

# Preparations

## 🌟 1. Alphabets ( $\Sigma$ )

- **Definition:** A finite **set of symbols**.
- **Notation:** Usually written as  $\Sigma$  (sigma).
- **Example:**
  - $\Sigma = \{0, 1\} \rightarrow$  Binary alphabet
  - $\Sigma = \{a, b, c, \dots, z\} \rightarrow$  English alphabet

🧠 Think of an alphabet as the **basic building blocks**.

---

## 🌟 2. Strings (Words)

- **Definition:** A **finite sequence** of symbols from an alphabet.
- **Notation:** Written as **w** (or any variable).
- **Example:**
  - If  $\Sigma = \{a, b\}$ , then `abba`, `aab`, `b` are **strings**.
  - `aaabbbb` is a string of length 6.

💡 A string can be empty too!

---

## 🌟 3. Empty String ( $\epsilon$ )

- **Symbol:**  $\epsilon$  (Greek letter epsilon)
  - **Meaning:** A string with **zero symbols**.
  - **Length:**  $|\epsilon| = 0$
- 

## 🌟 4. Length of a String

- **Notation:**  $|w|$  = number of symbols in string **w**
- **Examples:**
  - $w = \text{abc} \Rightarrow |w| = 3$

- $w = \text{aabb} \Rightarrow |w| = 4$
- $w = \varepsilon \Rightarrow |\varepsilon| = 0$

## ★ 5. Set of All Strings ( $\Sigma^*$ )

- $\Sigma^*$  = Set of **all possible strings (including  $\varepsilon$ )** over  $\Sigma$ .
- **Includes:**
  - $\varepsilon$  (empty string)
  - All strings of length 1: **a** , **b** , **c** , ...
  - All strings of length 2: **aa** , **ab** , **bc** , ...
  - And so on...

### Example:

If  $\Sigma = \{a, b\}$ , then

$\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, \dots\}$

## ★ 6. Words vs Strings

- In Automata Theory, **word = string**.
- Both mean a **finite sequence** of symbols from  $\Sigma$ .
- No difference, just terminology.

### Summary Table:

Term	Definition	Example		
<b>Alphabet (<math>\Sigma</math>)</b>	Set of symbols	$\{0,1\}$ or $\{a,b\}$		
<b>String</b>	Sequence of symbols from $\Sigma$	<b>abba</b> , <b>101</b>		
<b>Empty String</b>	A string with no symbol	$\varepsilon$		
<b>Length</b>	Number of symbols in string		<b>abc</b>	= 3
<b><math>\Sigma^*</math></b>	All possible strings from $\Sigma$ (incl. $\varepsilon$ )	$\{\varepsilon, a, b, aa, ab, \dots\}$		

# ✓ 1. What is a String in Automata?

A **string** is just a **sequence of symbols** taken from a given alphabet ( $\Sigma$ ).

## abc Example:

Let's say:

$\Sigma = \{a, b\}$

Then:

- **a** → a string of length 1
- **ab** → a string of length 2
- **bba** → a string of length 3
- $\epsilon$  (empty string) → string of length 0

👉 **Symbols** must come from  $\Sigma$  only.

# ✓ 2. How to Count the Length of a String?

The **length** is simply the **number of symbols** in the string.

## 🔍 Examples:

String	Length
<b>a</b>	1
<b>bb</b>	2
<b>abba</b>	4
$\epsilon$	0
<b>babab</b>	5

📌 Count every **symbol**, not letters in English words, but symbols from the **alphabet**  $\Sigma$ .

## ✓ 3. What Do We Mean by "Tokenizing" in Automata?

In Automata, **tokenizing** a string often means:

Breaking it into **individual symbols** from the alphabet.

### 🧠 Example:

Let  $\Sigma = \{0, 1\}$

Then:

- String `010` is made up of tokens: `0`, `1`, `0`  
→ 3 symbols → Length = 3

Even if it looks like one chunk, it's really:

→ `[0]` `[1]` `[0]`

---

## ⚠ Common Confusion:

### ✗ Mistake:

You see a word like `"dog"` and count it as 1 word, so you think:

Length = 1 ✗

But if  $\Sigma = \{d, o, g\}$ , then:

✓ `"dog"` = `d`, `o`, `g` → Length = 3

✓ Each **symbol/character** is counted separately.

