Pattern Matching, Regular Expressions, and Emacs

1. Globbing (Shell Pattern Matching)

Globbing is a simple, fast pattern-matching syntax used by shells (e.g., bash) for matching file names. It is related to, but simpler and more limited than, regular expressions. Globbing is used in commands like `ls`, `echo`, `rm`, and in shell control structures like `case`.

- **Globbing matches only filename components** (parts of a path between slashes). Globs never match `/`.
- **Dot files** (files starting with `.`) are not matched by `*` or `?` unless the pattern itself starts with a dot.
- If a glob pattern matches nothing, the shell usually passes the pattern unchanged to the command.

Globbing vs Regular Expressions

Feature	Globbing (Shell)	Regular Expressions (Regex)
Used for	Filenames	Text/string matching
Wildcards	`*`, `?`, `[abc]`	`.`, `*`, `+`, `?`, `[]`, `()`
Matches `/`?	Never	Can match `/` (unless told not to)
Syntax	Simple	More powerful, complex
Negation	`[!abc]`	`[^abc]`
Quoting needed	To prevent expansion	To prevent shell expansion

Special Characters and Syntax

Symbol	Meaning
`*`	Matches any sequence of zero or more characters
`?`	Matches exactly one character
`[abc]`	Matches any one character in the set (here: 'a', 'b', or 'c')
`[a-z]`	Matches any character in the range (here: 'a' to 'z')
`[!abc]`	Matches any character NOT in the set (negation)

- Use `-` at the end of a bracket expression to match a literal `-`.
- To match a literal `]`, put it first in the set: `[]abc]` matches `]`, `a`, `b`, or `c`.
- To match a literal `!`, put it anywhere but first.

Globbing and Directory Structure

- Globs never match `/` (directory separator).
- Patterns like `*/foo*` match `foo*` in direct subdirectories only.
- To match files two levels down: `*/*/foo*`.

Dot Files and Globbing

- `*` and `?` do **not** match files starting with `.` unless the pattern itself starts with a dot.
- Example: `echo *` will not show `.hidden` files.
- To match hidden files: use a pattern like `.??*` (matches hidden files with at least three characters).

Globbing Negation

- `[!abc]*` matches files not starting with `a`, `b`, or `c`.
- `[!abc.]*` matches files not starting with `a`, `b`, `c`, or `.`.

Globbing Examples

Pattern	Description	Matches
`*`	All files except hidden ones	`foo`, `bar.txt`
`?.txt`	One character followed by `.txt`	`a.txt`
`[abc]*`	Files starting with `a`, `b`, or `c`	`apple`, `cat`
`[!abc]*`	Files not starting with `a`, `b`, or `c`	`dog`

Pattern	Description	Matches
`*.bash`	Files ending with `.bash`	`script.bash`
`.??*`	Hidden files with name length ≥ 3	`.xrc`

Example Directory

Suppose directory contains: `foo`, `bar`, `.hidden`, `a.txt`, `a-b.txt`

Command	Matches
`echo *`	foo, bar, a.txt, a-b.txt
`echo .*`	.hidden, .., .
`echo ?.txt`	None (needs 1-char name)
`echo [a-c]*`	bar
`echo [!abc]*`	Anything not starting a/b/c

Globbing and Slashes

- Patterns never match `/`.
- A trailing slash in a pattern (e.g., `foo/`) requires the match to be a directory.

Globbing and Quoting

- If you want to prevent the shell from expanding a glob, quote it: `'[a-z]*'`.
- Quoting is also necessary to prevent the shell from interpreting special characters in regex patterns.

Why "glob"?

- The original function implementing this was called `glob`.

