Theory of Automata – Lecture 2 Summary

Topic: Regular Expressions (REs)

---

1. What is a Regular Expression?

- A **Regular Expression (RE)** is a **mathematical way of defining language rules** using symbols and operators.
- It provides a compact way to describe **sets of strings** that belong to a formal language.

Real-world examples of Regular Expressions:

- Online forms (DOB must be in `yyyy-mm-dd` format)
- CNIC field (must be in `xxxxx-xxxxxxx-x` format)
- Password requirements (e.g., at least 1 capital letter, 1 digit, 8 characters)

✅ All these validations are done using **regular expressions** behind the scenes.
They check if an input string follows **specific rules**, returning:

- `True` → if valid
- `False` → if invalid

---

2. Role of RE in Compiler Design

A compiler performs **6 major phases**. The **first phase is the Lexical Analyzer**, which:

- Breaks code into **tokens**
- Uses **Regular Expressions** to define and match valid tokens

---

3. Symbols Used in Regular Expressions

Symbol	Meaning
`+`	One or more repetitions (at least once)
`*`	Zero or more repetitions (can include null/empty string)
`()`	Grouping
+	`Logical OR`

---

4. Examples with E = {a, b}

Using +:

- `a+` represents:
  E = {a, aa, aaa, aaaa, ...}
  (One or more repetitions of 'a')

**Using :*

- `a*` represents:
  E = {ε, a, aa, aaa, aaaa, ...}
  (Zero or more repetitions, including empty string)

  ✅ **Key Difference**:

- `+` → Requires at least one repetition.
- `*` → Allows even **no repetition** (null/empty string ε is valid).

---

5. Why Do + and * Cause Recursion?

- Both `+` and `*` imply **repetition**.
- These repetitions can go **infinitely deep**, e.g.:
  - `a*` → means `a` can repeat 0 to infinite times
- This idea of **self-referencing repetition** is a form of **recursion**.
  - Example: `a* = ε | a | aa | aaa | ...`
    You can define `a* = ε | a.a*`, which shows how the pattern includes **itself** again.

---

6. Practical Expression Examples

Example 1

E = {a, b}
Rules:

- Start with `a`
- End with `b`

Look at rules carefully and write them down in expression.

Write ‘a’ in start and ‘b’ at end. What combination can occur in the middle we are unsure of it.

So, we can write (a+b)* in the middle since * symbol shows that there can be nothing in between ‘a’ and ‘b’ and there can be any random combination of ‘a’ or ‘b’

RE: a(a+b)*b

Explanation:

- Start is fixed as a
- End is fixed as b
- Middle part is uncertain, but may include any combination of a and b.
  So we use (a+b)* in the middle.

Example 2

E = {0, 1}
Rules:

- Start with 0
- Ends with 11

RE: 0(0+1)*11

Explanation:

- Start is 0, end is 11
- Middle can have any combination of 0 and 1, so (0+1)*
- We use * instead of + to **allow the shortest valid string** (like 011) — since + wouldn't allow the middle part to be empty.

---

7. Reverse Problem – From RE to Language

Example:

RE: (aa + bb)*

Interpretation:

- The string is made up of **even numbers** of as or bs
- Accepts strings like:
  - ε (null)

- `aa`
- `bb`
- `aaaa`
- `bbbb`
- `aabb`
- `bbaa`
- `aabbaa`

- Does **not accept**: `a`, `b`, `ab`, `ba`, `aab`, etc.

---

8. More Practice Problems

Problem 1:

E = {0,1}
Rule: The 10th symbol from the right must be `1`

Explanation:
(0+1)*1 _ _ _ _ _ _ _ _ _ (now there are left 9 places)

we can do this: $(0+1)^9$ since there are nine places we can use expontential form to write it it will be seen as count not as actual exponent. We may also write it like this:

(0+1)*1 (0+1) (0+1) (0+1) (0+1) (0+1) (0+1) (0+1) (0+1) (0+1)

Problem 2:

E = {0,1}
Rule: Starts with either `0` or `1`

RE: `(0+1)(0+1)*`

Explanation:

- First character is either `0` or `1`
- Rest of the string (if any) can be any combination of `0`s and `1`s

---

Problem 3:

E = {a, b}
Language L = {a^{2n} b^{2m+1} | n > 0, m ≥ 0}

Explanation:

- `a²ⁿ` means even number of a's, starting from at least 2 (n > 0)
- `b²ᵐ⁺¹` means **odd** number of b's (since `2m + 1` is always odd)
- At least one `b` must appear

RE:

`(aa)⁺(bb)*b`

Problem 4:

L = {a^n b^m | n ≥ 4, m ≤ 3}

RE:

`aaaa a* (ε + b + bb + bbb)`

Explanation:

- `aaaa` ensures minimum 4 `a`s
- `a*` allows any number of additional `a`s (n ≥ 4)
- `(ε + b + bb + bbb)` allows **0 to 3** `b`s (m ≤ 3)

Summary of Key Takeaways

Concept	Description
a+	One or more `a`s
a*	Zero or more `a`s
(a+b)	Either `a` or `b`
(a+b) *	Any string (including null) made from `a` and `b`
RE Usage	Used in Lexical Analysis (first compiler phase) to tokenize code
Real-world	Used in forms, CNICs, passwords for validation
