmp`

E. `$ cp a/b  /tmp  *.rs`

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

Standard input/output/error

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

Standard input/output/error

Standard input/output/error

Normal output goes to stdout (file descriptor 1)

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

Standard input/output/error

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

Redirection

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

Redirection

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

`>>file` — redirect stdout to `file`, but append

Redirection

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

`>>file` — redirect stdout to `file`, but append

`<file` — redirect input (stdin) to come from `file`

Redirection

`>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

Redirection

`>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

‣ $ ls | wc

Redirection

`>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
- ‣ $ ls | wc

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

Redirection

`>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
‣ $ ls | wc

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

`2>&1` — redirect stderr to stdout

14

Redirection examples

Redirection examples

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

Redirection examples

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

$ cat <input.txt
```

Redirection examples

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

$ cat <input.txt

$ sort <input.txt >output.txt
```

Redirection examples

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

$ cat <input.txt

$ sort <input.txt >output.txt

$ ps -ax | grep bash
```

Redirection examples

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

$ cat <input.txt

$ sort <input.txt >output.txt

$ ps -ax | grep bash

$ grep hello file | sort | uniq -c
```

Redirection examples

```
$ 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
```

Redirection examples

```
$ 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

D. $ foo </dev/eof

E. $ echo | foo
```