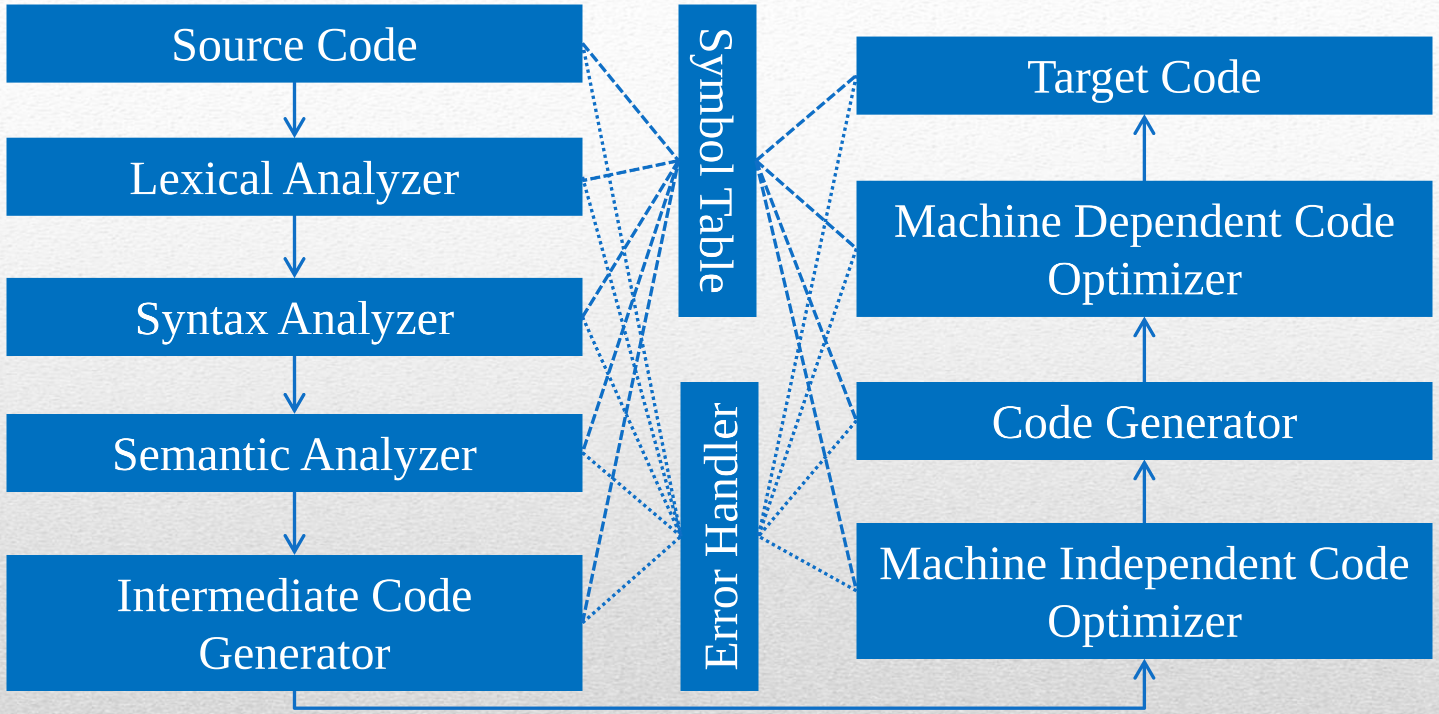
# COMPILER DESIGN

Adapted By Manik Hosen



# Phases of Compiler

- The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler:
- Lexical Analysis: This phase scans the source code as a stream of characters and converts it into meaningful lexemes.
- Syntax Analysis: It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar.
- Semantic Analysis: Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer.

## Phases of Compiler

- Intermediate Code Generation: After semantic analysis the compiler generates an intermediate code of the source code for the target machine.
- Code Optimization: The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources.
- Code Generation: In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language.

## Phases of Compiler

- This phase scans the source code as a stream of characters and converts it into meaningful lexemes. For each lexeme, the lexical analyzer produces as output a token of the form

  (token-name, attribute-value)

- That it passes on to the subsequent phase, syntax analysis.

## Lexical Analysis

- A single-pass compiler is a type of compiler that passes through the source code of each compilation unit only once.
- A multi-pass compiler is a type of compiler that processes the source code or abstract syntax tree of a program several times.

## Compiler

## Regular Expression

- Each regular expression **r** denotes a language L(r), which is also defined recursively from the languages denoted by r's sub-expressions. Here are the rules that define the regular expressions over some alphabet Σ and the languages that those expressions denote.
- There are two rules that form the basis of regular expression:
  1. Ɛ is a regular expression, and L (Ɛ) is {Ɛ} , that is, the language whose sole member is the empty string.
  2. If **a** is a symbol in Σ, then **a** is a regular expression, and L(a) = {a}, that is, the language with one string, of length one, with a in its one position.

*i.* $S \to aS, S \to aB, B \to bC, C \to aC, C \to a$

$S \to aS|aB$

$B \to bC$

$C \to aC|a$

*ii.* $S \to aA, S \to a, A \to aA, A \to bB, A \to a, B \to bB, B \to aA$

$S \to aA|a$

$A \to aA|bB|a$

$B \to bB|aA$

## Regular Expression

- Regular Expression is important because:
  - Regular expression is used in most of languages.
  - Regular expressions can help you write short code.
  - Regular expressions save time.
  - Regular expressions are fast.
  - Regular expressions can match just about anything.

