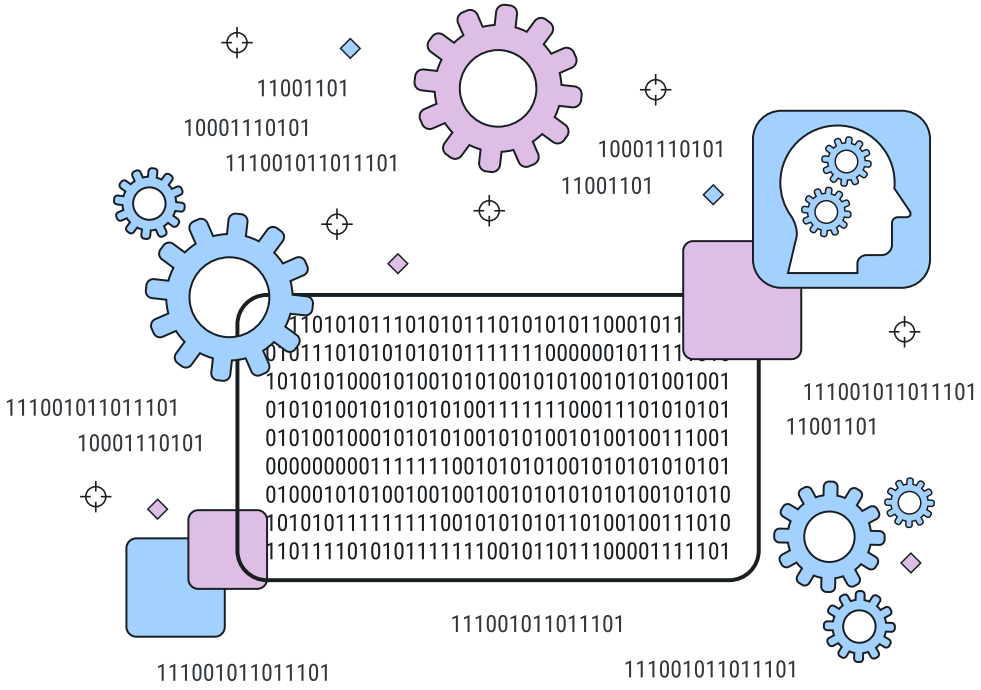


Theory of Computation
&
Compiler Design

CONTENTS

01. Symbol 01
02. Alphabets 01
03. String 01
04. Language 02
05. Power of Sigma 02
06. Finite Automata Model 03
07. State Finite Automata Model 03
08. DFA Model 04
09. How To Construct A DFA 06
10. Minimization of DFA 07
11. NFA Figure Construction 09
12. NFA to DFA Construction 10
13. Regular Expression 12
14. Regular Expression To NFA 14
15. Context-Free Grammar (CFG) 20
16. Language Processing System 23
17. Structure of a Compiler 24
18. First and Follow 32

Acknowledgment

With deep respect, I would like to express my heartfelt gratitude to my honorable madam, **Sumaia Rahman, Faculty of Computer Science and Engineering, Varendra University**, for conducting the Computation and Compiler Design lectures in class with great clarity, politeness, and excellent organization. Her insightful teaching and supportive approach inspired me to take detailed notes throughout the course.

I would also like to sincerely thank my friends from the 31st Batch—Tasmim Islam Tonni (ID: 223311035), Mst. Ayesha Muni (ID: 223311024), and Puja Shaha (ID: 223311014)—for their cooperation and continuous support in helping me prepare this note.

I hope these notes will be helpful for others in understanding the subjects more clearly.

Shah Md. Arshad Rahman Ziban
ID: 223311019
31st Batch, Department of CSE
Varendra University

Symbol

Defination

A symbol (also called a character) is the smallest building block in a system, which can be any alphabet, letter, number, or picture used to represent information or construct larger structures like strings or languages.

Examples of Symbols:

i. Alphabets (a to z): a,b,.....,z

ii. Digits (0 to 9): 0,1,2,....,9

Why uppercase letters aren't symbol?

Alphabets

Defination

An alphabet (also called an input in some contexts) is a finite set of distinct symbols, typically denoted by Σ (Greek letter sigma), used to construct strings and define languages. These symbols can be letters, digits, or special characters, and they serve as the basic building blocks for representing information in computational systems. The symbols in the alphabet are elements of the set.

Examples of Alphabets:

1. Σ={a,b,c}
   This is an alphabet containing three symbols: a,b and c.

2. Σ={0,1}
   This is an alphabet containing three symbols: 0 and 1.

3. Σ={a,b,c,0,1}
   This is an alphabet containing three symbols: a,b,c,0 and 1.

4. Σ={a,a,b,c}
   An alphabet must contain distinct symbols. In this case, adding a twice is redundant, and the set should be reduced to:

   Σ={a,b,c}

   This is an alphabet containing three symbols: a,b and c.

String

Defination

A string is a finite sequence of symbols (or characters) taken from an alphabet Σ. It can include any combination of symbols from the alphabet, and its length can vary from zero (an empty string) to any finite number of symbols.

Notation:

1. A string is typically denoted by a sequence of characters inside quotation marks, like w = "abc"
2. If Σ={a,b,c}, then w="abc" is a string made from those symbols.

Key Points:

1. Finite Sequence: A string is a sequence of symbols, and the length of the string is finite.
2. From an Alphabet: The symbols in a string come from a predefined alphabet Σ.
3. Empty String: The empty string, denoted by ε, is a string that contains no symbols. It has a length of zero.
4. Length of a String: The length of a string w, denoted as |w|, is the number of symbols it contains. For example, |"abc"|=3.

Language

Defination

A language is a set of strings formed from an alphabet that satisfies specific rules or properties.

Languages are two types: 1. Finite language 2. Infinite Language

A language is often denoted by L

Examples:

Σ = {a, b}
L1 = {W is a set of string where the length of each string must be 2}
= {aa, ab, ba, bb}
L2 = {W is a set of string where the length of each string must be 3}
= {aaa, aab, bab, bba, abb, aba, bbb, baa}
L3 = {W is a set of string where each string starts with 'a'}
= {a, ab, aa, aaa,,,,,,,,,,,,,,,,,,}

Here, L1 and L2 is called the finite language and L3 is infinite language.

Power of Sigma

Suppose, Σ = {a, b}

Σ^1= Set of all string over this Σ of the length is '1'
= {a, b}

Σ^2= Set of all string over this Σ of the length is '2'
= {aa, ab, ba, bb}

Note: Σ^0= empty string where the length of string is 0

$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots \ldots \ldots$
Σ* is called mother of all string.

Finite Automata Model

Defination

A Finite Automata Model (FA Model) is a mathematical model used to recognize patterns in input, such as strings.

Types

1. Deterministic Finite Automaton (DFA)
2. Nondeterministic Finite Automaton (NFA)

Why FA Model is used?

To give a finite representation of an infinite language

Example:

L(M) = {W is a set of string where each string starts with 'a'} and Σ={a,b}
W = {a, ab, aa, aaa,,,,,,,,,,,,,,,,,}

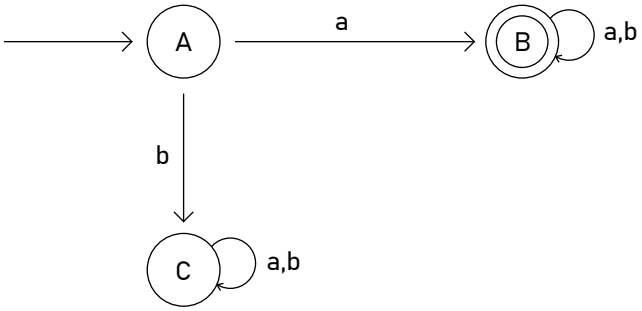


Figure: State Transition Diagram

This diagram accepts only those strings that start with 'a' and rejects those that start with 'b'. It is impossible to allocate infinite memory space for a language set. Therefore, a finite memory slot is used. For this reason, Finite Automata is implemented, as it requires very little memory and can efficiently determine whether a string is accepted or not.

