Assignment - III

1. Construct a turing machine for language

$$L = \{a^m b^{2m} c^m / m \geq 1\}$$

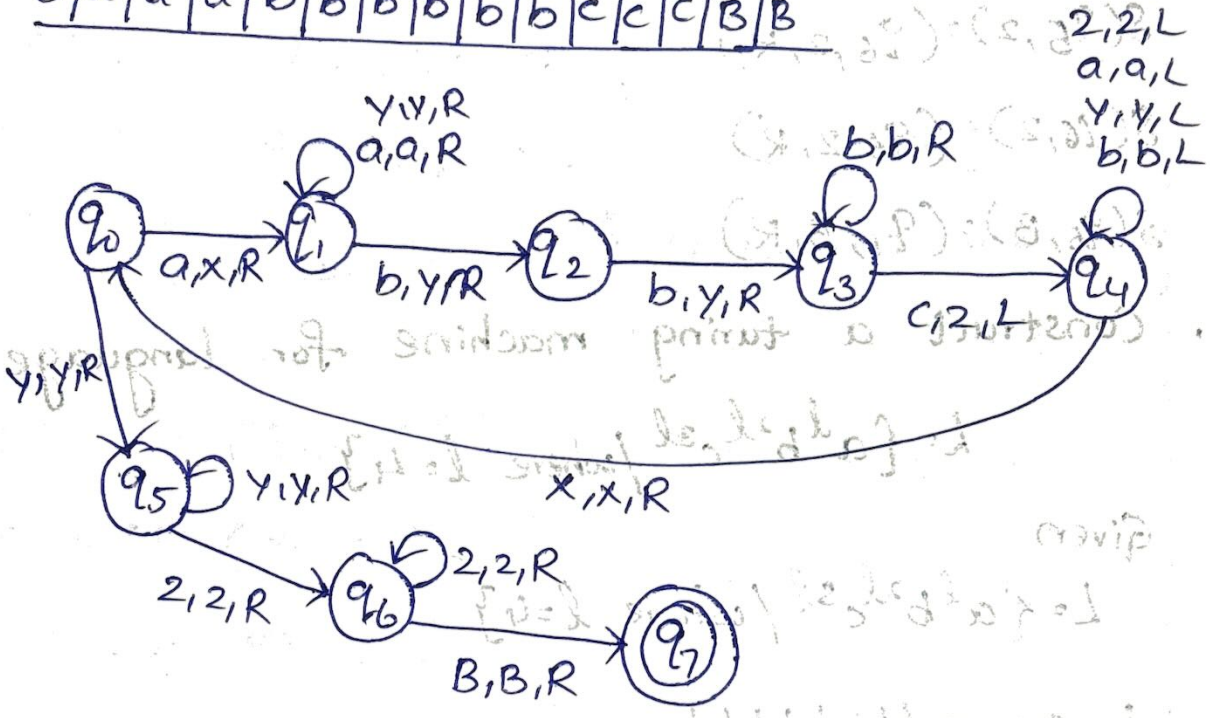
Given

$$L = \{a^m b^{2m} c^m / m \geq 1\}$$

$L = abbc, aabbbbcc, aaabbbbbbcc, ----\}$

$L = aaabbbbbbccc$

X	X	X	Y	Y	Y	Y	Y	Y	2	2	2		
B	a	a	a	b	b	b	b	b	b	c	c	c	B	B



$\delta(q_0, a) = (q_1, X, R)$

$\delta(q_1, a) = (q_1, a, R)$

$\delta(Y, a) =$

$\delta(q_1, Y) = (q_1, Y, R)$

$\delta(q_1, b) = (q_2, Y, R)$

$\delta(q_2, b) = (q_3, Y, R)$

$\delta(q_3, b) = (q_3, b, R)$