## Regular Expression

- Context-Free Grammars are important because:
- It distinguish the properties of the language from the properties of a particular grammar.
- Most programming languages are context-free.

## Context Free Grammar

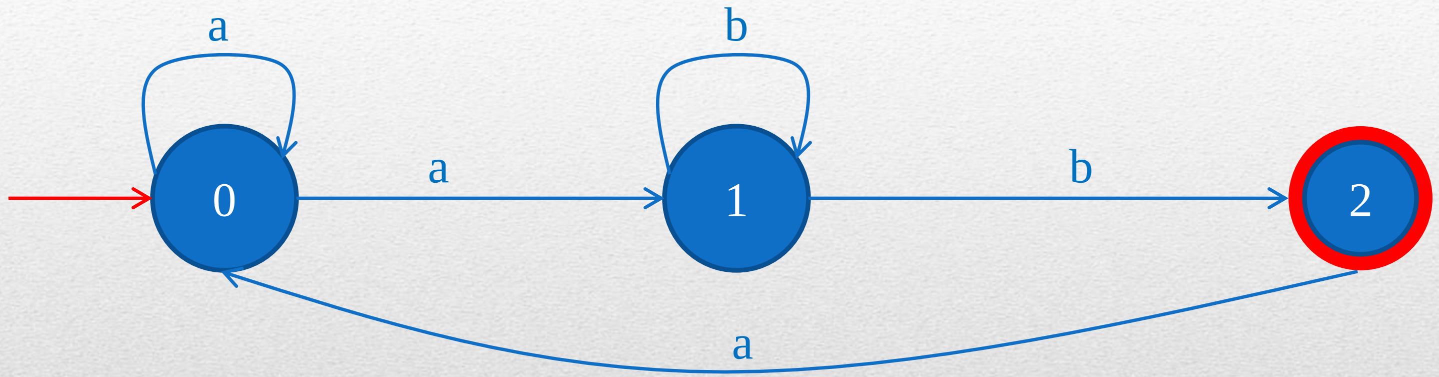
- A Deterministic Finite Automaton (DFA) is a finite-state machine that accepts or rejects strings of symbols and only produces a unique computation of the automaton for each input string. Deterministic refers to the uniqueness of the computation.

## D Finite Automata

- Finite Automata are used two of the three front-end phases of the compiler.
- The first phase, Lexical Analysis, uses Regular Expressions to tokenize the input. Regular expressions are usually implemented with Finite Automata.
- The more important part is the second phase, Parsing. It also use finite automata.

## Finite Automata

- DFA accepting w∈{a, b}*w starts and ends with different symbol:



# D Finite Automata

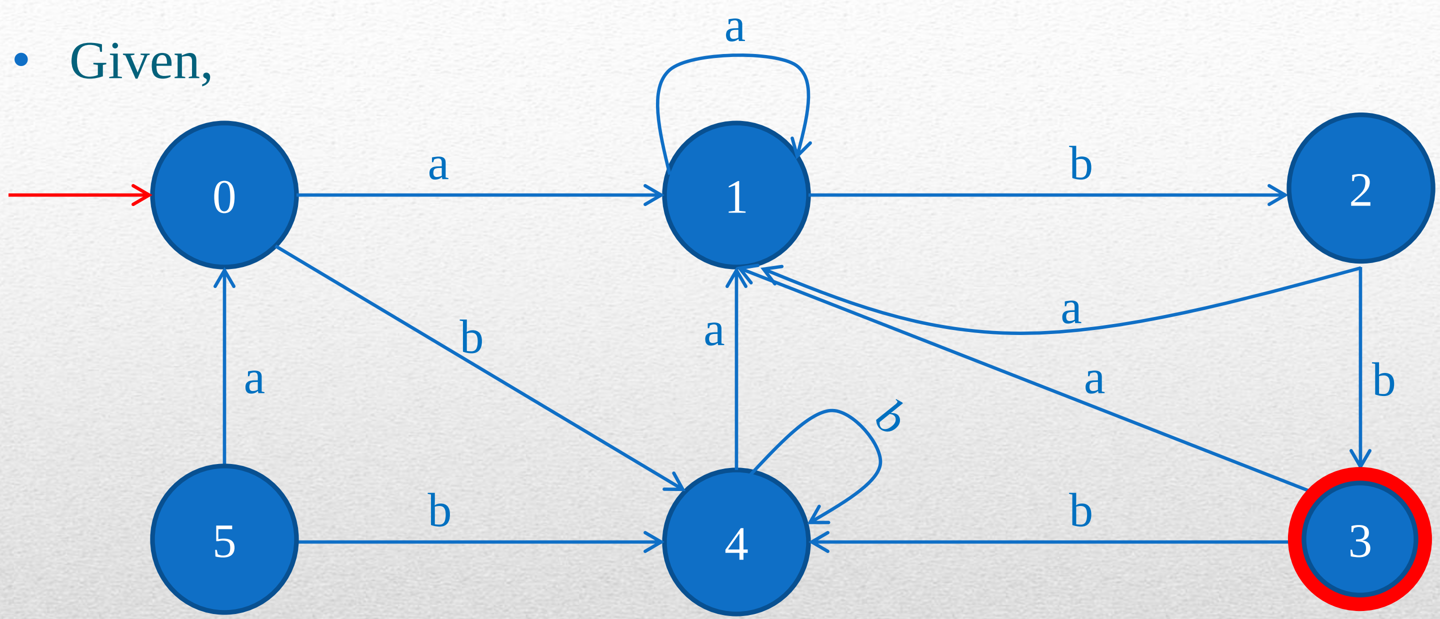
- DFA accepting w∈{a, b}*w has odd number of a and odd number of b:

$$\begin{tikzcd} {} \arrow[r] & 0 \arrow[r, bend left, "a"] & 1 \arrow[l, bend left, "a"] \arrow[r, bend left, "b"] & \boxed{2} \arrow[l, bend left, "b"] \end{tikzcd}$$

