Chapter 2 - Regular Expressions

1. What is a Regular Expression

- A **regular expression (regex)** is a pattern that describes a set of strings.
- Used extensively for pattern matching, text search, and substitution.
- Common in:
  - Unix tools: `[[`, `expr`, `test`, `vi`, `grep`, `sed`, `emacs`
  - Programming languages: C, awk, tcl, Perl, Python
  - Compilers and parsing tools (`scanf`, lexers, tokenizers)

---

2. String Matching in Bash

- Syntax: `[[ string =~ pattern ]]`
  - `pattern` is an **Extended Regular Expression (ERE)**.
  - Returns 0 if match succeeds, 1 otherwise.

Examples:

```
[[ "test" =~ es ]]     # true
[[ "test" =~ ^es ]]    # false
```

---

3. Fundamentals of Regular Expressions

- A regex matches if the **pattern appears as a substring**.
- **Atomic unit:** single-character match.
- **Literal example:** `cat` matches exactly “cat”.
- Regex can be **concatenated** and combined with **metacharacters** to form complex patterns.

---

4. BRE vs ERE Syntax Comparison

Feature	BRE (Basic)	ERE (Extended)
Ordinary characters	Match themselves	Match themselves
Start / End of line	`^` / `$`	Same
Any character	`.`	Same
Character class	`[a-z]`, `[^a-z]`	Same
Zero or more	`r*`	Same
One or more	N/A	`r+`
Zero or one	N/A	`r?`
Alternation	N/A	`r1
Grouping	`\(r\)`	`(r)`
Repetition	`\{n,m\}`	`{n,m}`

---

5. Character Classes

- Allow matches for **specific sets of characters**.
- Examples:
  - `[aeiou]` → any vowel
  - `[kK]orn` → matches “korn” or “Korn”
- **Ranges:**
  - `[1-9]` same as `[123456789]`
  - `[a-e1-9]` combines letters and digits
  - `[-123]` matches the literal `-`, `1`, `2`, or `3`

---

6. Negated Character Classes

- Syntax: `[^...]`
- Matches any character **not** listed.
- Example:

`b[^eo]at` → matches “brat” but not “beat” or “boat”.

---

7. Named Character Classes

- More portable than explicit ranges.
- Syntax: `[[:name:]]`

Named Class	Equivalent
`[:alpha:]`	`[A-Za-z]`
`[:alnum:]`	`[A-Za-z0-9]`
`[:lower:]`	`[a-z]`
`[:upper:]`	`[A-Z]`
`[:digit:]`	`[0-9]`
`[:punct:]`	Punctuation
`[:cntrl:]`	Control characters

---

8. Match Length and Greedy Matching

- Regex matches the **longest possible** substring.
- Example:

  Pattern `a.*e` on “Scrapple from the apple.”

  → Matches from the first `a` to the last `e`.

---

9. Repetition Ranges `{n,m}`

- `{n}` → exactly n times
- `{n,}` → at least n times
- `{,m}` → at most m times
- `{n,m}` → between n and m times

Examples:

- `.{0,}` = `.*`

- `a{1,}` = `aa*`

---

10. Subexpressions `( )`

- Group expressions so that quantifiers apply to the whole group.
- Examples:
  - `a*` → matches "", "a", "aa", ...
  - `abc*` → matches “ab”, “abc”, “abcc” ...
  - `(abc)*` → repeats “abc”
  - `(abc){2,3}` → matches “abcabc” or “abcabcabc”

---

11. Anchors

- Used to match positions in text.
  - `^` → start of line
  - `$` → end of line
- Examples:
  - `^b[eor]at` matches “beat” only at start of a line.
  - `b[eor]at$` matches “boat” only at end of a line.

---

12. Backreferences

- Refer back to **previously matched groups**.
- Syntax: `\1`, `\2`, etc.
- Example:

  `([a-zA-Z]{1,}) .* \1` → finds duplicate words (e.g., “one is one”).

---

13. Escaping Special Characters

- Special characters: `. * ^ $ ( ) { } \ | ? +`

- Precede with `\` to match literally.