$\delta(q_3, c) = (q_4, 2, L)$

$\delta(q_4, b) = (q_4, b, L)$

$\delta(q_4, Y) = (q_4, Y, L)$

$\delta(q_4, a) = (q_4, a, L)$

$\delta(q_4, 2) = (q_4, 2, L)$

$\delta(q_4, X) = (q_0, X, R)$

$\delta(q_0, Y) = (q_5, Y, R)$

$\delta(q_5, Y) = (q_5, Y, R)$

$\delta(q_5, 2) = (q_6, 2, R)$

$\delta(q_6, 2) = (q_6, 2, R)$

$\delta(q_6, B) = (q_7, B, R)$

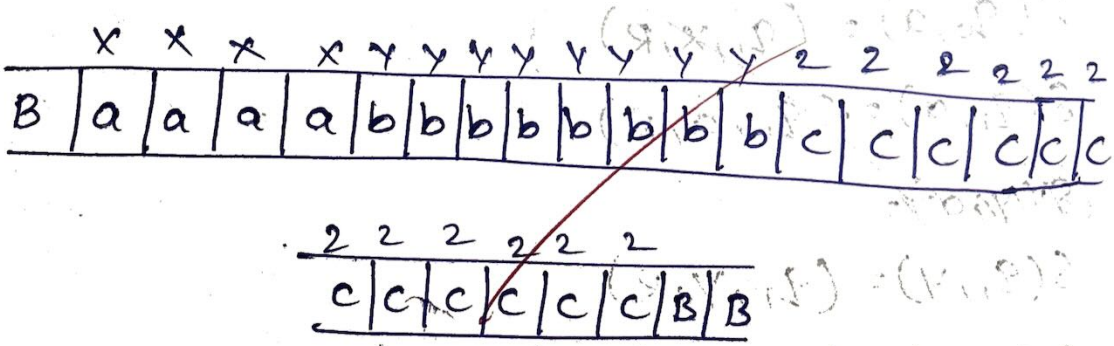
2. Construct a turing machine for language

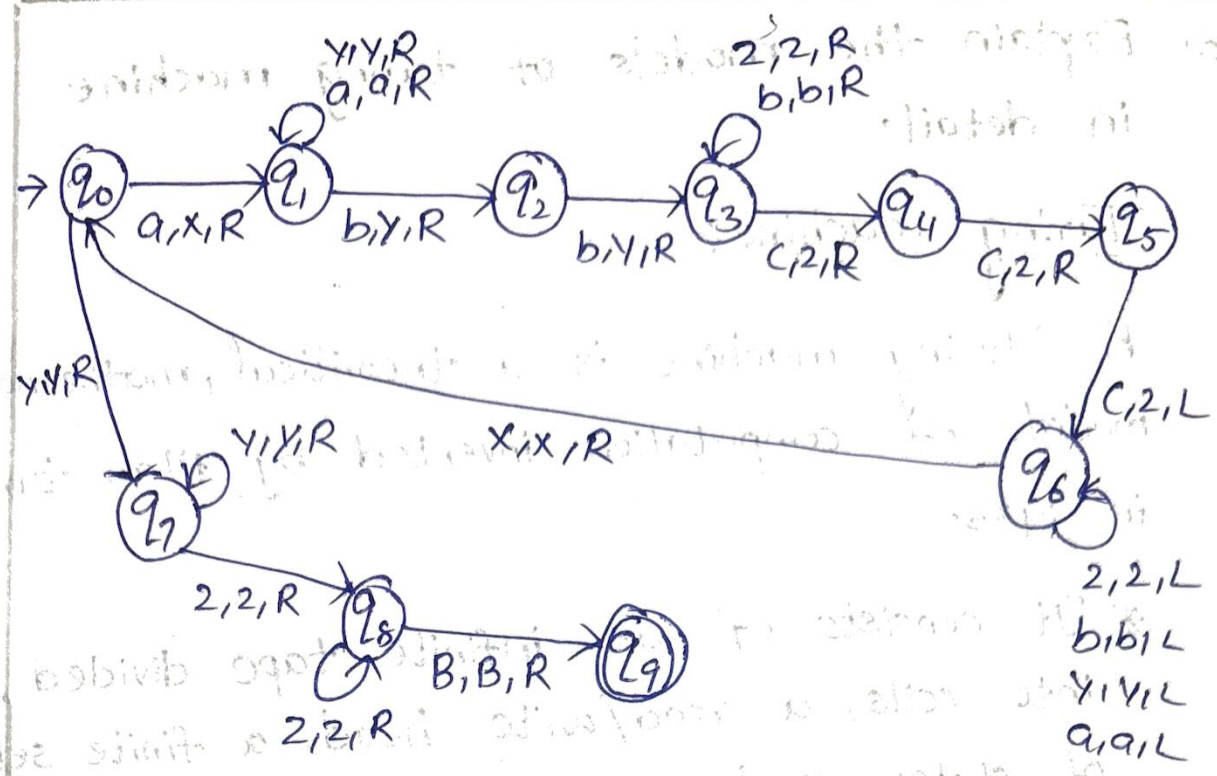
$L = \{a^l b^{2l} c^{3l} / \text{where } l = 4\}$