States Finite Automata Model

States mean memory, which is denoted by a circle (◯) = {A, B, C}

1. Initial State

i. Start scanning or traveling.

ii. A DFA (Deterministic Finite Automaton) and an NFA (Nondeterministic Finite Automaton) both contain one initial state.

iii. This arrow signifies the starting point of the automaton. Additionally, there is no label (value) on the edge leading to the initial state

2. Final State

i. Stop scanning or traveling.

ii. Denoted by double circle (◎)

iii. DFA (Deterministic Finite Automaton) and NFA (Nondeterministic Finite Automaton), there can be one or more final states.

3. Trap State (or Dead State)

i. A trap state (or dead state) always has a self-loop for all input symbols.

ii. We cannot go to the final state from a trap state

iii. It is not mandatory to use a trap state in a DFA or NFA

iii. When a rejected string is processed, the automaton eventually reaches the trap state (also called the dead state).

4. Normal State

i. A state that handles input symbols and is not a final or trap state.

DFA Model

A DFA is a type of Finite Automaton (FA) that consists of five tuples:

$$M=(Q, \Sigma, \delta, q_0, F)$$

Example:

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

Define the language L(M) as the set of all strings that start with the letter 'a'.
Now, consider the state transition diagram for L(M):

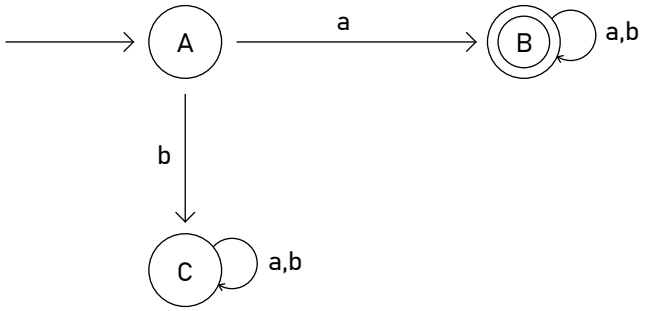


Fig: DFA Diagram

State Transition Diagram for L(M)

The five components of the DFA are described as follows:

Q - The set of all states: $Q = \{A, B, C\}$

Σ - The set of input alphabets: $\Sigma = \{a, b\}$

q_0 - The start (initial) state: $q_0 = A$

F - The set of final states: $F = \{B\}$

δ - The transition function

The transition function δ is defined as:

$$\delta : Q \times \Sigma \longrightarrow Q$$

The Transition Function (δ) in a DFA can be represented in two ways:

1. Transition Table (Tabular Form)
2. Using Equation

1. Transition Table (Tabular Form)

This method uses a table where rows represent states and columns represent input symbols. Each cell shows the next state for a given state-input pair.

State ↓ / Input →	a	b
A	B	C
B	B	B
C	C	C

Fig: DFA Transition Table

2. Using Equation

$$\delta : Q \times \Sigma \rightarrow Q$$

$$A \times a \rightarrow B$$
$$A \times b \rightarrow C$$
$$B \times a \rightarrow B$$
$$B \times b \rightarrow B$$
$$C \times a \rightarrow C$$
$$C \times b \rightarrow C$$

Characteristics of DFA:

1. For each input symbol, there is exactly one transition to a state.
2. A DFA has only one initial state.
3. A DFA can have more than one final state.

How To Construct A DFA

Problem 1

Construct a DFA which accepts the set of all strings over the Σ={a,b} where each string starts with an 'a'.

Solution

Alphabet: Σ={a,b}

Language: L(M)={a,ac,ab,aaa,aab,aba,abb,...}

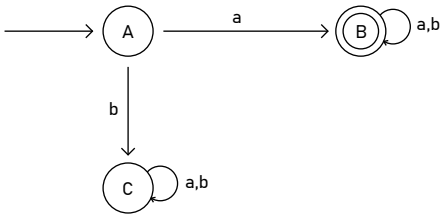


Fig: DFA Diagram

Problem 2

L(M)={w | w starts with 'aa' }

Solution

Alphabet: Σ={a,b}

Language: L(M)={aa,aaa,aab,aaab,aabb,...}

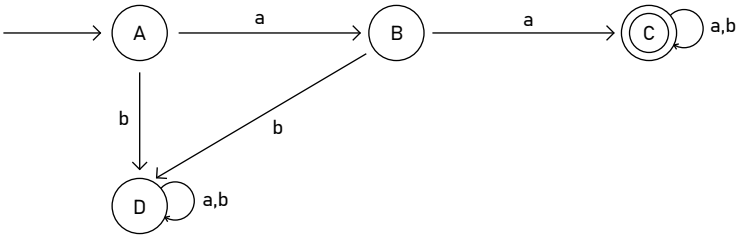


Fig: DFA Diagram

Problem 3

Construct a DFA which accepts the set of all strings over Σ={0,1} where each string starts with "10"

Solution

Alphabet: Σ={0,1}

Language: L(M)={10,100,101,1001,1011,10001,...}

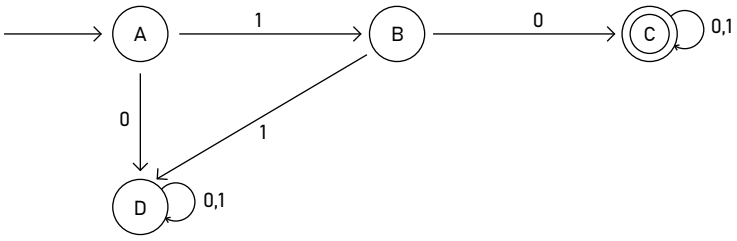


Fig: DFA Diagram

Minimization of DFA

Example 01

Given the language:
L = {w | w has an odd number of a's and ends with 'a' or 'b'}

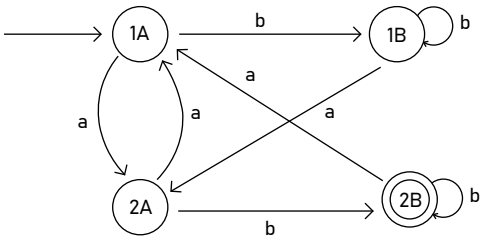


Fig: DFA Diagram

Steps for Minimizing a DFA

1. Remove Unreachable States:
Identify and eliminate any states that cannot be reached from the initial state.

2. Construct the State Transition Table:
Create a table representing state transitions for each input symbol.

3. Mark Initial and Final States:
Clearly indicate which states are initial and final.

4. Find Equivalent States:
Identify and merge states that belong to the same equivalence class.

Now, proceed with minimizing the DFA of Example 01 using these steps.

Step 1: Remove unreachable states

Remove those states which are unreachable from the initial state. No states are removed here because states 1B, 2A, and 2B are reachable from the initial state.