---

## ✓ Key Point:

Length = Number of symbols from  $\Sigma$  in the string.

No matter what the word means in English!

---

# ☀️ Let's Practice Together

If I say:

- $\Sigma = \{a, b\}$
- String = `ababa`

👉 What is the length of this string?

---

## 🧠 Given:

- $\Sigma = \{B, aB, bab, d\}$
- $s = BaBbabBd$

You are asking:

Is this a string over  $\Sigma$ ? And if yes, what is its **length**?

---

## ✅ Step 1: Understand Each Symbol in $\Sigma$

These are your allowed **symbols** (not characters):

- `B` → a single symbol
- `aB` → a symbol (made of 2 characters)
- `bab` → a symbol (made of 3 characters)
- `d` → a symbol

So here, even though some symbols look like “long words”, they are treated as **one atomic unit**.

---

## ✅ Step 2: Tokenize the String

Now let's try to **split** `s = BaBbabBd` using only valid symbols from  $\Sigma$ .

Try breaking it down:

`s = B | aB | bab | B | d`

✅ All parts are in  $\Sigma$ :

- `B` → in  $\Sigma$
- `aB` → in  $\Sigma$
- `bab` → in  $\Sigma$
- `B` → in  $\Sigma$
- `d` → in  $\Sigma$

👉 So the string is **valid** over  $\Sigma$ .

---

## ✅ Step 3: Count the Length

We count the **number of symbols used**, not characters.

In this case:

Symbol	Count
<code>B</code>	1
<code>aB</code>	1
<code>bab</code>	1
<code>B</code>	1
<code>d</code>	1

1	2
3	4

 Length = 5

---

## ✅ Final Answer:

- `BaBbabBd` is a valid string over  $\Sigma = \{B, aB, bab, d\}$
- Length of the string = 5 symbols

---

## 🔥 Bonus Tip:

When  $\Sigma$  has **multi-character symbols**, your job is to:

1. Break the string into **valid symbols** from  $\Sigma$  (like tokenization).
2. Count how many **symbols**, not characters.

---

## ✓ Recap: What Is a Word in Automata?

In **Automata Theory**:

A **word** is simply a **string** made up of **symbols** from an alphabet  $\Sigma$ .

So:

- **Word = String**
  - Both mean the **same thing** in this context.
  - A word is **any valid sequence of symbols** from  $\Sigma$ .
- 

## 🧠 In Your Case:

You gave:

- $\Sigma = \{B, aB, bab, d\}$
- **String (or word) =** `BaBbabBd`

We already showed:

- This string can be tokenized as:  
→ `B` `aB` `bab` `B` `d`

So this string:

- Is made up **only of symbols from  $\Sigma$**  ✓
  - Therefore, it **is a valid word** over  $\Sigma$  ✓
  - Its **length** (number of symbols) = 5
- 

## ✓ Final Answer:

- `BaBbabBd` is a **word** over  $\Sigma = \{B, aB, bab, d\}$
  - It is also called a **string** in automata theory
  - The word contains **5 symbols** from  $\Sigma$
-

### Bottom line:

If a sequence can be made using only the **symbols in  $\Sigma$** , it is a **word** over  $\Sigma$ .

✓ Great observation — and **YES**, you're on the right track now.

Let me **clear it 100%**:

## Strings vs Words in Automata

Concept	Definition	Validity
String	Any sequence of <b>symbols or characters</b>	Can be valid or invalid
Word (over $\Sigma$ )	A string that is made <b>only from symbols in <math>\Sigma</math></b>	Always valid

### ✓ So YES:

- ♦ All words are valid strings over  $\Sigma$
- ♦ But **not all strings are words** over  $\Sigma$

## Example:

Let's say:

- $\Sigma = \{a, ab, c\}$

Now look at this string:

- abcab

Can we break it into valid symbols from  $\Sigma$ ?