## D Finite Automata

## D Finite Automata

- Given,

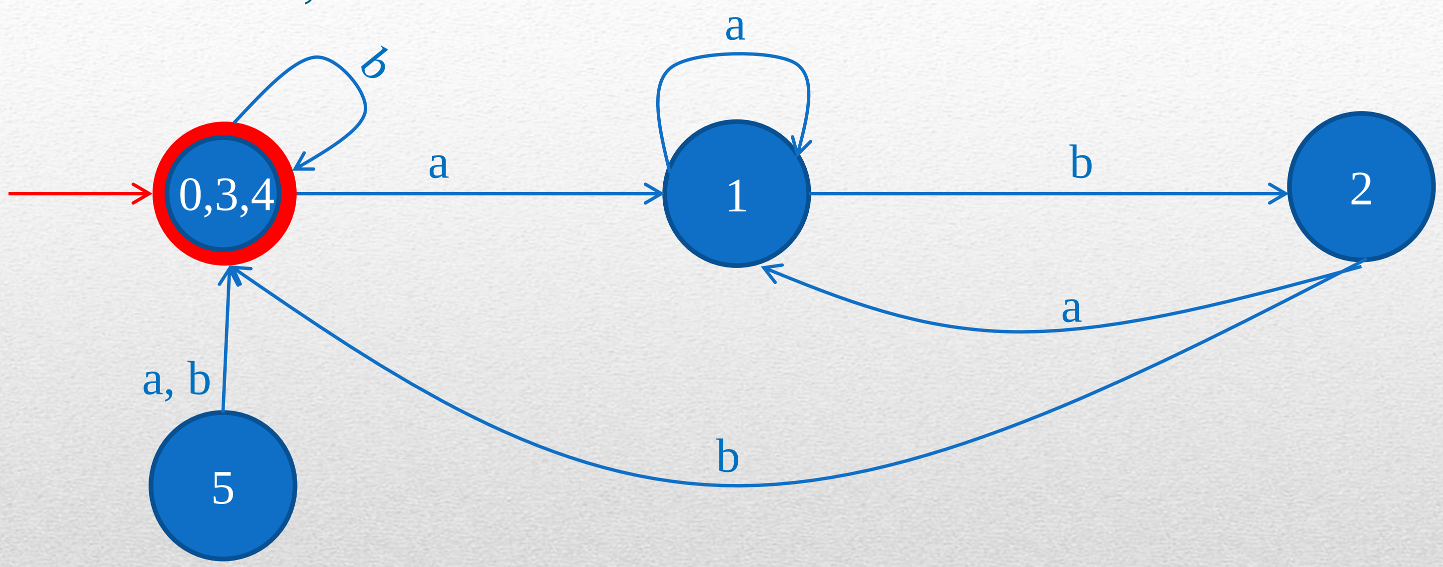


# D Finite Automata

- Table:

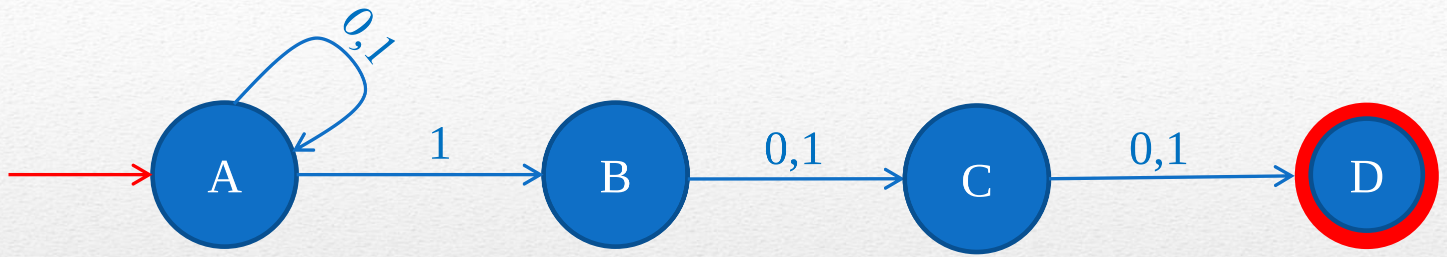
|   | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 🟩 | ⬛ | 🟦 | 🟩 | 🟩 | a |
| 1 | a | a | a | a | a | 🟪 |
| 2 | 🟩 | b | 🟦 | 🟩 | 🟩 | 🟪 |
| 3 | 🟩 | ⬛ | b | 🟩 | 🟩 | 🟪 |
| 4 | b | ⬛ | 🟦 | b | b | b |
| 5 | 🟩 | ⬛ | 🟦 | 🟩 | 🟩 | 🟪 |