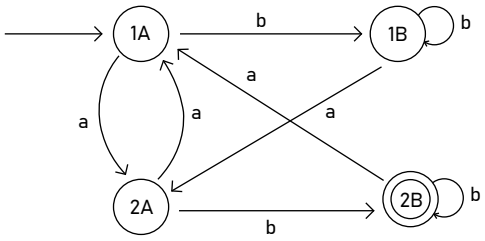


Fig: DFA Diagram

2. Construct the State Transition Table:

State ↓ \ Input →	a	b
1A	2A	1B
1B	2A	1B
2A	1A	2B
2B	1A	2B

Fig: DFA Transition Table

3. Mark Initial and Final States:
Clearly indicate which states are initial and final.

State ↓ \ Input →	a	b
→ 1A	2A	1B
1B	2A	1B
2A	1A	2B
* 2B	1A	2B

4. Find Equivalent States:
Identify and merge states that belong to the same equivalence class.

0-equivalent = [Set of non-final states] [Set of final state]

= [1A 1B 2A] [2B]

1-equivalent = [1A 1B] [2A] [2B]

2-equivalent = [1A 1B] [2A] [2B]

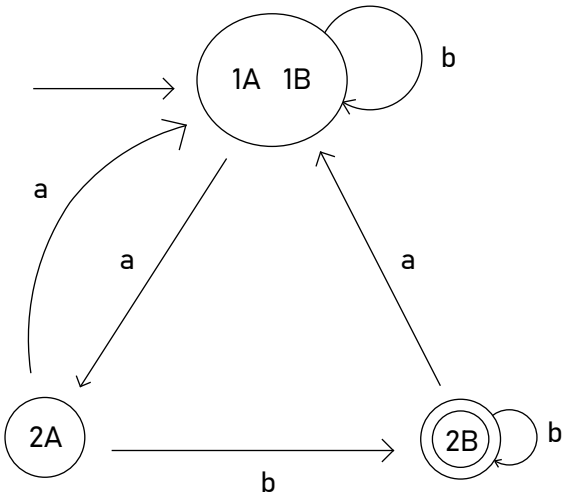


Fig: Minimized DFA Diagram

NFA Figure Construction

Rules for Each Input

1. No Transition:

1.1 Imagine you're walking on a path, but suddenly, the road ends. You have no way to move forward.

1.2 In an NFA, if there is no transition for a given input, the machine cannot continue from that state.

2. Single Transition:

2.1 Imagine you're at a door, and there is only one way to go forward.

2.2 In an NFA (or DFA), if a state has exactly one transition for a symbol, it moves to only one specific next state.

3. Multiple Transitions:

3.1 Imagine you're at a junction where you can go in two or more different directions at the same time.

3.2 In an NFA, a state can have multiple paths for the same input symbol, meaning the machine can follow multiple options at once.

Example 01

L(M) = { w | w starts with 'a' }

Σ = {a, b}

W = { a, ab, aba, abb, ... }

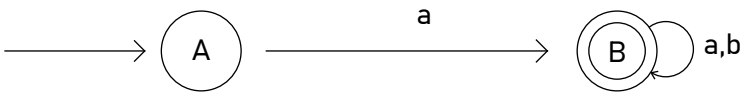


Fig: NFA Diagram

Example 02

L(M) = { w | w is a string that ends with 'a' }

Σ = {a, b}

W = { a, ba, aa, aba, bba, abba, ... }

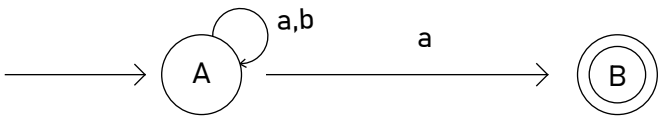


Fig: NFA Diagram

NFA to DFA Construction

Language

$$L(M) = \{ w \mid w \text{ ends with } ab \} \text{ where } \Sigma = \{a, b\}$$

Solution

1. Construct the NFA

Create an NFA for the given language. The transition diagram consists of states A, B, and C, with the following transitions:

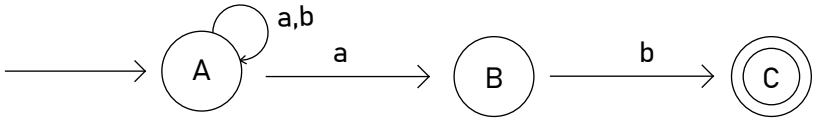


Fig: NFA Diagram

2. State Transition Table for NFA

State ↓ / Input →	a	b
A	{A, B}	{A}
B	{ }	{C}
C	{ }	{ }

Fig: NFA Transition Table

3. Construct the DFA Transition Table

State ↓ / Input →	a	b
A	[A B]	[A]
B	[A B]	[A C]
C	[A B]	[A]

Fig: DFA Transition Table

5. Construct the DFA Figure

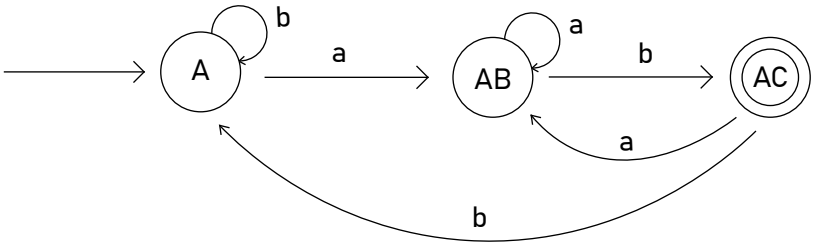


Fig: DFA Diagram

Regular Expression

A regular expression is a method or representation technique used to match patterns on strings. It is widely used in compiler design, is easy to understand, and has efficient implementation.

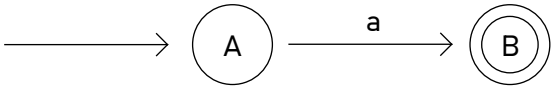
Formal Definition (6 Cases):

Case 1

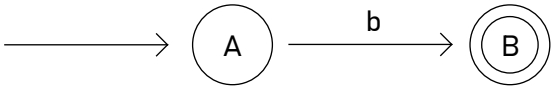
R = a or R = b where a, b ∈ Σ

A language that accepts only one single character: either 'a' or 'b'.

If R = a, then the language is: L = { "a" }



If R = b, then the language is: L = { "b" }

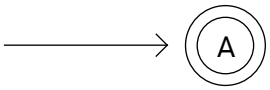


Case 2

R = ε

The empty string ε is a string with length 0 — it contains no characters.

The language is: L = { ε }
That means the NFA should accept the empty string, and only the empty string.



Case 3

R = ø

ø represents the empty set — a language that contains no strings at all, not even ε. So the NFA should reject everything.

Case 4

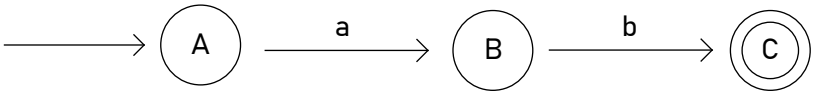
Union ($R = R_1 \cup R_2$ or $R_1 \mid R_2$ or $R_1 + R_2$)

Let's take:

$R_1 = ab$

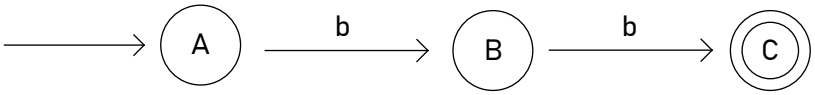
$R_2 = bb$

Build NFA for $R_1 = ab$



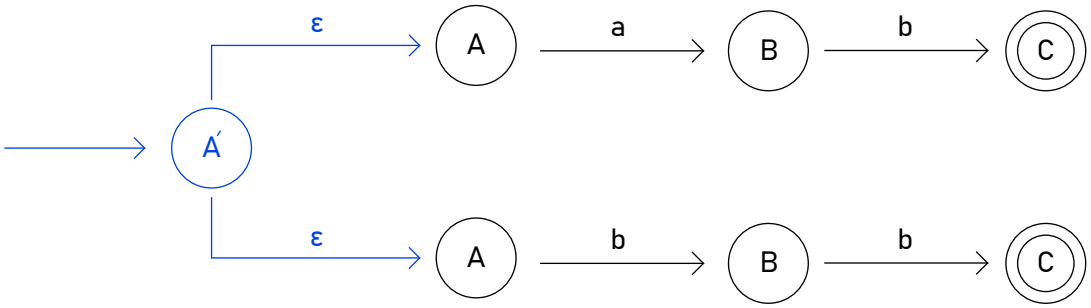
N1

Build NFA for $R_2 = bb$



N2

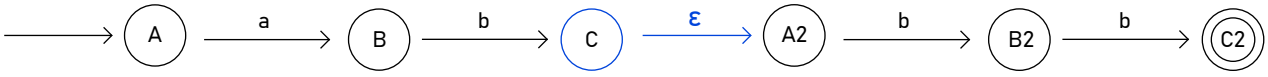
Now we have to union N1 and N2



When we perform a union, we create an extra state and connect it to the initial states of both NFAs using ε.