Examples:

```
grep 'a\.jpg' file     # matches literal a.jpg
grep 'a[jp]g' file     # matches ajg or apg
grep 'a\[jp\]g' file   # matches literal a[jp]g
```

---

14. Regular Expression Standards

Standard	Description
BRE	POSIX Basic Regular Expression (`\( \)` , `\{ \}`)
ERE	POSIX Extended (`( )` , `{ }` , supports `+` , `?` , `
PCRE	Perl-Compatible; used in Perl, Python, vim; supports richer syntax

---

15. Practical ERE Examples

Pattern	Meaning
`[a-zA-Z_][a-zA-Z_0-9]*`	Valid C variable name
`\$[0-9]+(\.[0-9][0-9])?`	Dollar amount with optional cents
`(1[0-2]	[1-9]):[0-5][0-9](am
`<[hH][1-4]>`	HTML header tags `<h1>` through `<H4>`

---

16. Example Exercises

1. `echo bbbbcccc | grep -E "b{3,4}c{1,3}"` → prints **bbbbccc**
2. Regex for lines not starting with lowercase and not ending with vowel:
   `grep -E '^[^a-z].*[^aeiou]$' document.txt`

---

17. The `grep` Command

Syntax:

`grep [-G|-E|-P] [options] regex [files]`

- **Standards supported:**
  - BRE (default): `grep -G`
  - ERE: `grep -E`
  - PCRE: `grep -P`

Option	Description
`-h`	Suppress filenames
`-i`	Ignore case
`-l`	List filenames only
`-n`	Display line numbers
`-v`	Invert match
`-o`	Print only matched portion

18. `grep` Examples

```
echo 'mechanism' | grep 'me'
grep 'fo*' file.txt
grep -E 'fo+' file.txt
grep -E -n '[Tt]he' file.txt
grep -E 'NC+[0-9]*A?' file.txt
grep -E '[-+][0-9]+\.?[0-9]*' *.c   # Find signed numbers
```

19. The `sed` Command (Stream Editor)

- Non-interactive, processes one line at a time.
- Syntax: `sed 'script' file`

Common Commands

Command	Action
`a`	Append below current line
`c`	Change current line

Command	Action
`d`	Delete line
`i`	Insert above current line
`p`	Print line
`r`	Read from file
`s`	Substitute (search and replace)
`w`	Write to file

Examples

```
sed 's/unix/linux/' geekfile.txt     # Replace first match
sed '2d' geekfile.txt                # Delete 2nd line
```

Options

- `f scriptfile` → load from file
- `e script` → multiple inline commands
- `n` → suppress output
- `/regex/` → apply only to matching lines
- Example: `2,4d` deletes lines 2 through 4.

---

20. The `awk` Command

- Processes structured text files by splitting lines into fields.
- Syntax:

```
awk 'pattern { action }' inputfile
awk -f scriptfile inputfile
```

Key Variables

- `$1`, `$2`, ..., `$0` → individual fields / entire line
- `FS` → input field separator

- `OFS` → output field separator
- `NR` → record number (line count)

---

21. `awk` Print and Printf Examples

```
ls -l | awk '{print $5, $9}'              # size, name
ls -l | awk '{print "Size is " $5 " bytes for " $9}'
ls -l | awk '{printf "Size is %10d for %s\n", $5, $9}'
```

---

22. `awk` Patterns and Actions

- Patterns can be **regex**, or special blocks `BEGIN` and `END`.

Examples

```
ls -l | awk 'BEGIN{print "Files found:\n"} /[ax].*\.conf$/ {print $9}'
ls -l | awk '/[ax].*\.conf$/ {print $9} END{print "--- end ---"}'
awk 'BEGIN{count=0} /[ax].*\.conf$/ {count++} END{printf "%d files found
\n", count}'
```

---

23. Key Takeaways

- Regex defines **pattern-based string operations** used across many UNIX tools.
- Differences between **BRE, ERE, and PCRE** lie mainly in syntax and available metacharacters.
- Tools like `grep`, `sed`, and `awk` extend regex for powerful text processing.
- Understanding grouping, quantifiers, and character classes is essential for precision.

---

24. Important Topics for Quick Review

1. **Regex Basics**

- What regex is and where it's used.
- How matching works (`[[ string =~ pattern ]]`).

2. **Metacharacters**
   - `^` , `$` , `.` , , `+` , `?` , `{n,m}` , `|` , `()` , `[]`
3. **BRE vs ERE**
   - Escaping rules and syntax differences.
4. **Character Classes**
   - [ ] `[ ]` , `[^ ]` , ranges, and named classes like `[:digit:]` .
5. **Greedy Matching**
   - Regex matches longest substring by default.
6. **Grouping and Backreferences**
   - Using `( )` and `\1` , `\2` for repeated matches.
7. **Regex in Tools**
   - `grep` (search), `sed` (edit), `awk` (analyze).
8. **Common Patterns**
   - Variable names, currency formats, times, HTML tags.
9. **Command Usage**
   - `grep -E` , `sed 's/pattern/replacement/'` , `awk 'pattern {action}'` .