- Minimized,



## D Finite Automata

- Given NFA,



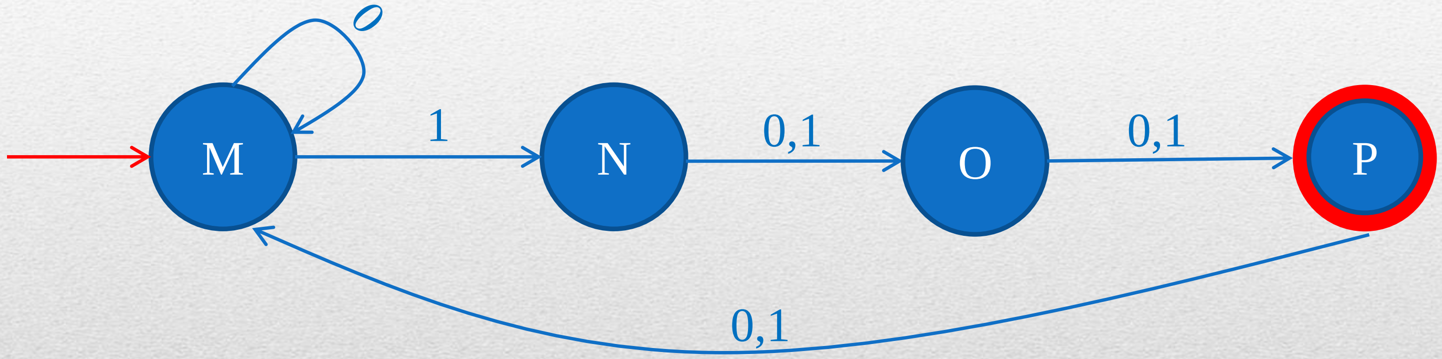
## Finite Automata

- Table:

| NFA State | DFA State | 0 | 1 |
|---|---|---|---|
| {A} | M | M | N |
| {A, B} | N | O | O |
| {A, C} | O | P | P |
| {A, D} | P | M | M |

# Finite Automata

- Converted DFA,

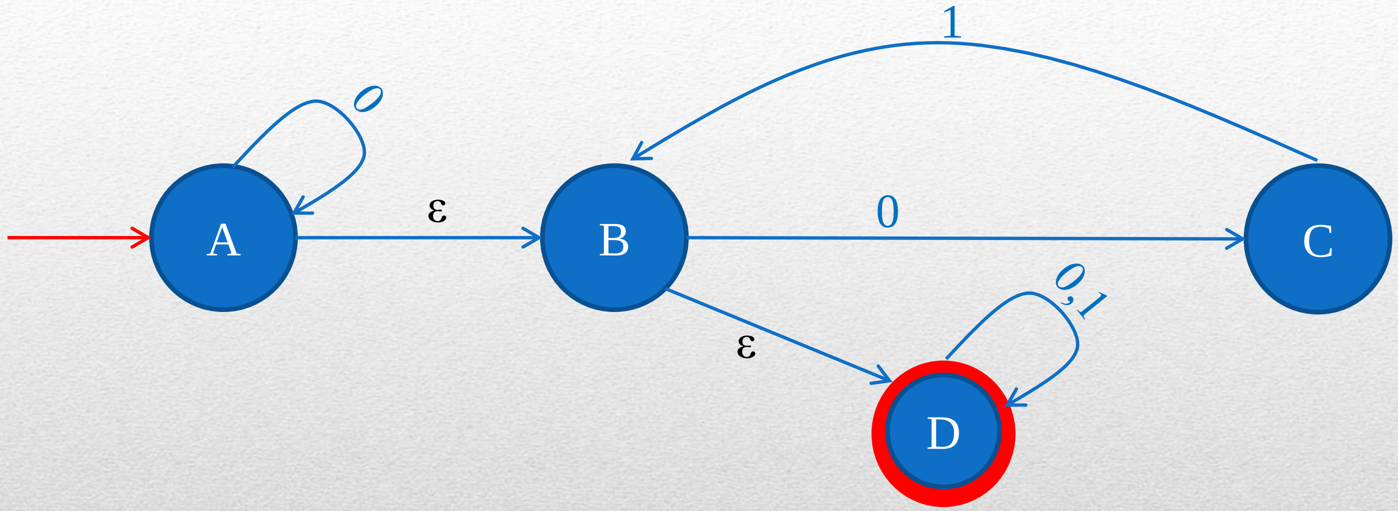


## Finite Automata

- ε-NFA: Nondeterministic finite automaton with ε-moves is a further generalization to NFA. This automaton replaces the transition function with the one that allows the empty string ε as a possible input.

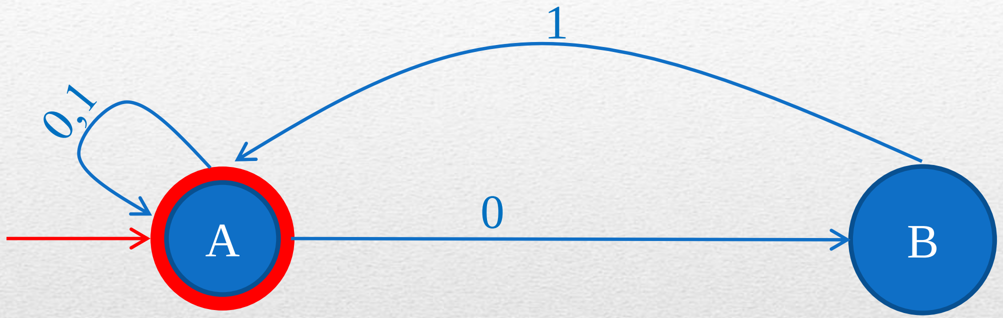
## N Finite Automata

- Given ε-NFA,


$$\begin{tikzcd} \arrow[r] & A \arrow[loop above, "0"] \arrow[r, "\varepsilon"] & B \arrow[rr, "0"] \arrow[dr, "\varepsilon"'] & & C \arrow[ll, bend right, "1"'] \\ & & & D \arrow[loop above, "0{,}1"] & \end{tikzcd}$$

# N Finite Automata

- Converted NFA,



# N Finite Automata

- An ambiguous grammar is a context-free grammar for which there exists a string that can have more than one leftmost derivation or parse tree, while an unambiguous grammar is a context-free grammar for which every valid string has a unique leftmost derivation or parse tree.