Given

$L = \{a^l b^{2l} c^{3l} / \text{where } l = 4\}$

$L = aaaa\ bbbbbbbb\ cccccccccccc$





$\delta(q_0, a) = (q_1, X, R)$

$\delta(q_1, a) = (q_1, a, R)$

$\delta(q_1, b) = (q_2, Y, R)$

$\delta(q_2, b) = (q_2, Y, R)$

$\delta(q_3, b) = (q_3, b, R)$

$\delta(q_3, c) = (q_4, Z, R)$

$\delta(q_4, c) = (q_5, Z, R)$

$\delta(q_5, c) = (q_6, 2, L)$

$\delta(q_6, 2) = (q_6, 2, L)$

$\delta(q_6, b) = (q_6, b, L)$

$\delta(q_6, Y) = (q_6, Y, L)$

$\delta(q_6, a) = (q_6, a, L)$

$\delta(q_6, X) = (q_0, X, R)$

$\delta(q_1, Y) = (q_1, Y, R)$

$\delta(q_3, 2) = (q_3, 2, R)$

$\delta(q_0, Y) = (q_7, Y, R)$

$\delta(q_7, Y) = (q_7, Y, R)$

$\delta(q_7, 2) = (q_8, 2, R)$

$\delta(q_8, 2) = (q_8, 2, R)$

$\delta(q_8, B) = (q_9, B, R)$

3. Explain the models of turing machine in detail.

Turing machine :-

A Turing machine is a theoritical, mathematical model of computation invented by ALan turing in 1936.

* It consists of an infinite tape divided into cells, a read/write head, a finite set of states, and a set of rules.
* By reading symbols on the tape, changing its internal state, and moving the head left or right, the Turing machine can perform any calcalation that a real-world computer can, making it an powerful tool for understanding the limits of computation.

Models of turing machine :-

There are 6 types of turing machine:

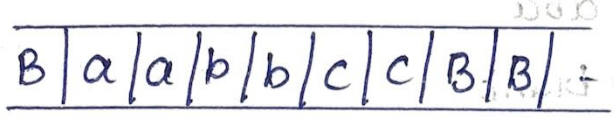
1) Two-way infinite tapes
2) Multi-tape turing machine
3) Non-deterministic turing machine
4) Multi-dimensional turing machine
5) Multi-head turing machine
6) offline Turing machine
7) multi-track turing machine

1) Two-way infinite tape turing machine:-

* Infinite tape of two-way infinite tape turing machine is unbounded in both directions left and right.

* Two-way infinite tape turing machine can be simulated by standard turing machine.

Example:-



Advantages:-

1) The tape being infinite in both directions allow easy processing of data positioned around a central point, making certain algorithms more natural.

2) It simplifies machine construction.

3) Removes left and right boundary issues.

Disadvantages:-

2) Multi-tape turing machine

* A multi-tape Turing machine is a variant where the machine has multiple tapes, each with its own read/write head, allowed it to perform operations on each tape independently and in parallel.

* Multi-tape turing machine have multiple tapes where each tape is assessed with a seperated other heads.

* Initially the input is on tape 1 and others are blank.

Example:-

Checking the string is palindrome or not.

Tape 1: contain aba

Tape 2: Initially blank

operations:-

1. copy the input string from Tape 1 to Tape 2. Tape 2 will contain "aba" as well.
2. position the heads of Tape 1 at the beginning and tape 2 at the end
3. compare the symbols under both heads:
   * If they match, move Tape 1 head right and Tape 2 head left.
   * If at any point the symbols differ reject.
4. Repeat until the heads cross each other or meet.
5. If the entire string matches backwards and forwards, accept.