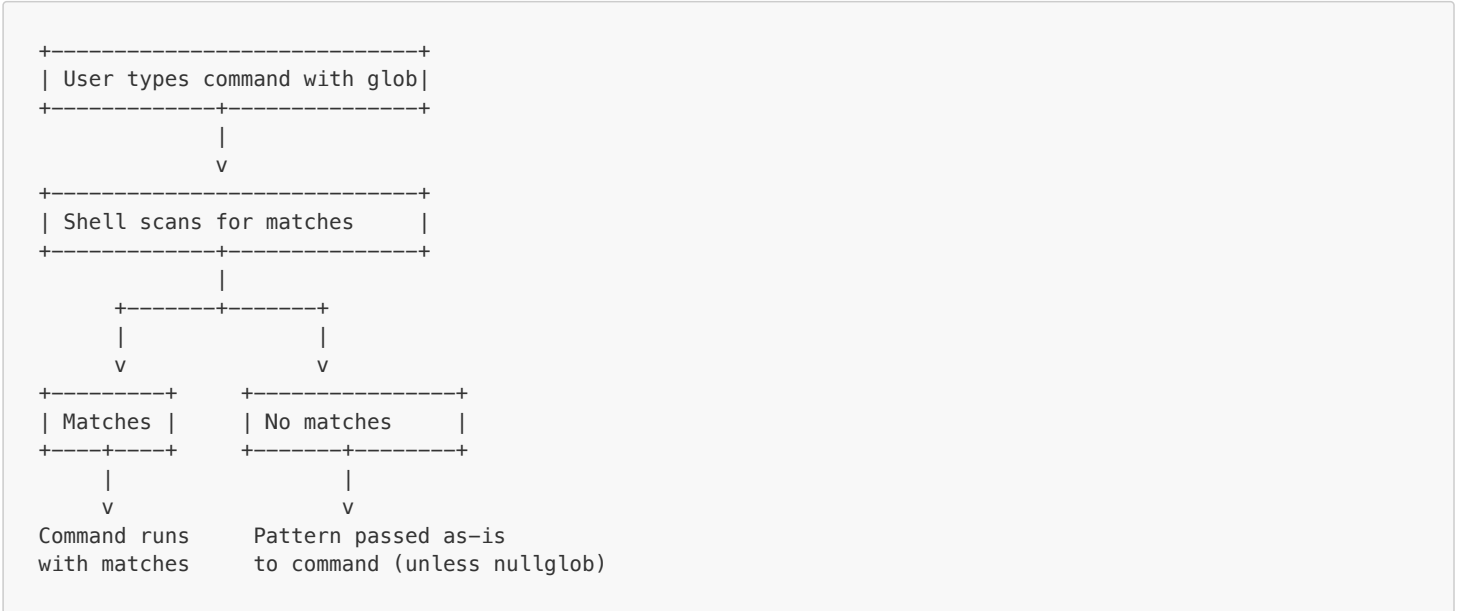
Escaping Special Characters in Globs

- To match a literal `*`, `?`, or `[`, you can escape them with a backslash (e.g., `\*`, `\?`, `\[`).
- Example: `echo file\*` matches the file named `file*` (not all files starting with `file`).

Shell Options Affecting Globbing

- `shopt -s dotglob`: Makes `*` and `?` match dotfiles.
- `shopt -s nullglob`: Expands globs to nothing if no match (instead of passing the pattern unchanged).

Globbing Expansion Flowchart



---

2. Shell I/O Redirection

Shells use file descriptors for I/O:

Descriptor	Description
0	Standard Input
1	Standard Output
2	Standard Error

Redirection Syntax

Syntax	Meaning
`>`	Redirect stdout to file (overwrite)
`>>`	Redirect stdout to file (append)
`2>&1`	Redirect stderr to where stdout points
`3< file`	Open file on descriptor 3 for reading
`3> file`	Open file on descriptor 3 for writing
`3<> file`	Open file for reading and writing on descriptor 3
`3>&1 1>file`	Save current stdout on fd 3, then redirect stdout

Here Documents

A "here document" feeds a block of text as stdin to a command:

```
cat <<EOF
line one
line two
EOF
```

- Variable expansion occurs unless the delimiter is quoted: `<< 'EOF'` prevents expansion.

File Descriptor Redirection Diagram

```
[stdin 0] <---+         +---> [stdout 1] ---> file or terminal
              |         |
              |         +---> [stderr 2] ---> file or terminal
              |
         [other fds: 3,4,...]
```

Common Redirection Idioms

Idiom	Meaning
`command > file`	stdout to file (overwrite)
`command >> file`	stdout to file (append)
`command 2> file`	stderr to file
`command > out 2>&1`	stdout and stderr to same file
`command < file`	stdin from file
`command 2>&1 >file`	stderr to old stdout, then stdout to file

Using Custom File Descriptors in Scripts

```
exec 3>log.txt   # Open log.txt for writing on fd 3
echo "log entry" >&3
exec 3>&-        # Close fd 3
```

---

3. Shell Scripting: Exit, Return, Functions, Scripts, Aliases

Exit and Return

Command	Effect
`exit`	Terminates shell or script
`exit 1`	Exits with status code 1
`return`	Exits from a shell function only

- `exit` ends the shell; `return` only exits a function.
- Exit status of last command: `$?`

Shell Functions vs. Shell Scripts

Shell Function

```
g() {
  grep "$@"
}
```

- Defined in `.profile` or interactively
- Lightweight, runs in current shell
- Not visible to other programs

Shell Script

File `g`:

```
#!/bin/bash
grep "$@"
```

- Saved in a directory in `$PATH`
- Executed as a new process
- Visible to all programs

Feature	Function (in-shell)	Script (file)
Scope	Local to shell	Global (all programs)
Overhead	Low	High (new process)
Portability	Low	High
Speed	Fast for small	Better for large

Aliases

- For simple substitutions only:

```
alias g='grep'
```

- Not suitable for complex logic; use functions instead.