## Ambiguity

i. $A \rightarrow A@B|B, B \rightarrow B\#C|C, C \rightarrow C@D|D, D \rightarrow d$

$\Longrightarrow A \rightarrow A@B|B, B \rightarrow B\#C|C, C \rightarrow C@d|d, D \rightarrow d$

$\Longrightarrow A \rightarrow A@B|B, B \rightarrow B\#C|C, C \rightarrow C@d|d$

$\text{Ans}: A \rightarrow A@B|B, B \rightarrow B\#C|C, C \rightarrow C@d|d$

ii. $E \rightarrow E - E|E * E|E^\wedge E| - E$

$\Longrightarrow E \rightarrow E - R|E * E|E^\wedge E| - E, E \rightarrow R$

$\Longrightarrow E \rightarrow E - R|E^\wedge E| - E, E \rightarrow R, R \rightarrow R * S, R \rightarrow S$

$\Longrightarrow E \rightarrow E - R|E^\wedge E| - E, E \rightarrow R, R \rightarrow R * S, R \rightarrow S$

$\Longrightarrow E \rightarrow E - R| - E, E \rightarrow R, R \rightarrow R * S, R \rightarrow S, S \rightarrow S^\wedge T, S \rightarrow T$

$\Longrightarrow E \rightarrow E - R, E \rightarrow R, R \rightarrow R * S, R \rightarrow S, S \rightarrow S^\wedge T, S \rightarrow T, T \rightarrow -T$

$Ans: E \rightarrow E - R|R, R \rightarrow R * S|S, S \rightarrow S^\wedge T|T, T \rightarrow -T$

# Ambiguity

- Left factoring is removing the common left factor that appears in two productions of the same non-terminal.
- It is done to avoid back-tracing by the parser. Suppose the parser has a look-ahead ,consider this example-

$$A \rightarrow qB \mid qC$$

- where A,B,C are non-terminals and q is a sentence. In this case, the parser will be confused as to which of the two productions to choose and it might have to back-trace. After left factoring, the grammar is converted to-

$$A \rightarrow qD$$

$$D \rightarrow B \mid C$$

- In this case, a parser with a look-ahead will always choose the right production.

## Parsing

- $S \rightarrow aSSbS|aSaSb|abb|b$

  $\Rightarrow S \rightarrow aS'|b$

  $\Rightarrow S' \rightarrow SSbS|SaSb|bb$

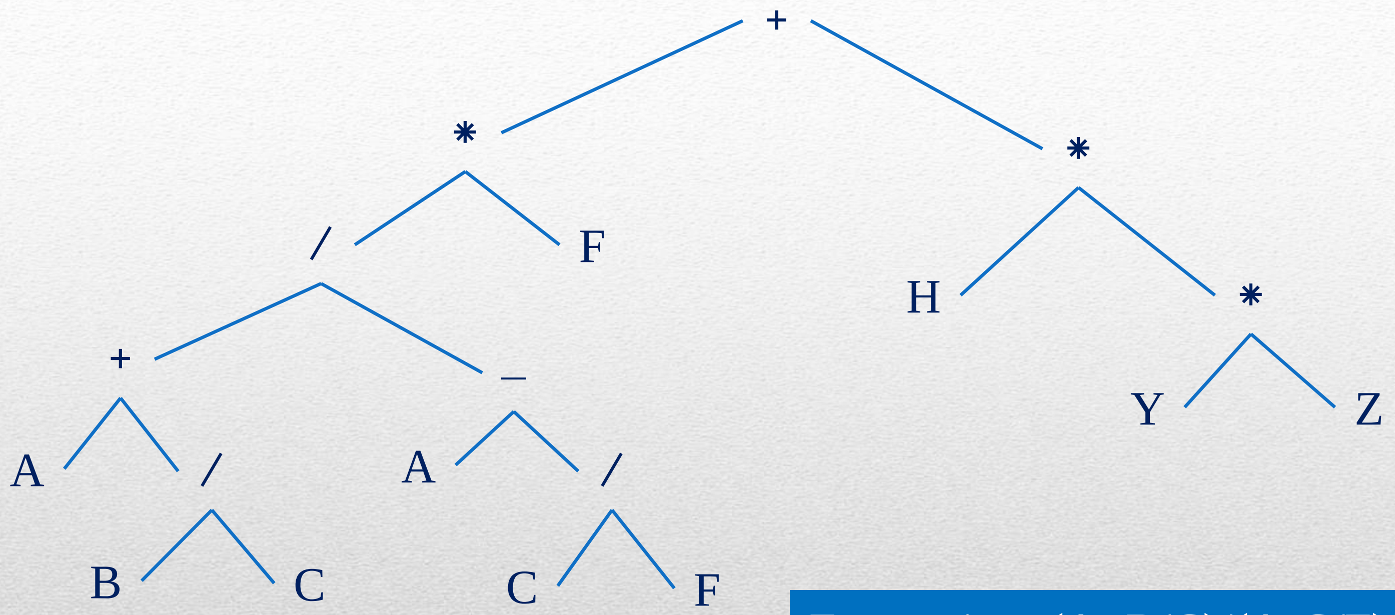
  $\Rightarrow S' \rightarrow SS''|bb$

  $\Rightarrow S'' \rightarrow SbS|aSb$

## Ambiguity

| Parse Tree | Syntax Tree |
|---|---|
| A parse tree is an ordered, rooted tree that represents the syntactic structure of a string according to some context-free grammar. | A syntax tree, is a tree representation of the abstract syntactic structure of source code written in a programming language. |
| It is also known as Parsing tree, derivation tree, and concrete syntax tree. | It is also known as abstract syntax tree. |
| parse tree contains records of the rules (tokens) to match input texts. | syntax tree contains records of the syntax of programming language. |