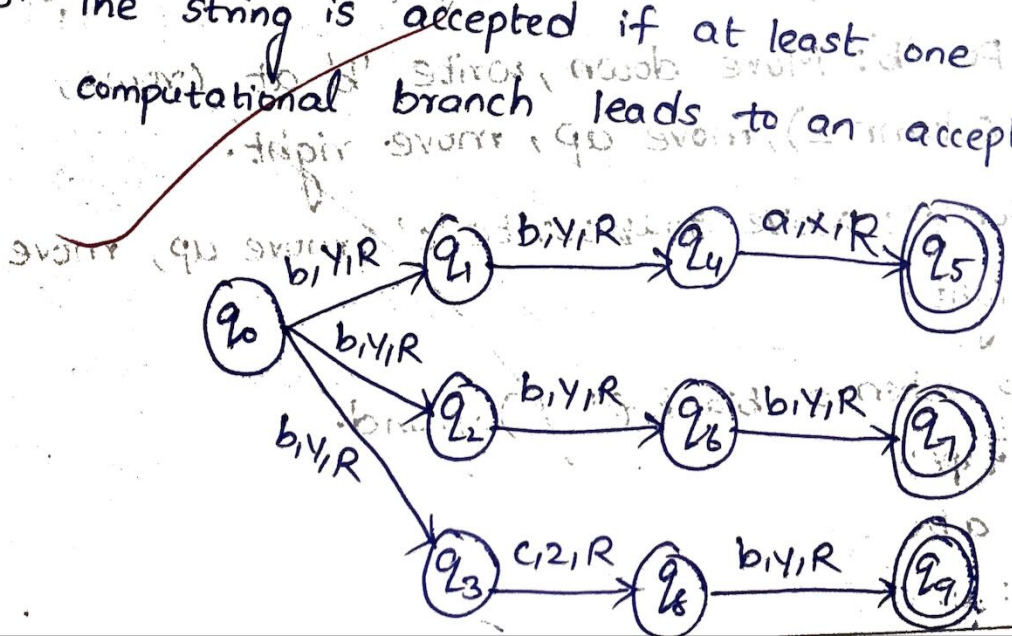
3) Non-deterministic turing machine:-

* A non-deterministic turing machine is a theoretical varient of the standard turing machine in autometa theory.

* In an Non-deterministic turing machine, for a given state and input symbol, the machine can choose from multiple possible transactions instead of just one.

Example:- Accepting string contain atleast one 'a'.

1. At each step, the NTM reads current symbol on the tape.
2. If the symbol is 'a', it has two options:
   1) option 1: Move to the accept state and halt.
   2) option 2: Move right and continue scanning the tape.
3. If symbol b, just move right and continue.
4. If the end of the input is reached without accepting, the machine rejects.
5. The string is accepted if at least one computational branch leads to an accept state.



4) Multi-dimensional turing machine:-

* A multi-dimensional turing machine is a generalization of the standard turing machine where the tape extends in more than one direction, such as a 2D grid instead of a 1D line.
* The multi-dimensional tape allows the head to move right, left, top and down.
* So copying is simpler.

Example:-

Copy all symbols from first row of 2D tape to second row.

Input tape:-

Row1: a b c _

Row2: _ _ _

Steps:-

1) place the head at row1, column1
2) Read the first symbol ('a'), move down write 'a' in (row2, column1), move up, move right.
3) Repeat the process for 'b', 'c':-
   * For 'b': Move down, write 'b' at (row2, column2), move up, move right.
   * For 'c': Move down write 'c', move up, move right.
4) Stop when a blank ('_') found.

Final tape:-

Row1: a b c _

Row2: a b c

5) Multi-head turing machine:-

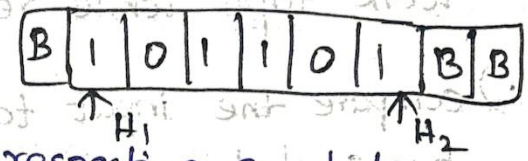
* A multi-head turing machine is a variant of the standard turing machine with multiple read/write heads on a single tape.
* Each head operates independently, allowing simultaneous access to different parts of the tape.

Example:-
checking palindrome (the string is same from forward and backward)

→ Input tape: 101101

Two heads: Head 1 at the leftmost, Head 2 at the rightmost.

steps:-



1) Both heads read their respective symbols.
2) If the symbols match, move head 1 right and Head 2, left and repeat.
3) If the symbols do not match at any step, reject.
4) If the heads cross or meet without mismatch, accept.

6) offline Turing machine:-

* An offline turing machine is a variant of a turing machine where the input is placed on a seperate turing machine that is read-only.
* This contrasts with the standard turing machine where the input and working tape or same.

* The input tape is read-only and seperate.
* The machine cannot modify or erase the input.
* work tapes are used for writing, computation, and intermediate storage.