Sourcing Scripts

- Use `source script.sh` or `. script.sh` to run a script in the current shell (variables/functions persist).

Variable Scope Example

```
myfunc() {
  local x=5
  echo "x in function: $x"
}
x=10
myfunc
# x outside function is still 10
```

Script/Function/Alias Execution Flowchart

```
+--------------------+
| User enters name   |
+---------+----------+
          |
          v
+--------------------+
| Is it an alias?    |--yes--> expand and run
|        |           |
|        no          v
|             Is it a function? |--yes--> run in current shell
|        |           |
|        no          v
|             Is it a script in $PATH? |--yes--> run as new process
|        |           |
|        no          v
|             Command not found
```

---

4. Regular Expressions (Regex)

Regular expressions are a "little language" for string pattern matching, used in tools like `grep`, `sed`, `awk`, Python, etc. There are multiple regex syntaxes (ERE, BRE, Perl, Python, etc.)—be careful to use the right one for your tool.

Extended Regular Expressions (ERE)

- Use with `grep -E` or `egrep`.

Core Syntax and Operators

Operator	Description
`.`	Any character except newline
`*`	0 or more repetitions
`+`	1 or more repetitions
`?`	0 or 1 occurrence (optional)
`|`	Alternation (OR)
`()`	Grouping (changes precedence)
`{n}`	Exactly n occurrences
`{n,m}`	Between n and m repetitions
`^`	Start of line anchor
`$`	End of line anchor
`[]`	Bracket expressions (character classes)

Operator Precedence (Highest to Lowest)

1. `*`, `+`, `?`, `{}`
2. Concatenation
3. `|` (alternation)

Bracket Expressions

- `[abc]`: matches 'a', 'b', or 'c'
- `[a-z]`: matches any lowercase letter
- `[^abc]`: matches any character except 'a', 'b', or 'c'
- `[[:alpha:]]`: matches any alphabetic character (locale-aware)
- To include `-`, put it at the end: `[a-z-]`
- To include `]`, put it first: `[]abc]`
- To include `^`, put it anywhere but first
- Inside brackets, `-` denotes range unless at start or end

- Backslash is NOT special inside brackets

Special bracket tricks

- `[.]`, `[-]`, `[]]` — see above for how to include these literally
- `[[:alnum:]]`, `[[:digit:]]`, `[[:print:]]` — POSIX character classes

Examples

Pattern	Matches Example
`abc`	Only the string 'abc'
`a.b`	'a' followed by any char, then 'b'
`a*`	zero or more 'a's
`a+`	one or more 'a's
`(ab|cd)+`	'ab' or 'cd', repeated
`^xyz$`	entire line must be 'xyz'
`^(.)(.)(.).\3\2\1$`	six-character palindromes

Anchors

- `^` at start: match only at beginning of line
- `$` at end: match only at end of line

Quoting and Escaping

- Backslash escapes special characters: `\*`, `\.`, `\(`, etc.
- To match `\`, often need `\\` (shell and regex both interpret backslash)
- Be careful: the shell may expand or interpret backslashes before passing to grep
- Always quote regex patterns in the shell to avoid globbing: `'[a-z]*'`

Common Pitfalls

- A regex with just a backslash (`\`) is invalid
- Quoting and escaping can be tricky—test your patterns

Bracket Expression Edge Cases

- To include `-` literally, put it at the start or end: `[-a-z]` or `[a-z-]`
- To include `]` literally, put it first: `[]a-z]`
- To include `^` literally, put it anywhere but first
- Ranges like `[z-a]` are invalid
- `[a-]` matches 'a' or '-'
- `[a-z-]` matches 'a' to 'z' and '-'

ASCII Diagram: Bracket Expression Rules

```
[ ^ ]
 |   |
 |   +-- If first after [, means negation
 +------ If not first, just a literal ^

[ - ]
 |   |
 |   +-- If at start or end, literal -
 +------ Else, range

[ ] ]
 |   |
 |   +-- If first after [, literal ]
 +------ Else, ends bracket expression
```

