**1. Define PDA.**  
A **Pushdown Automaton (PDA)** is a finite automaton equipped with an additional memory structure called a **stack**.  
It is used to recognize **context-free languages (CFLs)**.  
Formally, PDA = (Q, Σ, Γ, δ, q₀, Z₀, F)

* Q = finite set of states
* Σ = input alphabet
* Γ = stack alphabet
* δ = transition function
* q₀ = start state
* Z₀ = initial stack symbol
* F = set of final states

**2. State the Pumping Lemma for CFLs.**  
If **L** is an infinite context-free language, then there exists a constant **p** (pumping length) such that any string s in L with |s| ≥ p can be written as  
s = uvwxy,  
satisfying:

1. |vwx| ≤ p
2. |vx| ≥ 1
3. For all i ≥ 0, **uvⁱwxⁱy ∈ L**

**3. List components of the Turing Machine.**  
A **Turing Machine (TM)** consists of:

1. A finite set of **states** (Q).
2. An **input alphabet** (Σ).
3. A **tape alphabet** (Γ), including blank symbol (B).
4. A **transition function** δ : Q × Γ → Q × Γ × {L, R}.
5. A **tape** (infinite in both directions).
6. A **tape head** (that can read, write, and move left/right).
7. A **start state** (q₀).
8. A set of **accepting (final) states** (F).

**4. Differentiate between Compiler and Interpreter.**

| **Compiler** | **Interpreter** |
| --- | --- |
| Translates the entire program into machine code at once. | Translates and executes line by line. |
| Output: object/executable file. | No separate object file. |
| Faster execution (once compiled). | Slower execution. |
| Example: C, C++ compilers. | Example: Python, JavaScript interpreters. |

**5. What tables are maintained by an assembler during translation? (GATE 2016)**  
Assemblers maintain:

1. **Symbol Table** – stores labels/symbols and their addresses.
2. **Literal Table** – stores constants/literals and their addresses.
3. **Pool Table** – used for handling multiple literal pools.
4. **Opcode Table (OPTAB)** – contains machine opcodes and their binary representation.
   1. **Construct the Syntax tree for the source program a = b \* c + d \* f.**

=

/ \

a +

/ \

\* \*

/ \ / \

b c d f

* 1. **Consider the following statement: x = y + z \* 75;**

Tokens are the smallest units recognized by a compiler.

* x → identifier
* = → operator
* y → identifier
* + → operator
* z → identifier
* \* → operator
* 75 → constant
* ; → delimiter

**Total tokens = 8**

**8. What is intermediate code generation? Why is it used in modern compilers instead of translating directly into machine code?**

**Intermediate Code Generation (ICG):**  
The compiler translates the source code into an intermediate representation (IR) that is independent of machine architecture.  
Example: Three-address code, postfix notation.

* **Why used?**

1. Makes **compiler design easier** (separates front-end from back-end).
2. Provides **portability** (same front-end for multiple machines).
3. Enables **optimization** at IR level.
4. Simplifies **target code generation**.

**9. Define Lex.**

* **Lex** is a lexical analyzer generator.
* It automatically generates a **lexical analyzer (scanner)** from regular expressions given in its input.
* Lex converts the input source program into a sequence of **tokens** which are passed to the parser.
  1. Write the two approaches of language acceptance by PDA.

Two approaches for language acceptance by a PDA  
A Pushdown Automaton (PDA) can accept strings by either:

1. **Acceptance by final state**
   * The PDA processes the entire input and halts in one of its designated accepting (final) states.
   * Formally: input w is accepted if ∃ a computation that consumes w and ends in a state ∈ F (regardless of stack contents).
2. **Acceptance by empty stack**
   * The PDA accepts if, after consuming the entire input, its stack is empty (irrespective of the current state).
   * Formally: input w is accepted if ∃ a computation that consumes w and leaves the stack with only the initial bottom symbol removed (or empty as defined).
   1. What are the common data structures used to implement a symbol table?

 **Hash table** — average O(1) lookup/insert; most commonly used in compilers.

 **Binary Search Tree (BST) / Balanced BST (AVL, Red–Black)** — O(log n) worst-case lookup/insert; useful if ordered traversal needed.