## Parse Tree vs Syntax Tree



Expression: (A+B/C)/(A-C/F)*F+(H*Y*Z)

# Syntax Tree

|  | Operator | Operand-1 | Operrand-2 | Result |
|---|---|---|---|---|
| 1. | – | B |  | $T_1$ |
| 2. | + | C | D | $T_2$ |
| 3. | * | $T_1$ | $T_2$ | $T_3$ |
| 4. | := | $T_3$ |  | A |
| Quadruple representation of A: = –B*(C+D) |  |  |  |  |

# Quadruple Representation

|  | Operator | Operand-1 | Operrand-2 |
|---|---|---|---|
| 1. | – | B |  |
| 2. | + | C | D |
| 3. | * | 1. | 2. |
| 4. | := | 3. |  |
| Triple representation of  A: = –B*(C+D) |  |  |  |

# Triple Representation

- Symbol Table: Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler.

## Symbol Table

- Loop Optimization: In compiler theory, loop optimization is the process of increasing execution speed and reducing the overheads associated with loops. It plays an important role in improving cache performance and making effective use of parallel processing capabilities.

## Loop Optimization

- Quadruple: It is structure with consist of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

## Quadruple

- Operator Grammar: An operator precedence grammar is a kind of grammar for formal languages. Technically, an operator precedence grammar is a context-free grammar that has the property that no production has either an empty right-hand side or two adjacent non-terminals in its right-hand side.

## Operator Grammar

## LALR Parser

- LALR parser: In computer science, an LALR parser or Look-Ahead LR parser is a simplified version of a canonical LR parser, to parse (separate and analyze) a text according to a set of production rules specified by a formal grammar for a computer language. In the LALR (1) parsing, the LR (1) items which have same productions but different look ahead are combined to form a single set of items
- NB: LR means left-to-right, rightmost derivation.

- A compiler is a computer program that translates computer code written in one programming language into another language. It translates source code from a high-level programming language to a lower level language to create an executable program.

## Compiler

| BASIS FOR COMPARISON | COMPILER | INTERPRETER |
|---|---|---|
| Input | It takes an entire program at a time. | It takes a single line of code or instruction at a time. |
| Output | It generates intermediate object code. | It does not produce any intermediate object code. |
| Working mechanism | The compilation is done before execution. | Compilation and execution take place simultaneously. |
| Speed | Comparatively faster | Slower |
| Memory | Memory requirement is more due to the creation of object code. | It requires less memory as it does not create intermediate object code. |
| Errors | Display all errors after compilation, all at the same time. | Displays error of each line one by one. |
| Error detection | Difficult | Easier comparatively |
| Pertaining Programming languages | C, C++, C#, Scala, typescript uses compiler. | PHP, Perl, Python, Ruby uses an interpreter. |

# Compiler vs Interpreter

- Lexical analyzer, syntax analyzer, semantic analyzer, intermediate code generator are work as front end. The front end part  is platform independent because the output of front end is as three address code which is useful for every system.
- Code optimization and Code generator are work as back end. Back end of a typical compiler converts the intermediate representation of program into an executable set of instructions.
- These makes easier the whole compilation process.

## Pass

- **Lexical** : name of some identifier typed incorrectly
- **Syntactical** : missing semicolon or unbalanced parenthesis
- **Semantical** : incompatible value assignment
- **Logical** : code not reachable, infinite loop

## Errors

- A **sentence** is a sentential form consisting only of terminals such as a + a * a.
- A language is a set of **sentences** formed the set of basic symbols.
- A grammar is the set of rules that govern how we determine if these **sentences** are part of the language or not.
- Syntax is the way in which words are put together to form, phrases, clauses or sentences.
- Parse is to resolve a sentence into component parts of speech and describe them grammatically.

# Definitions

- Here,

  $E \rightarrow -E$

  $E \rightarrow -(E)$

  $E \rightarrow -(E+E)$

  $E \rightarrow -(id+id)$

  Hence $-(id+id)$ is a sentence of the grammar $E \rightarrow E + E|E*E|(E)|-E|id$.

# Grammar

- A parse tree is an ordered, rooted tree that represents the syntactic structure of a string according to some context-free grammar.

