USN | | | | | | | | | | |

Internal Assessment Test 1 – December2023

Sub:	Automata Theory & Compiler Design					Sub Code:	21CS51	Branch:	AIML/AIDS	
Date:	**18/12/2023**	Duration:	90 min's	Max Marks:	50	Sem/Sec:	V /A		OBE	

Answer any FIVE FULL Questions

		MARKS	CO	RBT
1	Explain the various phases of compiler in detail. 5marks Also explain the language preprocessors. 5 marks	10	CO2	L1

Solution:

A compiler is a software program that translates source code written in a high-level programming language into machine code or an intermediate code that can be executed by a computer. The compilation process involves several phases, each serving a specific purpose. Here are the various phases of a compiler:

Lexical Analysis:

Input: Source code.
Output: Tokens.
This phase is also known as scanning or lexical scanning. The source code is passed through a lexical analyzer, which breaks it into tokens. Tokens are the smallest units in a programming language, such as keywords, identifiers, literals, and operators.
Syntax Analysis:

Input: Tokens.
Output: Abstract Syntax Tree (AST) or Parse Tree.
In this phase, the compiler checks whether the sequence of tokens follows the grammatical structure of the programming language. It generates a hierarchical structure like an Abstract Syntax Tree (AST) or a Parse Tree, representing the syntactic structure of the program.
Semantic Analysis:

Input: AST or Parse Tree.
Output: Intermediate Code.
The semantic analysis phase checks for semantic errors and ensures that the program adheres to the language's semantics. It also performs type checking. If the input program is semantically correct, the compiler generates an intermediate code that captures the essence of the program's logic.
Intermediate Code Generation:

Input: AST or Parse Tree.
Output: Intermediate Code.
The compiler generates an intermediate code representation of the source program. This code is independent of the target machine architecture and serves as an intermediate step between the high-level source code and the machine code. Common intermediate representations include three-address code, quadruples, or bytecode.
Code Optimization:

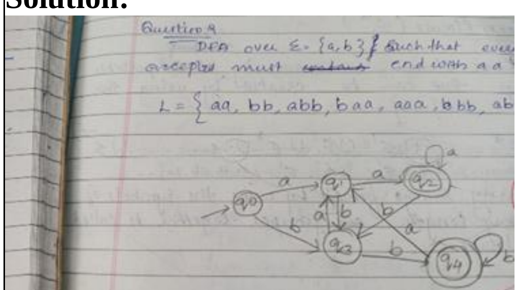
Input: Intermediate Code.
Output: Optimized Intermediate Code.

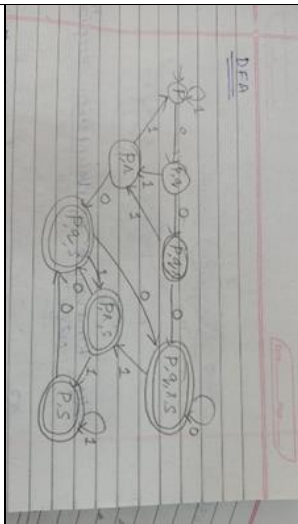
<table>
<tr><td></td><td>This phase improves the intermediate code to enhance the program's efficiency in terms of execution time and/or memory usage. Optimization techniques include constant folding, loop optimization, and dead code elimination.
Code Generation:

Input: Optimized Intermediate Code.
Output: Assembly Code or Machine Code.
The compiler generates the target machine code or assembly code from the optimized intermediate code. This phase involves choosing appropriate machine instructions and allocating registers to variables.
Code Generation:

Input: Assembly Code or Machine Code.
Output: Executable Code.
The compiler generates the final executable code that can be run on the target machine. This phase may involve linking multiple object files, resolving addresses, and producing the final executable file.
Code Optimization (Optional):

Input: Executable Code.
Output: Optimized Executable Code.
Some compilers perform additional optimization on the generated executable code. This phase is optional and depends on the compiler implementation.
Code Generation (Optional):

Input: Optimized Executable Code.
Output: Final Executable Code.
Similar to the optional optimization phase, this step may involve additional code generation for further improvements.</td><td></td><td></td><td></td></tr>
<tr><td>2</td><td>Design a DFA over ∑={a,b} such that every string accepted must end with aa or bb.
Solution:

</td><td>10</td><td>CO2</td><td>L2</td></tr>
<tr><td>3</td><td>Construct a DFA equivalent to NFA({p,q,r,s},{0,1}, δ,p,{s}) where δ is defined as follows:
<table>
<tr><th>δ</th><th>0</th><th>1</th></tr>
<tr><td>p</td><td>{p, q}</td><td>{p}</td></tr>
<tr><td>q</td><td>{r}</td><td>{r}</td></tr>
<tr><td>r</td><td>{s}</td><td>–</td></tr>
<tr><td>s</td><td>{s}</td><td>{s}</td></tr>
</table></td><td>10</td><td>CO2</td><td>L2</td></tr>
</table>



Solution:

4a)	What are the Kleen closure and positive closure in automata theory? 5 marks What are the steps of minimization for finite automata? 5 marks Solution:	5	CO2	L1

Kleene Closure (or Kleene Star):

- **Definition:** In formal language theory, the Kleene closure of a set A of strings, denoted $A*$, is the set of all possible finite-length combinations of zero or more strings from A, including the empty string. It is a fundamental concept in regular expressions and regular languages.