 **Trie (Prefix tree)** — good for storing identifiers with common prefixes; lookup proportional to identifier length.

 **Linked list / Array (simple)** — O(n) lookup; used only for very small scopes or simple educational implementations.

 **Scoped symbol tables (stack of hash tables)** — to support nested scopes: a stack of tables, each table implemented by hash or BST.

 **Perfect hashing / Cuckoo hashing** — variants for performance-critical implementations.

 **Disk-based structures (B-tree)** — for very large symbol tables (rare in typical compilers).

* 1. An arithmetic expression with an unbalanced parenthesis is a lexical or syntax error. Justify your answer.

It is a **syntax error**.

**Why:**

* Lexical analysis (scanner) divides the input into tokens (identifiers, numbers, operators, parentheses, etc.). The presence of a parenthesis character ( or ) is recognized at lexical stage as a parenthesis token — no lexical rule fails because the character exists.
* **Balance** of parentheses is a structural property of the token sequence (matching opening and closing parentheses) and must be checked by the parser (syntax analyzer) or by a context-free grammar. Unbalanced parentheses violate grammar rules (well-formedness), so they are detected during **syntax analysis** (or semantic/structural checks), not during lexing.
  1. Construct the Syntax tree for the source program SI=P\*N\*R /100.

Operator precedence and left associativity: \* and / have same precedence and are left-associative. So P \* N \* R / 100 parses as (((P \* N) \* R) / 100).

syntax tree:

=

/ \

SI /

/ \

\* 100

/ \

\* R

/ \

P N

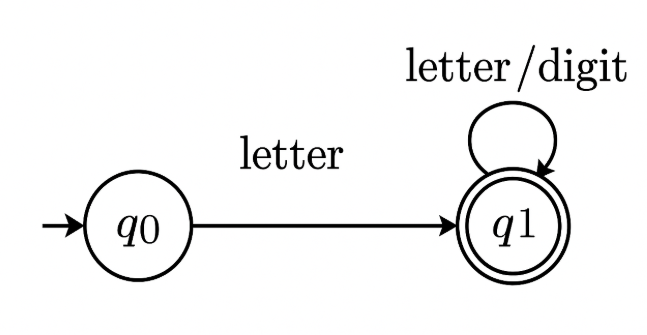
* 1. Draw the transition diagram for identifier, white space.

(a) **Identifier** (typical rule: identifier = letter (letter | digit)\*)  
States:

* q0 = start (non-accepting)
* q1 = accepting identifier state

Transitions:

* q0 -- letter --> q1
* q1 -- letter/digit --> q1 (loop)
* Any other character from q0 => tokenization fails or go to other token recognizers.



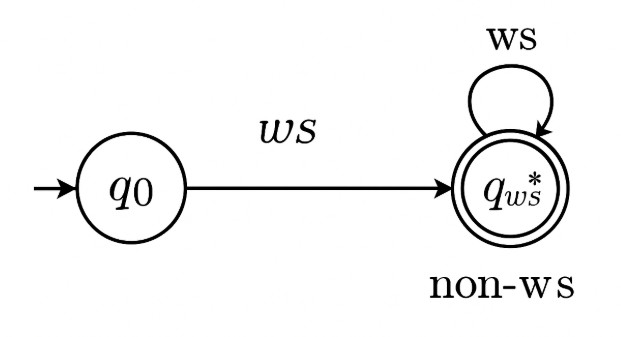
(b) **White space** (spaces, tabs, newlines) — usually skipped (scanner consumes and ignores):

States:

* q0 = start
* qws = white-space consuming state (accepting but action = ignore)

Transitions:

* q0 -- whitespace --> qws
* qws -- whitespace --> qws (loop)
* qws -- non-whitespace --> return to q0 (and reprocess non-space in tokenizer)



* 1. Write the Structure of the Lex Program.
* A Lex program consists of three parts and is separated by % delimiters:-
* Declarations  
  %%  
  Translation rules  
  %%  
  Auxiliary procedures
* **Declarations:**The declarations include declarations of variables.
* **Transition rules:** These rules consist of Pattern and Action.
* **Auxiliary procedures:** The Auxilary section holds auxiliary functions used in the actions.