Case 5

$R = R_1 \cdot R_2$



Why ε-transition?

Because after finishing R_1, the machine jumps to R_2 without reading any input — and continues reading.

When we concatenate two NFA models, the second NFA remains unchanged. We make the final state of the first NFA non-final and connect them using an ε.

Case 6: Kleene Star

$R = R_1^*$ This means: 0 or more repetitions of R_1

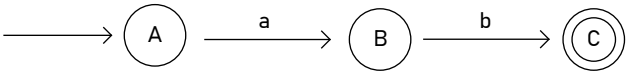
Let's say

R_1 = ab, so R = (ab)*

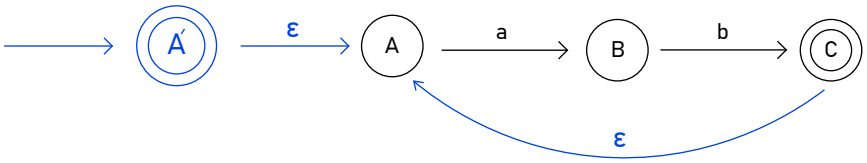
That means:

Accept: "", "ab", "abab", "ababab", etc.

Build NFA for R_1 = ab



Build NFA for R_1 = (ab)*



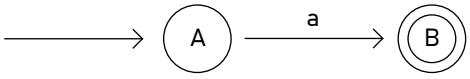
When we apply the Kleene Star, we create an extra final state, but the previous final state remains unchanged. We connect the new final state to the NFA using ε-transitions. Additionally, we draw an ε-transition from the old final state back to the initial state.

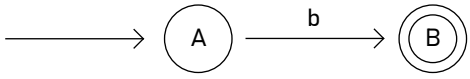
Regular Expression To NFA

Example 01: (ab U a)*

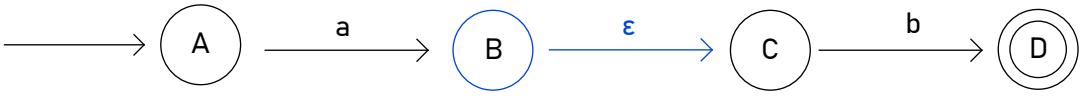
a* = 0 or more
= {ε, a, aa, aaa, ...}
a+ = 1 or more
= {a, aa, aaa, ...}

Plot NFA for each symbol. Here, symbol is a and b.

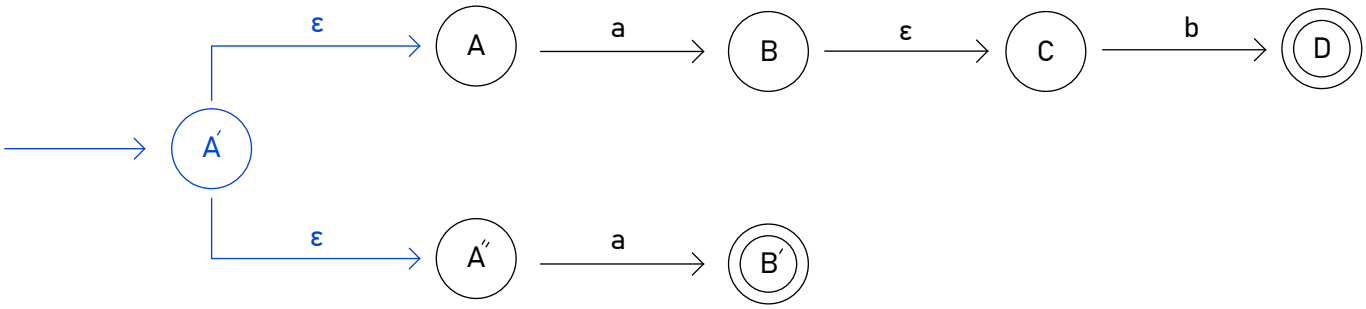




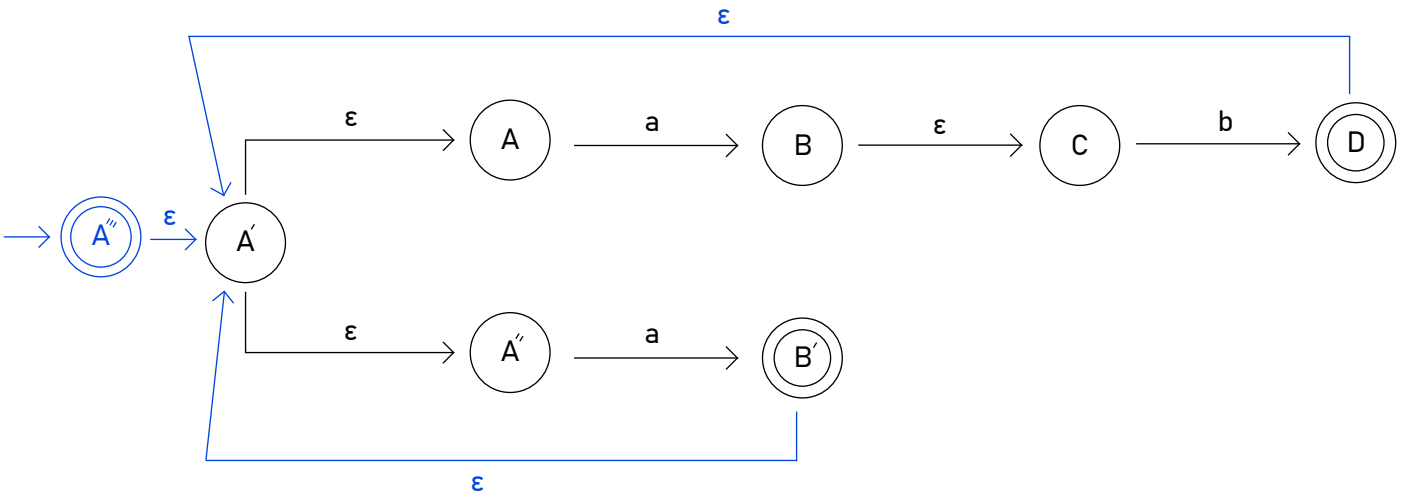
here a and b are concatenated, so **Case 5** is used here



(ab U a)



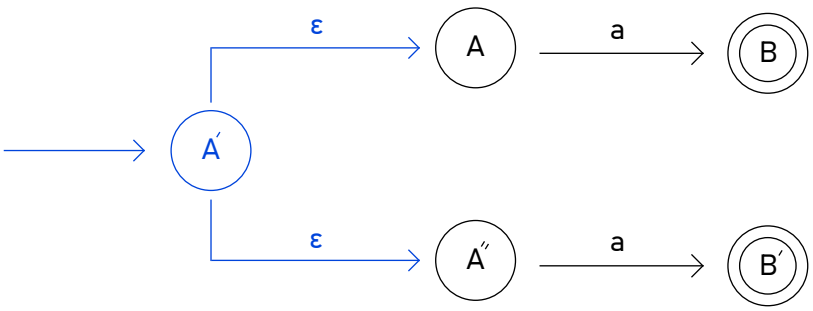
(ab U a)*



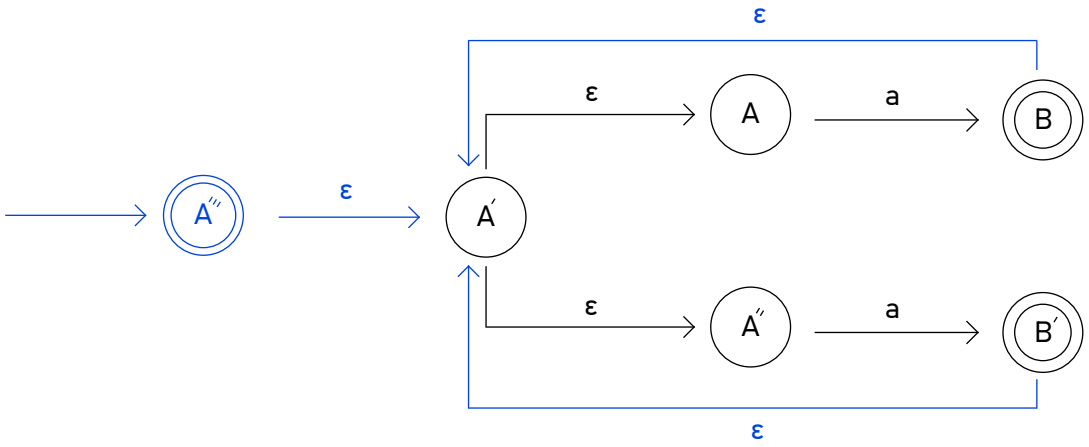
Example 02: (a U b)* aba

plot NFA for each symbol. Here, symbol is a and b.