Basic Regular Expressions (BRE)

- Used with `grep` (no `-E`)

- Fewer metacharacters are special: `+`, `?`, `{}` are NOT special
- Use `\(...\)` for grouping
- Use `\{n,m\}` for repetition
- No alternation (`|`) operator

Backreferences (BRE only)

- `\1`, `\2`, ... refer to the nth parenthesized group
- Example: `\(abc\)\1` matches 'abcabc'
- Example: `^\(\.\)\(\.\)\2\1$` matches 4-character palindromes
- Backreferences are slow and non-regular—avoid if possible

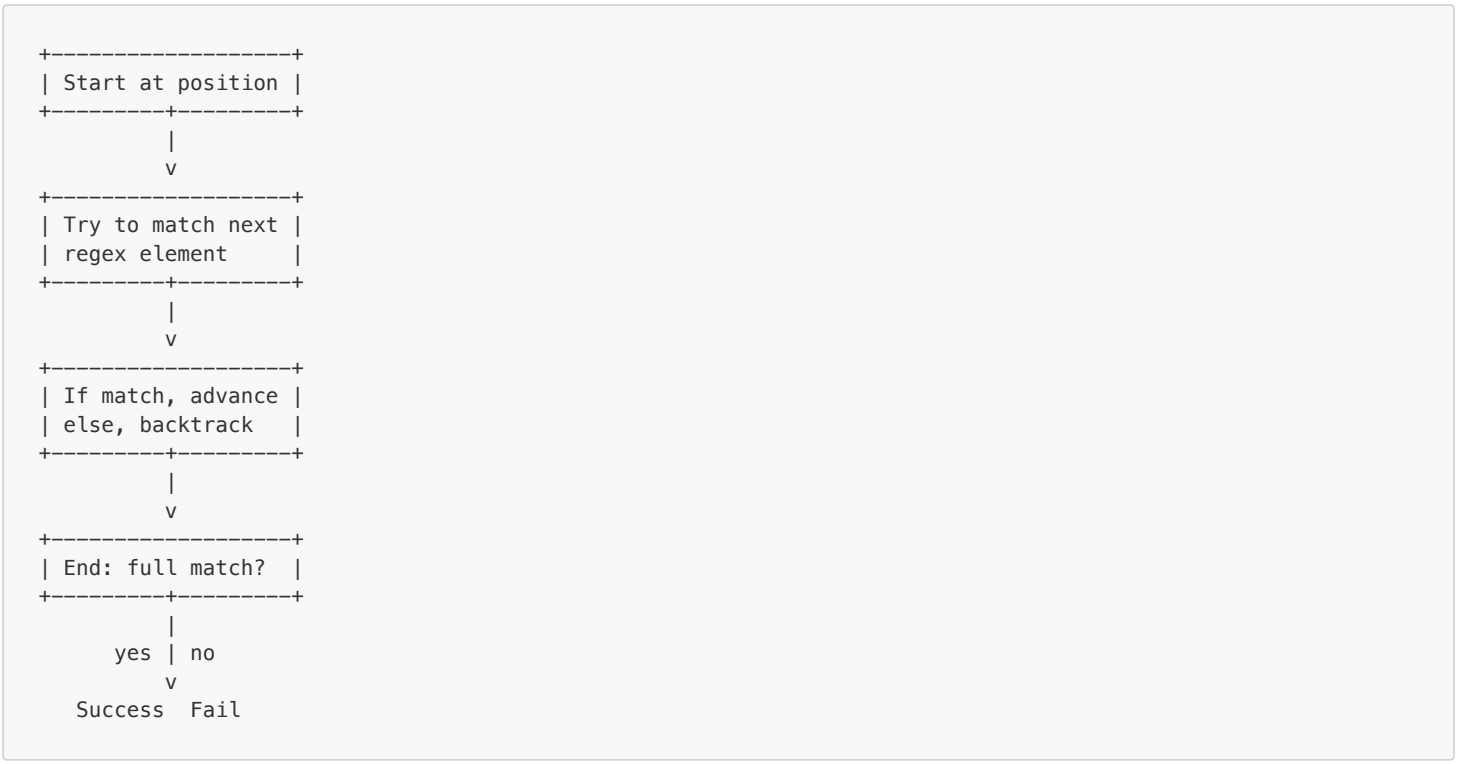
Example Table

Pattern	Description
`\(abc\)\1`	Matches 'abcabc'
`\(^a.*b$\)`	Entire line starting with a, ending with b
`^\(.\)\(.\)\2\1$`	Matches 4-character palindromes