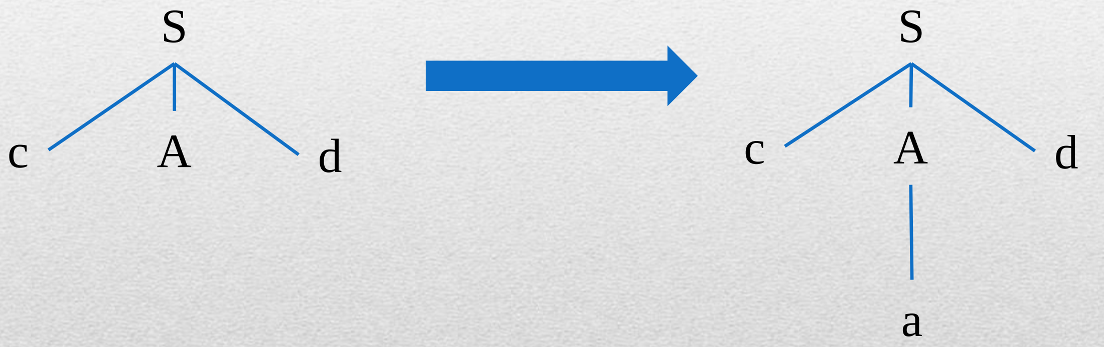
## Parse Tree

## Parse Tree

- Here,

  $S \rightarrow cAd$

  $S \rightarrow cad$



- We can eliminate the left recursion by introducing new non-terminals and new production rules. Algorithm for a left-recursive nonterminal A,
  - Discard any rules of the form $A \rightarrow A$ and consider those that remain:
    $A \rightarrow A\alpha_1|\ldots\ldots|A\alpha_n|\beta_1|\ldots\ldots|\beta_n$ where:
    each $\alpha$ is a nonempty sequence of $\mathrm{non-terminals}$ and terminals, and
    each $\beta$ is a sequence of $\mathrm{non-terminals}$ and terminals that does not start with A.
  - Replace these with two sets of productions.
    One set for A:
    $A \rightarrow \beta_1 A'|\ldots\ldots|\beta_n A'$
    Another set for pure non-terminals,
    $A' \rightarrow \alpha_1 A'|\ldots\ldots|\alpha_n A'|\varepsilon$
  - Repeat this process until no direct left recursion remains.

# Left Recursion

# Left Recursion

- Given,

  $S \rightarrow Aa|b, A \rightarrow Ac|Sd|\varepsilon$

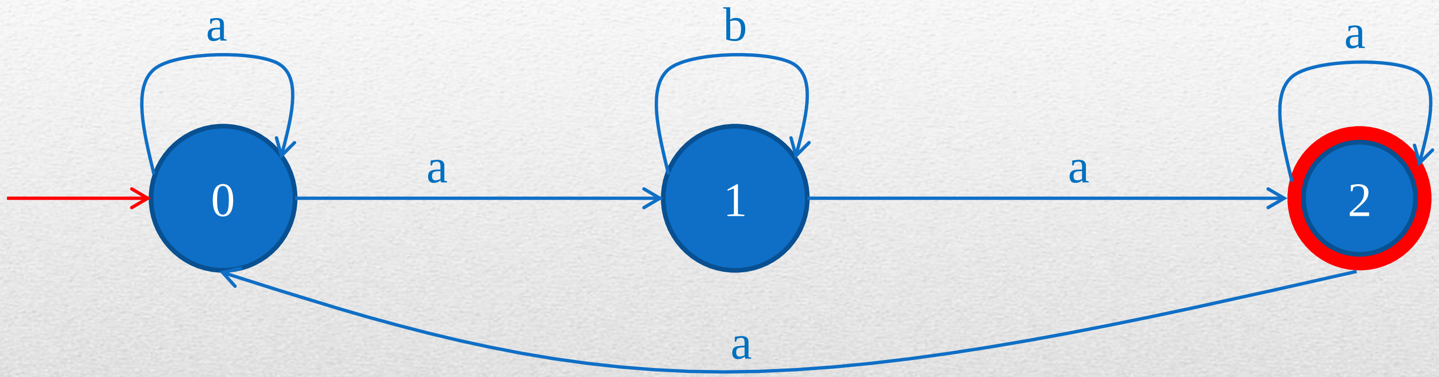
  $\Rightarrow S \rightarrow Aa|b, A \rightarrow Ac|Aad|bd|\varepsilon$

  $\Rightarrow S \rightarrow Aa|b, A \rightarrow AA'|bd|\varepsilon$

  $\Rightarrow S \rightarrow Aa|b, A \rightarrow AA'|bd|\varepsilon, A' \rightarrow c|ad$

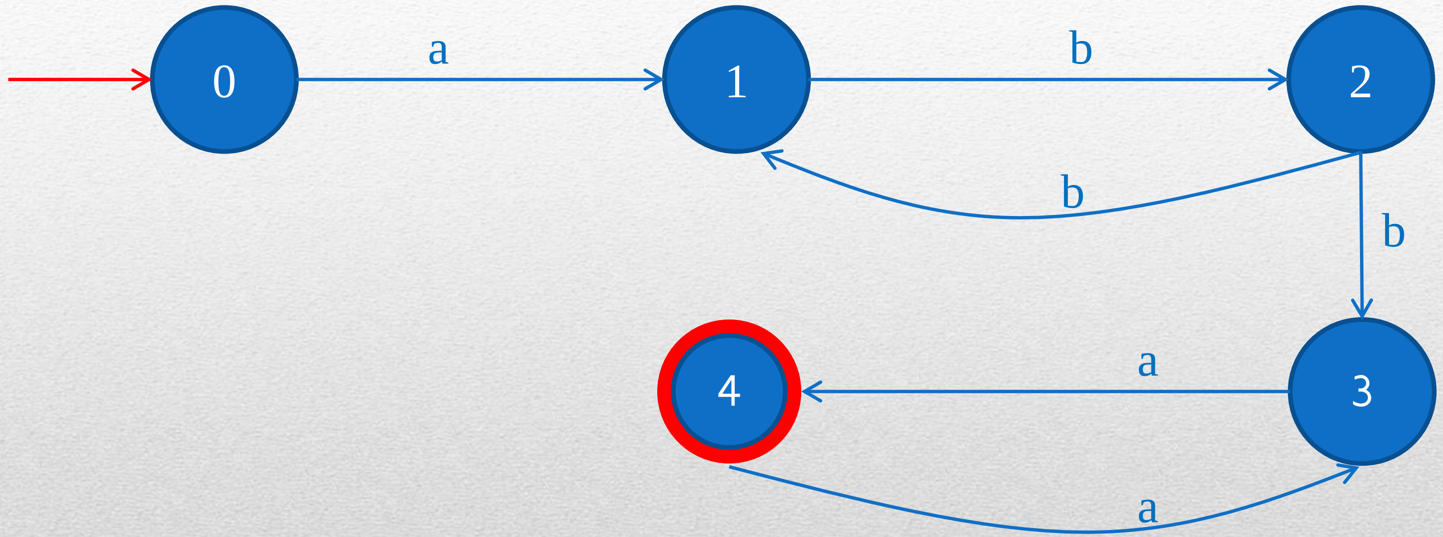
  $Ans: S \rightarrow Aa|b, A \rightarrow AA'|bd|\varepsilon, A' \rightarrow c|ad$