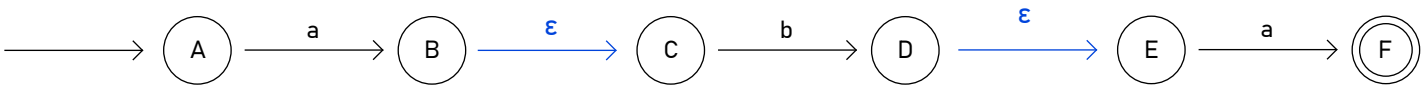
(a U b)



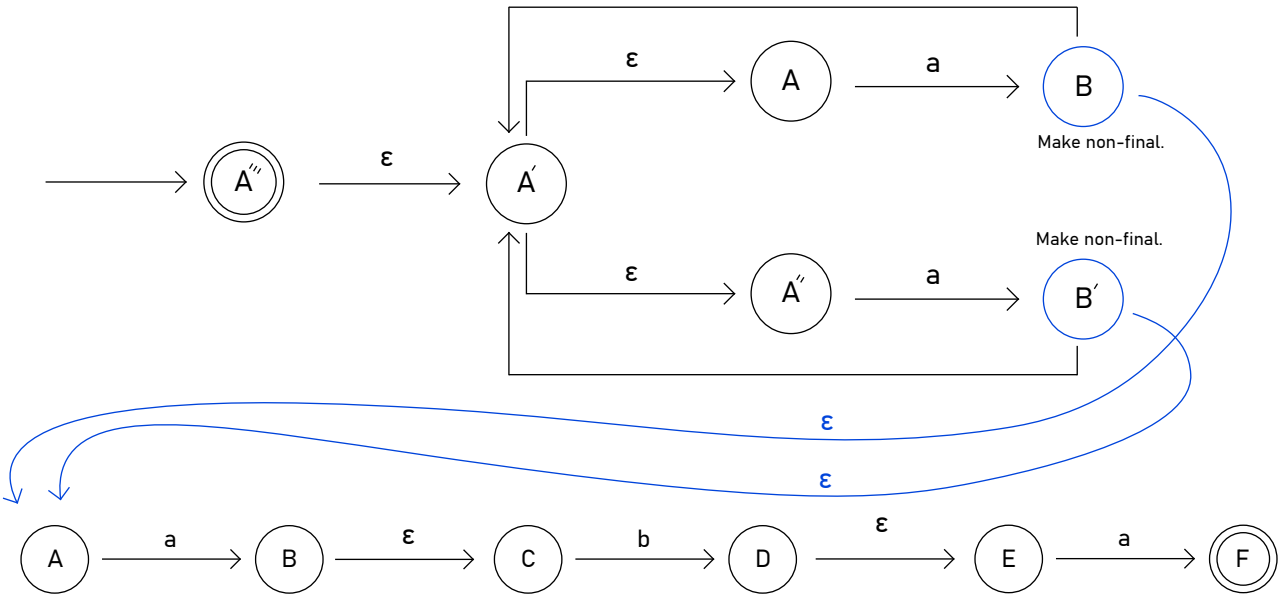
(a U b)*



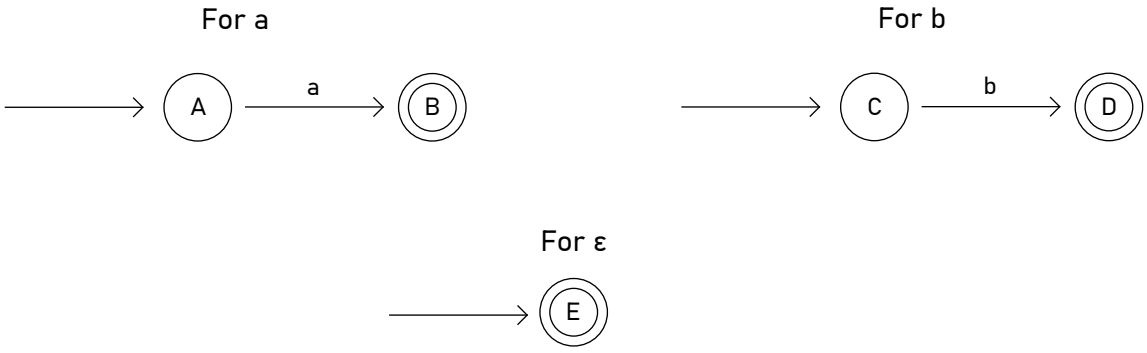
aba



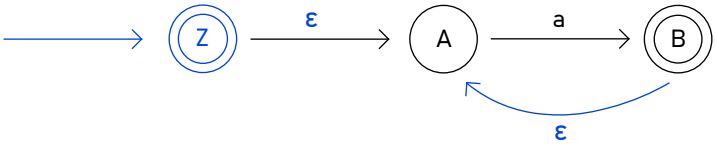
(a U b)* aba



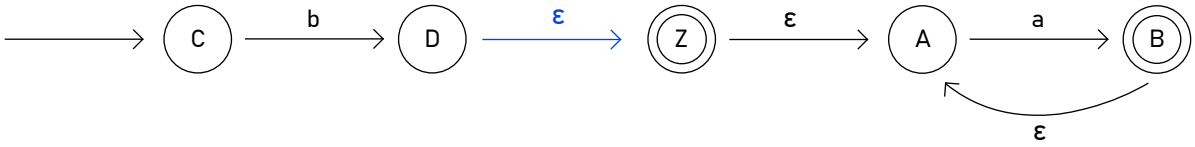
Example 03: ba* + ε



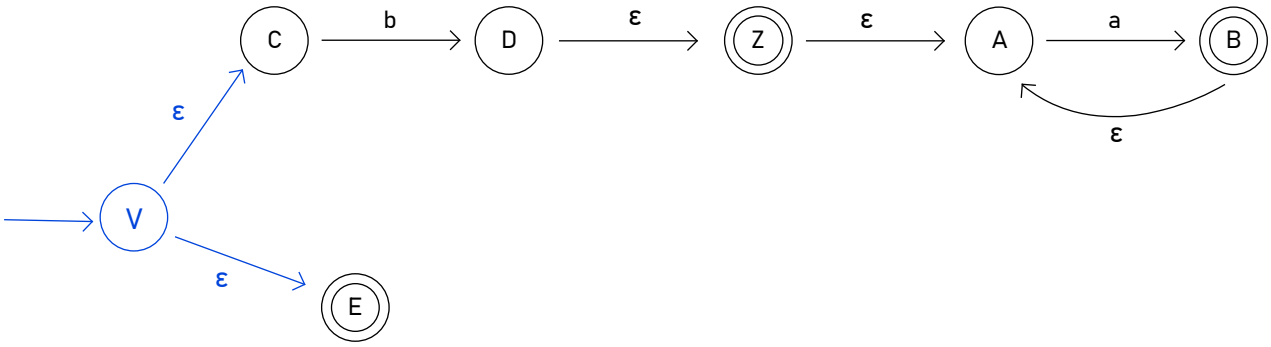
a*



ba*



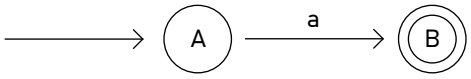
ba* + ε



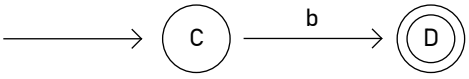
Example 04: a+ U (ab)+

aa* U ab(ab)*