Positive Closure (or Positive Power):

- **Definition:** The positive closure of a set A of strings, denoted $A+$, is the set of all possible finite-length combinations of one or more strings from A, excluding the empty string. It is a variation of the Kleene closure that does not include the empty string.

Minimization of a finite automaton is a process to simplify the structure of the automaton while preserving its language recognition capabilities. The steps for minimizing a finite automaton typically involve the following:

Determine Equivalent States:

Identify pairs of states that can be merged while preserving the equivalence of the language recognized by the automaton. Two states are equivalent if, for any input string, they lead to the same state and accept or reject the input in the same manner.
Initialize Equivalence Classes:

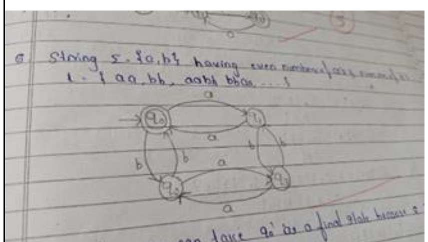
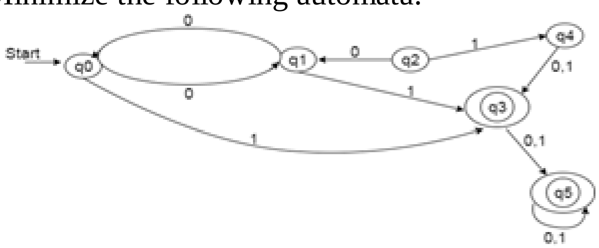
Initially, place states into two classes: accepting and non-accepting states. This is the starting point for the minimization process.
Refine Equivalence Classes:

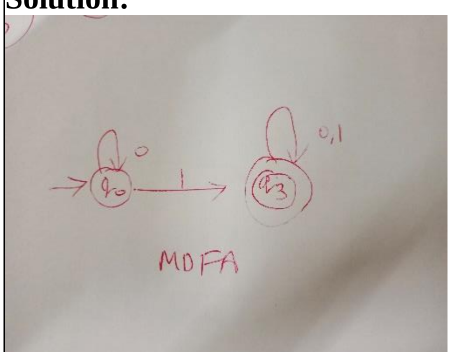
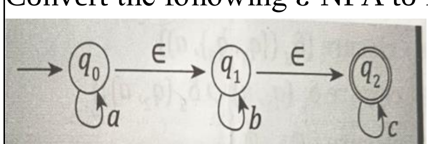
Iteratively refine the equivalence classes by considering the transitions from each state on each input symbol. If two states in the same equivalence class lead to different classes under a specific input symbol, split the equivalence class.
Repeat Until No Further Refinement:

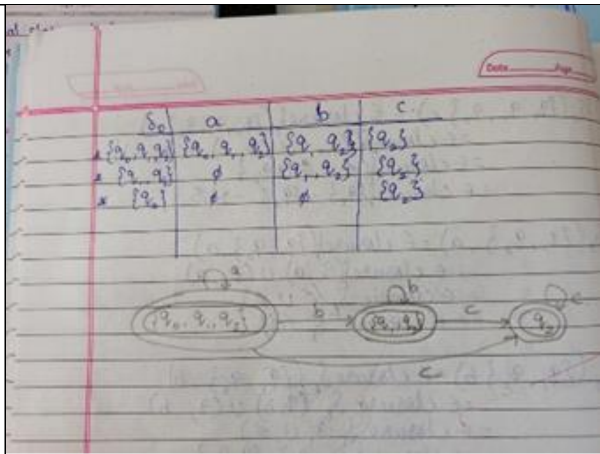
Continue the refinement process until no further changes are possible. At this point,

<table>
<tr><td></td><td>all states within each equivalence class are indistinguishable with respect to the language recognized by the automaton.
Construct the Minimized Automaton:

Create a new automaton with states corresponding to the equivalence classes obtained in the previous steps. The transitions in the new automaton represent transitions between equivalence classes. The start state is the equivalence class containing the original start state, and the accepting states are those equivalence classes containing the original accepting states.
Remove Redundant States and Transitions:

Eliminate any redundant states and transitions in the minimized automaton. This includes removing states that are not reachable from the start state and transitions that do not contribute to the language recognition.</td><td></td><td></td><td></td></tr>
<tr><td>4b)</td><td>Obtain a DFA to accept strings of a’s and b’s having even number of a’s and even number of b’s.
Solution:
</td><td>5</td><td>CO2</td><td>L2</td></tr>
<tr><td>5</td><td>Minimize the following automata.


Solution:

</td><td>10</td><td>CO3</td><td>L3</td></tr>
<tr><td>6</td><td>Convert the following ε-NFA to its equivalent DFA.

</td><td>10</td><td>CO2</td><td>L3</td></tr>
</table>



δ_D	a	b	c
* $\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2\}$
* $\{q_1, q_2\}$	ϕ	$\{q_1, q_2\}$	$\{q_2\}$
* $\{q_2\}$	ϕ	ϕ	$\{q_2\}$

Solution

/

Faculty Signature

CCI Signature

HOD Signature