Example:-

Suppose the task is to check the input string is palindrome or not.

Input tape: Contains the string "aba"
work tape: Initially blank, used to store a reversed copy of the string.

Steps:-

1) copy the input from the input tape to the work tape while reversing it.
2) compare the input tape and work tape symbol by symbol.
3) Accept if all strings match; otherwise reject;

7) Multi-track turing machine:-

* Multi-track turing machines, a specific type of multi-tape turing machine, contain multiple tracks but just one tape head reads and writes on all tracks.
* Here, a single tape head reads n symbols from n tracks at one step.
* It accepts recursively enumerable languages like a normal single-track single-tape turing machine accepts.

Example:-

Doubling the no.of 1's on the tape

→ Track 1 holds the input initially., 111 meaning three one's.

→ output aims to double the no.of one's i.e., 111111 on track2.

steps:-

1) For each '1' on track 1 at the current position, write '1' on track 2 in the same cell.

2) After copying all one's from track 1, more right and write additional one's on track 2 to double the count.

3) Halt after output is completed.

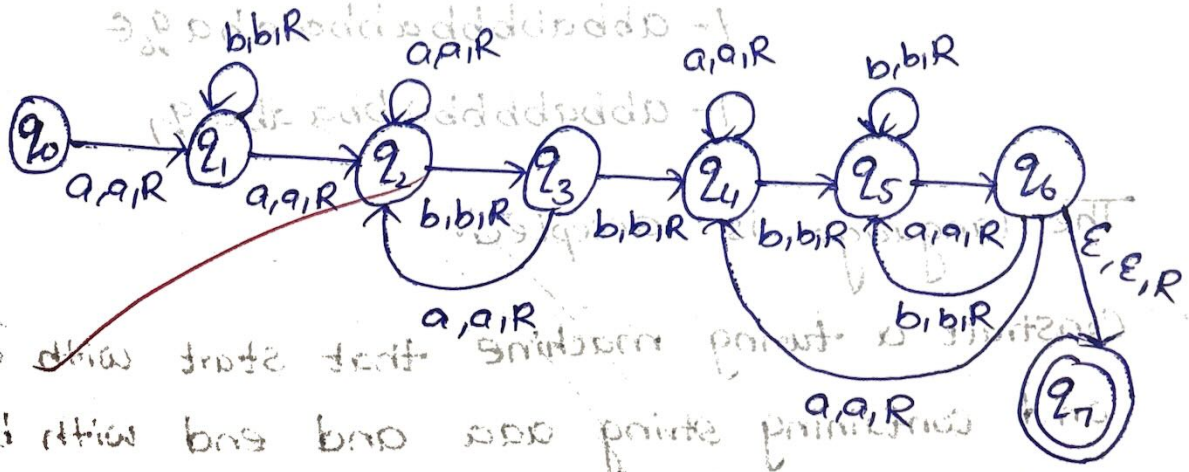
4) Construct a turing machine which start with 'a' and containing string 'abb' and ends with 'ba'. And check the turing machine accept the language abbabbbbabbaaaba.

Given

Start - a

string - abb

end - ba



The language is

abbabbbbbabbaaba

$q_0$abbabbbbbabbaaba |- a q_1 bbabbbbbabbaaba

|- ab q_1 babbbbbabbaaba

|- abb q_1 abbbbbabbaaba

|- abba q_2 bbbbbabbaaba

|- abbab q_3 bbbbabbaaba

|- abbabb q_4 bbabbaaba

|- abbabbb q_5 babbaaba

|- abbabbbb q_5 abbaaba

|- abbabbbba q_6 bbaaba

|- abbabbbbab q_5 baaba

|- abbabbbbabb q_5 aaba

|- abbabbbbabba q_6 aba

|- abbabbbbabba q_4 ba

|- abbabbbbabbaab q_5 a

|- abbabbbbabbaaba q_6 ∈

|- abbabbbbabbaaba q_7

The language is accepted.

5. Construct a turing machine that start with a/b and containing string aaa and end with b. And check whether the turing machine accept a language abbaaabbaaabbb.