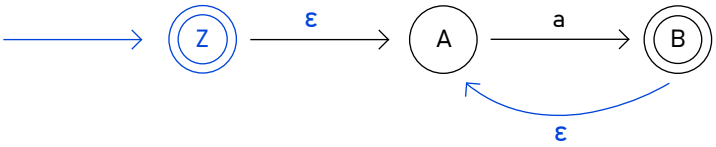
For a



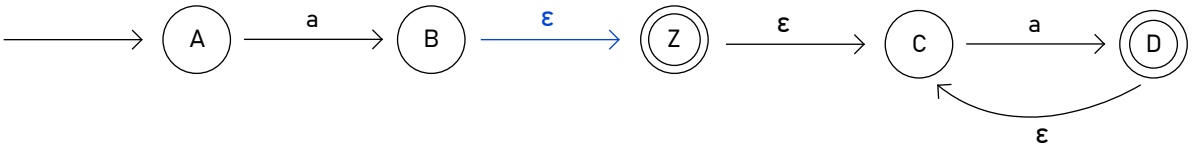
For b



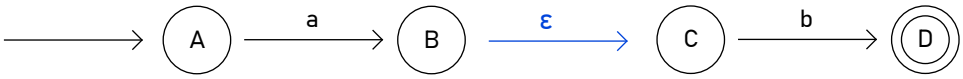
a*



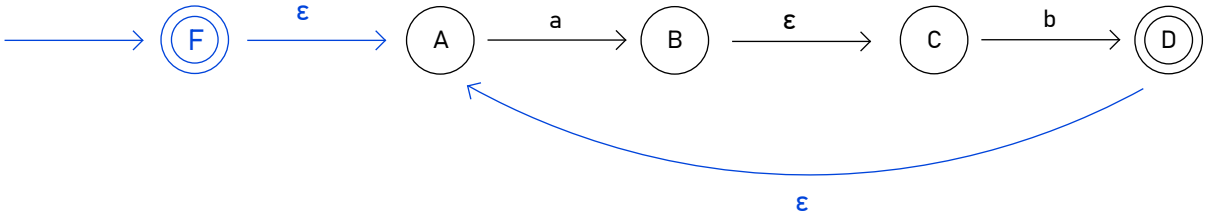
aa*



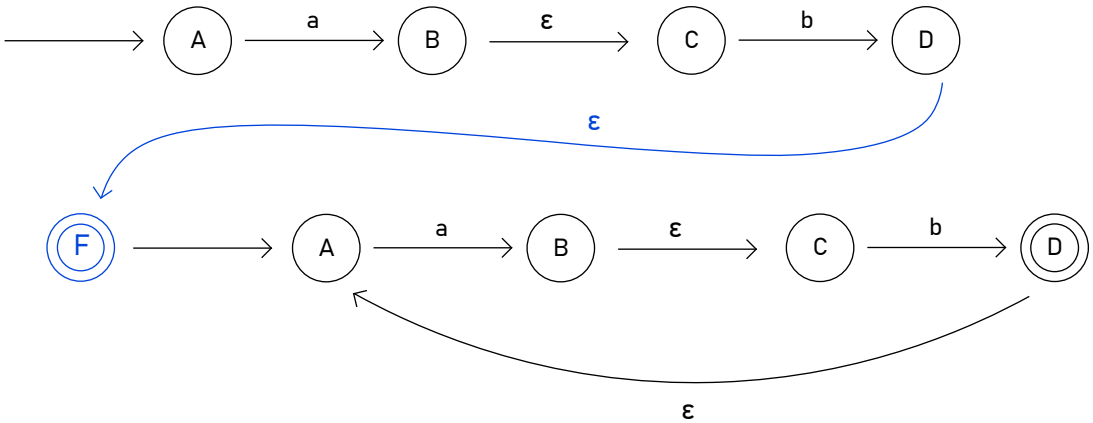
ab



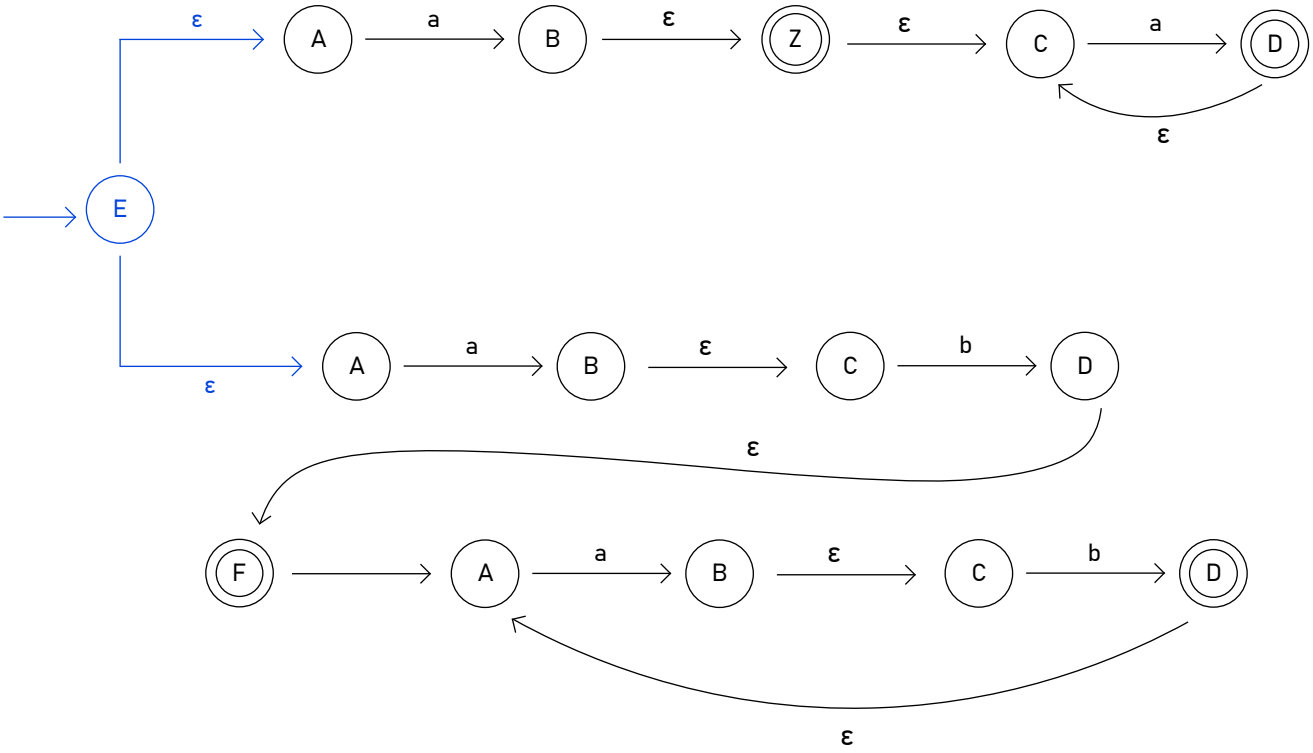
(ab)*



ab(ab)*



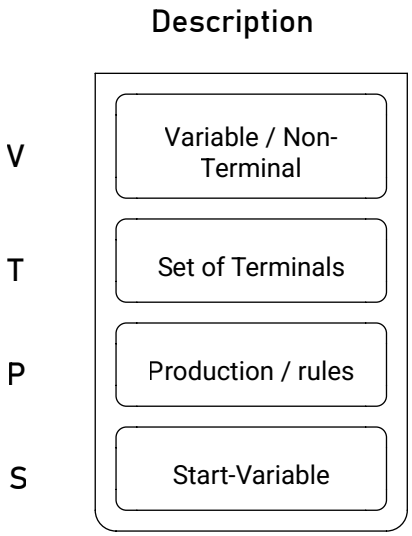
aa* U **ab(ab)***



Context-Free Grammar (CFG)

Context Free Grammar is a set of rules/production which is used to generate patterns of string.