- DFA accepting w∈{a, b}*w starts and ends with same symbol:


$$\begin{tikzcd}
\arrow[r] & 0 \arrow[loop above, "a"] \arrow[r, "a"] & 1 \arrow[loop above, "b"] \arrow[r, "a"] & 2 \arrow[loop above, "a"] \arrow[ll, bend left, "a"]
\end{tikzcd}$$

# D Finite Automata

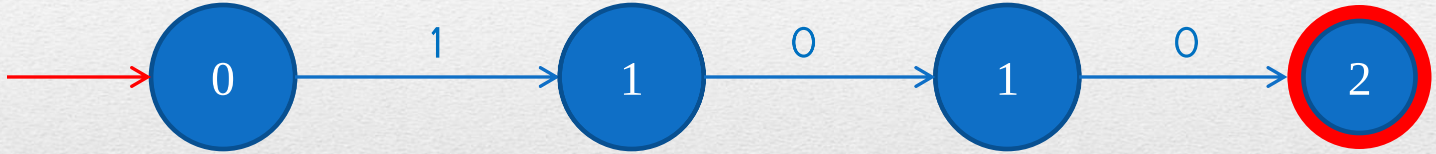
- DFA accepting w∈{a, b}*w has even number of a and even number of b:



# D Finite Automata

# D Finite Automata

- DFA accepting w∈{0, 1}*w is a string interpreted as binary number ≡ mod(4):



- *Lexical analysis* is the process of reading the source text of a program and converting it into a sequence of tokens. A common way to implement a lexical analyzer is to,
  - Specify regular expressions for all of the kinds of tokens in the language. Then, use the alternation operator to create a single regular expression that recognizes the language of all valid tokens.
  - Convert the overall regular expression specifying all possible tokens into a deterministic finite automaton (DFA).
  - Translate the DFA into a program that simulates the DFA. This program is the lexical analyzer.
- This approach is so useful that programs called *lexical analyzer generators* exist to automate the entire process.
- So, If NFA is implemented directly the autometic process won’t be executed.

## Lexical Analyzer

- In NFA, for a particular input symbol, the machine can move to any combination of the states in the machine. In other words, the exact state to which the machine moves cannot be determined. Hence, it is called Non-deterministic Finite Automaton.

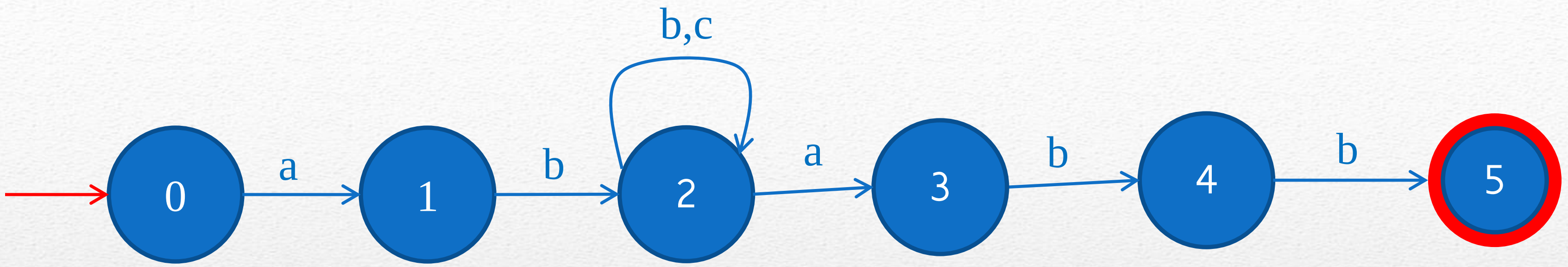
## N Finit Automata

- Why $\varepsilon$ Given a regular expression constructing an equivalent NFA with epsilon is easier than constructing equivalent NFA. Also given two DFAs you can easily construct NFAs with epsilon moves accept concatenation, intersection, union, and Kleene closure of the languages.

## N Finite Automata

- From a regular expression $E$, the obtained automaton $A$ with the transition function $\delta$ respects the following properties:
  - $A$ has exactly one initial state $q_0$.
  - $A$ has exactly one final state $q_f$.
  - Let $c$ be the number of concatenation of the regular expression $E$ and let $s$ be the number of symbols apart from parentheses — that is, |, *, $a$ and $\varepsilon$. Then, the number of states of $A$ is $2s - c$.
  - The number of transitions leaving any state is at most two.

## Thompson’s Construction



Here,

Number of Symbols s = 4 (a, b, |, *)

Number of  Concatenation c = 2

Hence, Number of States = 2s–c =6

# Finite Automata