Regex Syntax Comparison

Feature	ERE (`grep -E`)	BRE (`grep`)	Perl/Python	Globbing
Grouping	`()`	`\(\)`	`()`	N/A
Alternation	`|`	N/A	`|`	N/A
`+`, `?`, `{}`	Yes	No (use `\{}`)	Yes	N/A
Backrefs	N/A	`\1`, `\2`	`\1`, `\g<1>`	N/A
Wildcard	`.`	`.`	`.`	`*`, `?`
Negation	`[^abc]`	`[^abc]`	`[^abc]`	`[!abc]`

Regex Matching Process Diagram



Greedy vs Non-Greedy Quantifiers

- **Greedy**: Match as much as possible (default: `*`, `+`, `?`)
- **Non-greedy (lazy)**: Match as little as possible (Perl/Python: `*?`, `+?`, `??`)
- Example (Python):

```python
import re
s = 'aaaab'
print(re.match(r'a+?', s).group())  # 'a' (non-greedy)
print(re.match(r'a+', s).group())   # 'aaaa' (greedy)
```

Multiline Matching Example (Python)

```python
import re
text = """foo\nbar\nbaz"""
print(re.findall(r'^b.*', text, re.MULTILINE))  # ['bar', 'baz']
```

Example: Regex in Python

```python
import re
pattern = r'\b\w{3}\b'
text = 'cat dog bird fish'
print(re.findall(pattern, text))  # ['cat', 'dog']
```

---

5. Emacs: Keyboard-Driven Text Editing

Emacs is a highly extensible, keyboard-driven text editor. Its philosophy is to maximize efficiency by minimizing hand movement from the keyboard. It is modular, with major and minor modes, and features a powerful mini-buffer for commands and scripting.

Core Concepts

- **Buffers**: In-memory views of files, outputs, or shell sessions
- **Windows**: Viewports into buffers (not OS windows)
- **Kill Ring**: Stores multiple cut/copied text entries; cycle with `M-y` after `C-y`

Essential Commands

Command	Action
`C-x C-f`	Open file
`C-x C-s`	Save file
`C-g`	Cancel current command
`C-SPC`/`C-@`	Set mark at cursor (start selection)
`M-w`	Copy region to kill ring
`C-w`	Cut (kill) region
`C-y`	Yank (paste) last kill ring
`M-y`	Cycle through kill ring after yank
`C-x C-x`	Exchange point and mark
`C-x b`	Switch buffer
`C-x C-b`	List all buffers
`C-x o`	Switch windows
`C-x 2`	Split window horizontally
`C-x 3`	Split window vertically
`C-x 0`	Close current window
`C-x 1`	Maximize current window
`C-h k`	Describe key binding
`C-h m`	Describe current mode
`C-h i`	Info documentation browser

- **Meta key (`M-`)**: Usually `Alt` or `Esc`

Mini-buffer Operations

- `M-x`: Run Emacs command by name
- `M-:`: Evaluate Emacs Lisp
- `M-!`: Run shell command
- `M-|`: Run shell command with region as input

Examples

- `M-! date`: Run shell `date` command
- `M-| sort`: Sort selected region

Modes

- **Major modes**: Tailor Emacs for file type or buffer (e.g., Fundamental, Dired)
- **Minor modes**: Add auxiliary features (e.g., line numbers)
- Use `C-h m` to see active modes and key bindings

ASCII Diagram: Emacs Buffer/Window/Region

```
+-------------------------------+
|           Emacs Frame         |
| +------------+ +------------+  |
| |  Buffer 1 | |  Buffer 2 |  |
| +------------+ +------------+  |
|                               |
| [Kill Ring: C-w, C-y, M-y]    |
+-------------------------------+
```

Emacs Configuration Files

- User configuration is stored in `~/.emacs` or `~/.emacs.d/init.el`.
- You can customize Emacs by adding Emacs Lisp code to these files.

Example: Simple Emacs Lisp Function

```
(defun hello-world ()
  (interactive)
  (message "Hello, world!"))
```

- Run with `M-x hello-world`.

Kill Ring Cycling Diagram

```
[First kill] -> [Second kill] -> [Third kill]
      ^                              |
      |------------------------------|

C-y: yank most recent
M-y: cycle through previous kills
```