CFG has 4-tuples. These are V, T, P, S



Now, from a grammar, we introduce with four-tuples.

Suppose the grammar is, S → 01 | 0S1

This grammar is for the language, L(M) = { w | w is a set of string of 0^n 1^n where n ≥ 1 }

0^1 1^1 = 01

0^2 1^2 = 0011

The grammar is defined with the production rule: S → 01 | 0S1
The 4-tuples are determined as

V = {S} (set of variables/non-terminals)

T = {0, 1} (set of terminals)

S = S (start variable)

P = {S → 01 | 0S1} (set of production rules)

Examples:

Identify V, T, P, S from the grammar G:-

1) E => E + T | T
T => T * F | F
F => (E) | id

V = {E, T, F}
T = {+, *, (,), id}
P = {E => E + T, E => T, T => T * F, T => F, F => (E), F => id}
S = E

Identify V, T, P, S from the grammar G:-

2) S => (L) | a
L => L, S | S

V = {S, L}
T = { (,), a, ,}
P = {S => (L), S => a, L => L, S, L => S}
S = S

USING Context-free grammar, we can match pattern of a string in two ways:-

1) Mathematical derivation

2) Parsing Tree derivation

Now, using CFG, we match the pattern of a string in two ways for the following languages.

Example 01:

L = {a^n where n > 0}
= {ε, a, aa, aaa, ..., ...}

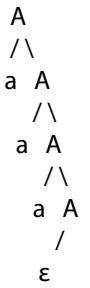
Grammar for the language is,
A => aA | ε

NOW, using the grammar, we match the pattern of string in two ways,

Mathematical Derivation: Suppose, here we match the pattern of a string which is 'aaa'

A => aA
A => aaA
A => aaaA
A => aaaε
A => aaa

Parsing Tree Derivation: Using this derivation technique, now we match the pattern of a string which is 'aaa'



From the above, two derivation techniques, we see that the string pattern 'aaa' is matched by two techniques / CFG techniques. So it can be said that, the grammar A => aA | ε is correct for the language, L(M) = {a^n where n > 0}

Example 02:

Match the pattern of the given string 'abbaababba' using context free grammar:-

S => aSa | bSb
S => ε

Show both mathematical and parsing tree derivation

Mathematical: [abbaabba]

S => aSa
S => a bSb a
S => a b bSb ba
S => abb aSa bba
S => abba ε abba
S => abbaabba

Parsing tree:

Language Processing System

How to compiler convert a pure high level language code to machine code ?
How to compiler translate a pure high level language code to binary file ?

1. Pre-Processor

a. Handles file inclusion (e.g., #include <iostream>) and macro processing (e.g., #define Limit 40).

b.Converts pure high-level language (HLL) code to assembly code.

2. Compiler

a. Translates high-level language code into assembly code.

Example: int add(int a, int b) { return a + b; }, The compiler converts this C++ function into assembly instructions like mov, add, etc.

3. Assembler

a. Converts assembly code into object code or object files (e.g., test.obj).

Example: Assembly code: mov eax, ebx The assembler translates this into machine-readable object code (e.g., 89 D8 in hex).

4. Linker

a. Links object files into a single binary during compilation.

Example: Links test.cpp to test.obj.

Types: Static (compile-time) and
Dynamic (runtime, e.g., Java)

5. Loader

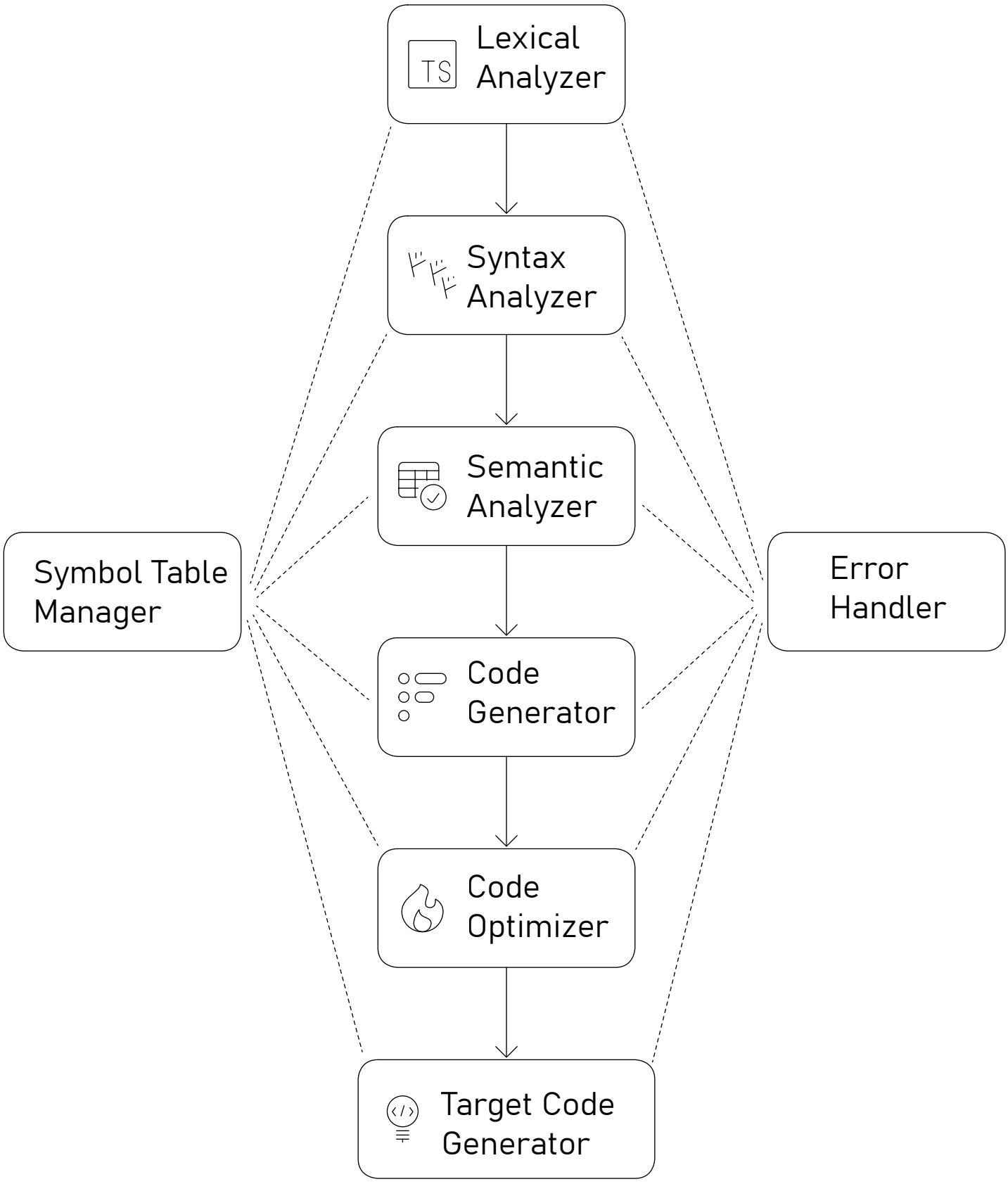
a. Allocates memory and loads the program into memory for execution.

Example: Running ./test makes the loader set up memory, load the test binary, and start it at main.

Differences between a Linker and a Loader:

Feature	Linker	Loader
Part of	Binary file system	Operating system
Function	Combines object files into a binary	Loads the program into memory
Timing	Occurs during compilation	Occurs during program execution
Example	Links `test.cpp` to `test.obj	Loads `./test` to start at `main

Structure of a Compiler



Lexical Analyzer (Scanner)

1. Processes the pure high-level language input.
2. Converts the input into a stream of tokens.

Syntax Analyzer

1. Takes the stream of tokens as input.
2. Produces a syntax tree (also known as a parsing tree).

Semantic Analyzer

1. Verifies the syntax tree for semantic correctness.
2. Outputs a verified syntax tree or parsing tree.
3. Generates intermediate code.