Try:

1. a ✓
2. b ✗ ← b is not in  $\Sigma$



So:

- `abcab` is **not** a valid word over  $\Sigma$
  - It is a **string**, but **not a word** over  $\Sigma$
- 

## ✓ Valid Word Example:

Try: `aabcc`

Break it as:

- `a` ✓
- `a` ✓
- `b` ✗ ← again, invalid

So again: ✗ Not a word

Try: `abac`

- `ab` ✓
  - `a` ✓
  - `c` ✓ → ✓ Valid word
- 

## 🔄 Summary:

Term	Valid Over $\Sigma$ ?	Example
String	Maybe ✓ or ✗	<code>abc</code> , <code>xy</code> , <code>z</code>
Word	Always ✓	<code>abac</code> , <code>abab</code> , <code>c</code> (only if all symbols $\in \Sigma$ )

---

## ◆ Situation:

- $\Sigma = \{a, ab, c\}$
  - String = `abcab`
  - Goal: Check if `abcab` is a valid **word** over  $\Sigma$  using **tokenization**
-



# Step-by-Step: Tokenizing `abcab`

We need to **break it into valid symbols from  $\Sigma$** , which are:

```
{a, ab, c}
```

So we're allowed to pick:

- `a` ✓
- `ab` ✓
- `c` ✓

Let's try to tokenize `abcab` :

## ✓ Attempt 1: Greedy left-to-right

```
abcab
↑ try "ab"? Yes → match ✓
  ab | cab
    ↑ try "c"? Yes → match ✓
      ab | c | ab
        ↑ "ab"? Yes → match ✓
Final tokenization: ab | c | ab → all in  $\Sigma$  ✓
```

→ Yes! This tokenization works!

## ✓ Final Answer:

- `abcab` can be tokenized as `ab` , `c` , `ab`
- Each part is a valid **symbol from  $\Sigma$**
- So, `abcab` is a **valid word over  $\Sigma = \{a, ab, c\}$**



## Important Tokenizing Rule:

In Automata, we often use **greedy matching**:

Always try to match the **longest possible symbol** from left to right.

So:

- At `ab` , match `ab` (not `a` then `b` ) — because `b` alone is **not** in  $\Sigma$ .

## Atomic Value = Symbol in $\Sigma$

Each symbol from  $\Sigma$  (like `ab` , `a` , `c` ) is considered an **atomic unit**. You can't break it further when validating a word.

## Summary:

Term	Meaning
Atomic Value	Each indivisible symbol in $\Sigma$
Tokenizing	Breaking a string into symbols from $\Sigma$
Word	A string that can be tokenized with $\Sigma$ only

Awesome question! You're now stepping into the connection between **Automata Theory** and **Compiler Design** — specifically the **Lexical Analyzer (Lexer)**.

Let me explain how a **lexical analyzer** checks if a string is a **valid word** over a given  $\Sigma$  (**alphabet**) — just like you do in Automata.

## What is a Lexical Analyzer?

A **lexical analyzer** is the **first phase** of a compiler.  
It reads an **input string (source code)** and breaks it into **valid tokens (words)** based on predefined **lexical rules** (a kind of  $\Sigma$ ).

- ✓ The **tokens** must match the **patterns** defined in the **language grammar or symbol set ( $\Sigma$ )**.

# Lexical Analyzer vs Automata

Automata	Lexical Analyzer
Uses $\Sigma$ to define valid symbols	Uses token definitions (keywords, IDs, etc.)
Checks if a string is a valid word	Breaks source code into valid tokens
DFA (Deterministic Finite Automata)	Is built using DFAs internally

So a lexer is basically an **automaton (DFA)** that recognizes **valid words/tokens** from a stream of characters.








## PROCESS: How Lexical Analyzer Checks a Word

Let's go step by step using two examples:

### Example 1: Simple

- $\Sigma = \{\text{int, main, =, +, num, ;}\}$
- Input String:** `int num = num + num ;`

 Lexer should **tokenize** this as:

Token	Valid in $\Sigma$ ?
int	
num	
=	
num	
+	
num	
;	

 All tokens are valid →  This is a **valid word** (program statement) over  $\Sigma$

## ❌ Example 2: Complex / Invalid

- $\Sigma = \{\text{int, main, =, +, num, ;}\}$
- Input String: `int numb = num - num ;`

Try tokenizing:

Token	Valid in $\Sigma$ ?
int	✓
numb	❌
=	✓
num	✓
-	❌
num	✓
;	✓

⚠️ `numb` and `-` are **not in  $\Sigma$**

➡️ So lexer flags this string as **invalid** ❌

---

## 🧠 What's Happening Internally?

1. Lexer scans from **left to right**
2. Tries to match the **longest possible valid token**
3. Uses a **DFA** for each token definition (keyword, identifier, operator, etc.)
4. If it finds a symbol that doesn't match any DFA:
  - It reports an **invalid token error**

---

## 🤖 Automata Behind the Scenes

Each token is accepted by a **DFA**.