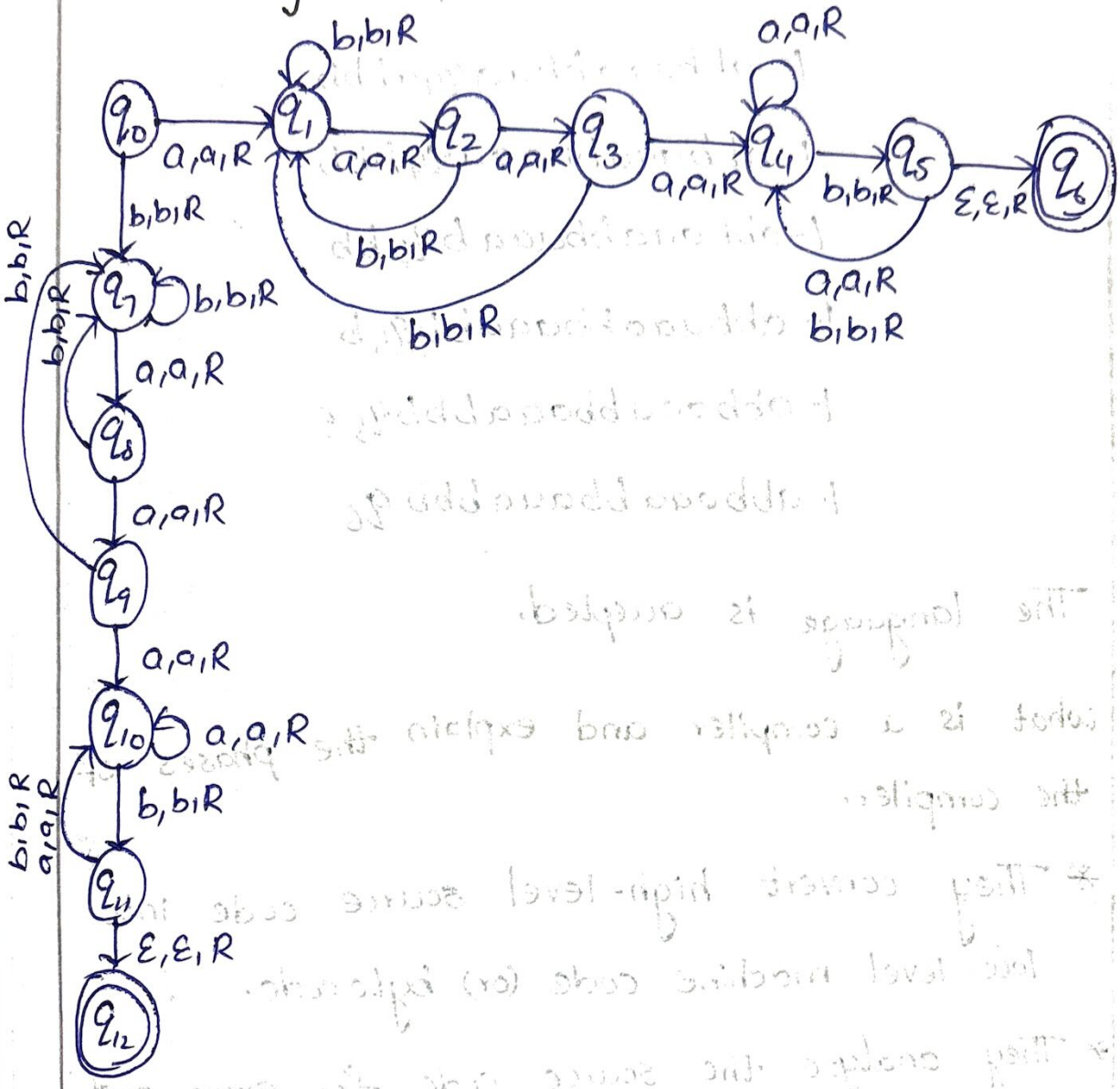
Given

Start - a/b

string - aaa

end - b

The turing machine is



The language is

abbaaaabbaaaabbb.

$q_0$abbaaaabbaaaabbb |- a$q_1$bbaaaabbaaaabbb

|- ab$q_1$baaaabbaaaabbb

|- abb$q_1$aaaabbaaaabbb

|- abba$q_2$aabbaaaabbb

|- abbaa$q_3$abbaaaabbb

$\vdash abbaaa\, q_4 bbaaabbb$

$\vdash abbaaab\, q_5 baaabbb$

$\vdash abbaaabb q_4 aaabbb$

$\vdash abbaaabba q_4 aabbb$

$\vdash abbaaabbaa q_4 abbb$

$\vdash abbaaabbaaa q_4 bbb$

$\vdash abbaaabbaaab q_5 bb$

$\vdash abbaaabbaaabb q_4 b$

$\vdash abbaaabbaaabbb q_5 \epsilon$

$\vdash abbaaa\, bbaaabbb\, q_6$

The language is accepted.