Code Generator

1. Converts the intermediate code into three-address code.
2. Follows three-address code rules for this conversion.

Code Optimizer

1. Optimizes the three-address code.
2. Produces optimized code for better performance.

Target Code Generator

1. Converts the optimized code into assembly code.
2. Outputs the final assembly code

Lexical Analyzer (Scanner)

Lexeme

A lexeme is a sequence of characters in the source code that forms a single unit of meaning during the lexical analysis phase of compilation. It's essentially the raw input that the lexical analyzer (scanner) identifies and categorizes into a token. For example, in the code int x = 5;, the lexemes would be int, x, =, 5, and ;. Each lexeme is then classified into a token type (like keyword, identifier, operator, or literal) for further processing by the compiler.

A pattern is a rule defining the structure of a lexeme for a token type.

Example: For identifiers, the pattern is letter (letter | digit)*.

Starts with a letter [a-zA-Z].

Followed by zero or more letters [a-zA-Z] or digits [0-9].

Valid examples: x, var, myVar123.

Invalid examples: 1var (starts with a digit), _var (starts with an underscore, if not allowed).

Function of the Lexical Analyzer

The Lexical Analyzer, or scanner, breaks down the source code into a sequence of tokens by:

1. Reading the input character by character.
2. Grouping characters into lexemes (meaningful units like keywords, identifiers, operators).
3. Matching lexemes against predefined patterns.
4. Assigning each lexeme to a token type (e.g., "KEYWORD", "IDENTIFIER", "OPERATOR").
5. Removing whitespace, comments, and newline characters.
6. Passing the stream of tokens to the Syntax Analyzer.

How the Lexical Analyzer Performs Its Job

flaot postion, initial, rate;
postion = initial + rate * 60

ID/ Number / Serial Number	Data Type	Variable / Function
01	Float	position
02	Float	pnitial
03	Float	rate

Token

Syntax : <token_name, token_number>

<id, 1> = <id, 2> + <id, 3> * 60

id1= id2 + id3 * 60

Syntax Analyzer

A Syntax Analyzer, also called a Parser, is a part of a compiler that checks if the code is written in a valid way according to the programming language rules.

How It Works (Step by Step)

Input:

1. It takes tokens from the Lexical Analyzer (tokens = small pieces like keywords, variables, symbols).

Grammar Rules:

1. The parser uses grammar rules (like a set of instructions) to know what a correct program should look like.
2. Most of the time, these rules are written using something called Context-Free Grammar (CFG).

Parsing (Building a Tree):

1. The parser builds a tree-like structure (called a syntax tree or parse tree) that shows how the code fits the grammar.
2. There are two main ways to do this:

    2.1 .Top-Down Parsing: Starts from the top (the main rule) and breaks it down step by step.

    2.2 . Bottom-Up Parsing: Starts from the tokens and builds up to the main rule.

Error Checking:

1. If something is written wrong (like a missing semicolon or bracket), the parser will detect it and show an error message.

Output:

1. If the code is correct, it gives a syntax tree.
2. This tree is then passed to the Semantic Analyzer (next step in the compiler) for deeper understanding.

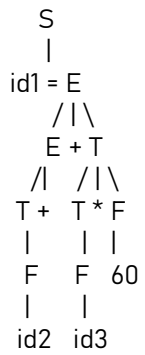
Example :

S => id = E
E => E + T | T
T => T * F | F
F => id | Number

id1= id2 + id3 * 60

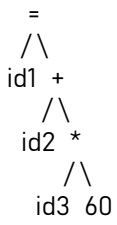
S => id1 = E
id1 = E + T
id1 = T + T
id1 = F + T
id1 = id2 + T
id1 = id2 + T * F
id1 = id2 + F * F
id1 = id2 + id3 * F
id1 = id2 + id3 * 60

Construct the Parsing Tree



Construct the Syntax Tree

id1 = id2 + id3 * 60

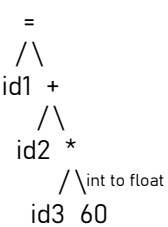


Semantic Analyzer

The Semantic Analyzer checks the meaning of code beyond syntax, verifying types, scope, and context using the syntax tree, and outputs a verified tree or errors.

Example :

id1 = id2 + id3 * 60



Code Generator

The Code Generator translates the intermediate code (e.g., three-address code) into target machine code or assembly language.

Three-Address Code (TAC) Rules

Three-address code is an intermediate code used in compilers. It breaks down complex statements into simple instructions with at most three addresses (or operands). These rules help to generate TAC during the Code Generation phase.

1. Instruction Format

General form: x = y op z
x: Where the result is stored
y, z: Operands (variables or constants)
op: Operator like +, -, *, /, <, ==, etc.

Example:
t1 = a + b → Adds a and b, stores result in t1

2. One Operation Per Instruction

Each instruction should have only one operation.
Complex expressions are split into multiple steps.

Example:
For x = a + b * c

```
t1 = b * c
t2 = a + t1
x = t2
```

3. Control Flow Support

TAC supports if, goto, and labels for decision-making and loops.

```
if a < b goto L1
t1 = a + b
goto L2
L1: t2 = a - b
L2: ...
```

4. Assignments

Simple assignments like x = y or x = 5 are allowed.
No need for an operator here.

Example:

```
x = y
z = 100
```

5. Temporary Variables

Temporary variables (t1, t2, etc.) hold intermediate results.
Helps in breaking down complex calculations step by step.

Example:
y = (a - b) * (c + d)

```
t1 = a - b
t2 = c + d
y = t1 * t2
```

6. Address Calculations

Used in arrays and pointers.
Compute memory locations manually in TAC.

Example (array access):
For x = a[i], assuming 4-byte elements:

```
t1 = i * 4
t2 = &a + t1
x = *t2
```

7. Three Addresses Only

Each instruction can use at most 3 addresses: one for the result and two for operands.

Keeps TAC simpler than one-address or two-address code used in machine instructions.

Example

id1 = id2 + id3 * 60.00

```
t1 = 60.00
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Code Optimizer

The Code Optimizer enhances the efficiency of the intermediate code by reducing resource usage and improving performance.

Optimized Three-Address Code

```
t2 = id3 * 60.00
id1 = id2 + t2
```

Target Code Generator

The Target Code Generator is the final stage in the compiler pipeline, responsible for translating the optimized intermediate code (e.g., optimized three-address code) into target assembly code or machine code specific to the target architecture (e.g., x86, ARM).

First and Follow

First

The FIRST(X) set of a grammar symbol X (terminal or non-terminal) is the set of terminals that begin the strings derivable from X.

Follow

The FOLLOW(X), set of terminals that can appear immediately to the right of X in some production.

Examples For First

1. A ⟶ aA	FIRST (A) = {a}
2. A ⟶ a	FIRST (A) = {a}
3. A ⟶a | ε	FIRST (A) = {a, ε}
4. A ⟶a B| ε	FIRST (A) = {a, ε}
5. A ⟶(a B)| ε	FIRST (A) = {(, ε}
6. A ⟶Ta A ⟶*FT´	FIRST (A) = FIRST (T) = {*}
7. A ⟶Ta A ⟶*FT´| ε	FIRST (A) = FIRST (T) = {*, ε}

Example For First and Follow

	First	Follow
E ⟶ T E'	{id, (}	{$,)}
E' ⟶ + T E' | ε	{+, ε}	{$,)}
T ⟶ F T'	{id, (}	{+, $,)}
T' ⟶ * F T' | ε	{*, ε}	{+, $,)}
F ⟶ id | (E)	{id, (}	{*, +, $,)}

	First	Follow
S ⟶ a A B c | a	{a}	{$}
A ⟶ c | ε	{c , ε}	{d, b}
B ⟶ d | ε	{d, ε}	{b}

To write it, it took three months; to conceive it, three minutes; to collect the data in it, all my life.

-F. Scott Fitzgerald