For example:

- `int` is accepted by DFA1
- `num` by DFA2

-  by DFA3, etc.

The lexer runs the **input through all DFAs** in parallel or in order to find the **longest valid match**.

---

## Summary: Lexical Analyzer vs Word Checking

Concept	Description
Alphabet ( $\Sigma$ )	Set of valid symbols/tokens
String	Sequence of characters (program code or sentence)
Tokenizing	Breaking string into symbols from $\Sigma$
Valid Word	A string that is fully made of valid tokens from $\Sigma$
Lexical Analyzer	DFA-based tool to tokenize and validate the input string

---

## Want to try?

Here's a challenge for you:

$\Sigma = \{\text{if, else, id, ==, =, +, ;}\}$

**String:** `if id == id + id ; else id = id + id ;`

Can you:

1. Tokenize this string
2. Check if all tokens are in  $\Sigma$ ?

Reply and I'll check it for you!

---

Of course! Here's your teacher's statement translated into **clear, simple language**, just like the explanation above:

---



## Original Statement:

“As when the string is scanned by lexical analyzer (compiler) first symbol 'B' is identified as a letter the lexical analyzer would not be able to identify, so while defining an alphabet it should be kept in mind that ambiguity should not be created.”



## Easy Explanation:

When the **compiler** (lexical analyzer) starts scanning the string from left to right, it sees the first symbol is **B** .

Now, because **B** , **Ba** , and **bab** are all in the alphabet ( $\Sigma$ ), the analyzer might get **confused** — should it pick just **B** , or try for **Ba** ?

If the lexer picks the wrong one first (like just **B** ), it may **fail to understand the rest of the string**.

So, to **avoid this confusion (ambiguity)**, when we **design our alphabet  $\Sigma$** , we should **not include overlapping symbols** like **B** and **Ba** .

This makes tokenization clear and avoids errors when recognizing words (valid strings).



## In One Line:

**Avoid using symbols in  $\Sigma$  that start with the same letters**, or it will confuse the compiler during scanning.

Absolutely! Let's add **reverse of a string** (according to Automata Theory) to your study summary — using **your example** too.



## Reverse of a String in Automata



## Definition:

The **reverse** of a string  $w$  , written as  $w^R$  , is the string obtained by **reversing the order of symbols** (not characters) in  $w$  .

---

## Important:

In Automata, when reversing:

- You reverse **symbols** from  $\Sigma$ , **not characters**
- So, tokenization **must come first**

---

## Example Based on Your Case:

**Given:**

- $\Sigma = \{B, Ba, bab, d\}$
- String  $s = BababB$

---

### Step 1: Tokenize

We use the correct tokenization (as discussed earlier):

```
s = (Ba)(bab)(B)
```

---

### Step 2: Reverse the Tokens

Now reverse the **symbols**, not characters:

```
Original: Ba | bab | B  
Reversed: B | bab | Ba
```

---

### Step 3: Final Reversed String:

```
Reverse(s) = BbabBa
```

 This is the **reversed string**, made using **symbols from  $\Sigma$** , in **reverse order**.





## Summary of Reversal

Original String	Tokenized as	Reversed Order	Final Reversed String
BababB	Ba, bab, B	B, bab, Ba	BbabBa



## Why Not Reverse Characters?

If you did `reverse("BababB")` by character, you'd get:

Original: B a b a b B  
Reversed: B b a b a B  $\rightarrow$  Bbabab

But this may not tokenize properly based on  $\Sigma$ .



## Automata Rule:

**Always reverse symbols (tokens), not characters**, especially when  $\Sigma$  contains multi-character symbols.