6. What is a compiler and explain the phases of the compiler.

* They convert high-level source code into low-level machine code (or) bytecode.
* They analyze the source code for error and optimize the output code.
* Compilers help in creating efficient, portable and secure executable programs.
* Compilation usually involves multiple stages such as lexical analysis, syntax analysis, sematic analysis, optimization and code generation.
* Some compilers can translate between different programming languages.

* The output code runs faster than interpreted code, which is converted line-by-line during execution.

phases of compiler:-

The phases of compiler is

1) lexical analysis
2) syntax analysis
3) semantic analysis
4) Intermediate code generation
5) code optimization
6) code generation.

1. lexical analysis:-

* Lexical analysis is the first phase of a compiler.
* It takes modified source code from language preprocessors that are written in the form of sentences.
* The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespaces or comments in the source code.
* If the lexical analyzer finds a token invalid, it generates an error.
* The lexical analyzer works closely with the syntax analyzer.
* It reads character streams from the source code, checks for legal tokens, and passes the

data to the syntax analyzer when it demands.

* The tokens contains alphabets, strings, special symbols etc---

Example:-

Consider the statement:

```
x=5;
```

Token	description
x	Identifier (a variable name)
=	Assignment operator
5	Numeric literal
;	Delimiter (semicolon marking the end of statement).

So, the lexical analyzer turns the string x=5; into a sequence of tokens.

<identifier,"x">, <operator,"=">, <literal,"5">
<Delimiter,";">

2. Syntax analyzer:-

* It is second phase of compiler
* It takes the tokens from the lexical analyzer as input.
* It checks the syntactic correctness of the token sequence based on the language's grammer.

* If the arrangement violates the syntax rules, it reports syntax errors with information about their location.

* The syntax analyzer generates a parse tree or syntax tree, which is a hierarchical representation of the syntactic structure of the source code.

* It also helps in building a symbol table that stores information about identifiers and their attributes.

* Syntax analysis can be performed by various parsing techniques, such as top-down parsing or bottom-up parsing.

Example:-

```
int x=10;
```

The syntax analyzer verifies:

1) int is a valid data type keyword.

2) x is a valid identifier.

3) The assignment operator is used correctly.

4) 10 is a valid integer literal.

5) The semicolon ; properly terminates the stmt.

3) Semantic analysis:-

* It is a third phase of compiler.

* Semantic analysis makes sure that declarations and statements of program are semantically correct.

* It is a collection of procedures which is called by parser as and when required by grammer.

* Both syntax tree- of previous phase and symbol table are used to check the consistency of the given code.

Example:-

```
float x = 10.1;
float y = x * 30;
```

semantic analysis will ensure the integer 30 is typecasted to float 30.0 before multiplication to maintain type consistency.

4) Intermediate code generation:-

* It serves as a bridge between analysis phases and synthesis phases.

* Intermediate code is typically easier to generate and optimize compared to directly producing machine code.

* It improves portability because the intermediate code can be adapted to different machine architectures without rewriting the whole compiler.

* Typical forms of intermediate code include three- address code, postfix notation, and other linear or tree based representations.

Example:-

A complex expression like a = b+c*d might be translated into intermediate instructions.

$t_1 = c * d$

$t_2 = b + t_1$

$a = t_2$

* This representation simplifies later optimization and target machine code generation.

5) code optimization:-

*code optimization is a phase in the compiler that aims to improve the performance of the generated code by enhancing its efficiency while preserving its original functionality and semantics. The goal is to produce code that executes faster, uses fewer resources, and reduces overall execution time.

Example:-

Before optimization:-

```
int a=4*i;
int b=4*j;
int c=4*i+n;
```

after optimization:-

```
int a=4*i;
int b=4*j;
int c=a+n;
```

6) Code generation:-

* Code generation is the final phase of a compiler where the intermediate representation of the source code is transformed into machine code or assembly code that can be executed directly by a computer's CPU.

* The code generation phase produces efficient, correct machine code from the intermediate code so that the program can run on the specific hardware for which it is completed.

Example:-

Consider the high-level statement

```
print( 5+7);
```

Code generation for a stack machine:

The generated code steps might be:

```
acc→5        // Load 5 into accumulator
push acc     // push accumulator content on the
               stack
acc→7        // load 7 into the accumulator
acc←acc+top_of_stack // Add top of stack
pop          // pop the accumulator
```

Explain the role of lexical analyzer and input buffering in compiler.

####

* The role of the lexical analyzer in a compiler is crucial as it is the first phase of the compilation process. Its main functions are:

1) It reads the input source code character by character and groups these characters into meaningful sequences called lexemes.

2) It converts these lexemes into tokens, which are the basic building blocks like keywords, identifiers, constants, operators, and punctuation symbol.

3) The lexical analyzer also detects lexical errors such as invalid characters and reports them with precise information, including the line number.

4) It interacts with the parser by providing a stream of tokens for syntax analysis.

5) The lexical analysis may interact with the symbol table to store identifier and constant information.

* The role of lexical analyzer function contains

1) lexical analyzer function
2) Seperation of lexical analyzer from syntax analyzer.
3) Token, lexies and patterns
4) lexical errors

lexical analyzer function:-

Functions of the lexical analyzer in a compiler include:

1) Reading the input: It reads the source program character by character from left to right.

2) Tokenization: It groups input characters into meaningful sequences called tokens.

3) Error detection: It identifies invalid characters or malformed tokens and reports lexical errors.

4) Interacting with symbol table: It may add identifiers and literals to the symbol table.

5) Removing non-essential input

6) providing tokens to the parser

Seperation of lexical analyzer from syntax analyzer:-

The seperation of the lexical analyzer from the syntax analyzer in a compiler is a design principle that improves simplicity, efficiency and portability.

Reasons for seperation:-

1) simplicity

2) Effeciency

3) portability

4) clear interface

* Lexical analyzer: Handles character level processing, forms tokens, removes irrelevant characters, and flags lexical errors.

* Syntax analyzer: Takens tokens from lexical analyzer and checks grammatical correctness, builds parse trees, and reports syntax errors.

* This seperation aligns with the fact that lexical analysis uses regular languages, while syntax analysis uses context-free grammers.

3) Tokens, lexices and patterns:-

Tokens:

A token is a category or class of lexemes that have collective meaning in the source code. Tokens are the basic units that the parser uses. Common types include:

1) keyword

2) Identifiers

3) constants (or) literals

4) operators

5) punctuation

Lexemes:-

A lexeme is a actual sequence of characters in the source code that matches a token. For example, for the token type "identifier", lexemes might be 'x', count or main. It is the concrete instance of a token found in the source program.

patterns:-

A patterns is a rule or description that defines the set of lexemes that belong to a token. pattems are often expressed using regular expressions.

For example, the pattern for an identifier in many language is:

[A-2a-z] [A-2a-2o-9_]*

lexical errors:-

Lexical errors occur during the lexical analysis phase of a compiler when the source code contains sequences of characters that do not match the patterns defined for any valid token.

common types of lexical errors:-

1) illegal characters
2) Exceeding length limits
3) Unmatched strings
4) Invalid identifiers

Input buffering in compiler:-

Input buffering in a compiler is a technique used by the lexical analyzer to efficiently read and process the source program characters.

* Instead of reading characters one by one directly from secondary storage, input buffering reads blocks of characters into a buffer in memory. The lexical analyzer then processes characters from this buffer, significantly reducing costly input/output operations and improving performance.

How it works:-

1) The source code is read into a buffer in chunks.

2) Two pointers are maintained:
   i) Begin pointer (bp or lexemeBegin): points to the start of the current lexeme.
   ii) Forward pointer: Moves ahead to scan characters and identify the end of the current lexeme.

3) When fp reaches the end of the buffer, the other buffer is filled if using two-buffer scheme.

4) Special sentinel characters are placed at buffer ends to identify buffer boundaries efficiently without extra checks.

5) When a lexeme is identified, both pointers move to the next lexeme start.

Buffering schemes:-

1) One-buffer scheme- Uses a single buffer for input. Has issues if lexemes cross buffer boundaries.

2) Two-buffer scheme: Uses two equal-sized buffers that are alternately filled, allowing seamless scanning across buffer boundaries.

Advantages:-

1) Reduces the number of systems calls to read input.

2) Improves scanning speed and compiler efficiency.

3) Simplifies input reading and management logic.