For example:

* **declaration**  
  number[0-9]  
  %%  
  **translation**  
  if {return (IF);}  
  %%  
  **auxiliary function**  
  int numberSum()

PART B

|  |  |
| --- | --- |
| 1 | Prove that **L = {WW|W ∈ {a,b}**is not Context Free Language. |

Assume LLL is context-free. Then by the pumping lemma for CFLs there exists a pumping length p≥1p\ge1p≥1 such that any string s∈Ls\in Ls∈L with ∣s∣≥p|s|\ge p∣s∣≥p can be written s=uvwxys = u v w x ys=uvwxy with

1. ∣vwx∣≤p|vwx|\le p∣vwx∣≤p,
2. ∣vx∣≥1|vx|\ge 1∣vx∣≥1,
3. ∀i≥0,  uviwxiy∈L\forall i\ge0,\; u v^{i} w x^{i} y \in L∀i≥0,uviwxiy∈L.

Choose the specific string

s=apbp  apbp.s = a^{p} b^{p} \; a^{p} b^{p}.s=apbpapbp.

This sss is in LLL because s=WWs = W Ws=WW with W=apbpW = a^{p} b^{p}W=apbp. Also ∣s∣=4p≥p|s| = 4p \ge p∣s∣=4p≥p.

By the pumping lemma take a decomposition s=uvwxys = u v w x ys=uvwxy satisfying (1) and (2). Because ∣vwx∣≤p|vwx|\le p∣vwx∣≤p, the substring vwxvwxvwx lies entirely inside a block of length ppp in the concatenation apbpapbpa^{p} b^{p} a^{p} b^{p}apbpapbp. Therefore vwxvwxvwx must lie entirely in one of these four blocks:

* the first apa^{p}ap (block 1), or
* the first bpb^{p}bp (block 2), or
* the second apa^{p}ap (block 3), or
* the second bpb^{p}bp (block 4).

(It cannot simultaneously affect corresponding positions in both halves because each block is length ppp and ∣vwx∣≤p|vwx|\le p∣vwx∣≤p.)

Now examine pumping i=0i=0i=0 (remove vvv and xxx). Removing ∣vx∣≥1|vx|\ge1∣vx∣≥1 symbols from only one block changes the content or length of *one half* of the string WWWWWW without identically changing the corresponding portion of the other half. Concretely:

* If vwxvwxvwx lies entirely in block 1 or 2 (i.e. inside the **first half** WWW), then after pumping down (i=0i=0i=0) the first half is shortened or its symbol sequence altered while the second half remains exactly apbpa^{p}b^{p}apbp. So the result cannot be of the form UUUUUU (two identical halves). Contradiction.
* If vwxvwxvwx lies entirely in block 3 or 4 (inside the **second half**), then pumping down changes the second half but not the first half; again the two halves become different, so the pumped string is not in LLL. Contradiction.

In all cases, pumping produces a string not in LLL, contradicting the pumping lemma condition (3). Hence our assumption that LLL is context-free is false.

Therefore L={WW∣W∈{a,b}∗}L=\{WW\mid W\in\{a,b\}^\*\}L={WW∣W∈{a,b}∗} is **not** a context-free language. ∎

|  |  |
| --- | --- |
| 2) | Design a PDA which accepts L = {anb2n|n>1}  So, the strings which are generated by the given language are as follows −  L={abb,aabbbb,aaabbbbbb,….}  Here a’s are followed by double the b’s  Whenever ‘a’ comes, push ‘a’ two times in the stack and if ‘a’ comes again then do the same.  When ‘b’ comes then pop one ‘a’ from the stack each time. Note that b comes after ‘a’.  Finally at the end of the strings, if nothing is left in the STACK, then we can declare that language is accepted in the PDA.  The **PDA** for the problem is as follows −  https://www.tutorialspoint.com/assets/questions/media/53285/infinite_amount.jpg  Transition functions  The transition functions are as follows −  Step 1: δ(q0, a, Z) = (q0, aaZ)  Step 2: δ(q0, a, a) = (q0, aaa)  Step 3: δ(q0, b, a) = (q1, ε)  Step 4: δ(q1, b, a) = (q1, ε)  Step 5: δ(q1, ε, Z) = (qf, Z)  Explanation  **Step 1** − Consider input string: "aabbbb" which satisfies the given condition.  **Step 2** − Scan string from left to right.  **Step 3** − For input 'a' and STACK alphabet Z, then  **Step 4** − For input 'a' and STACK alphabet 'a', then  Push the two 'a's into STACK: (a,a/aaa) and state will be q0. Now the STACK has "aaaa".  **Step 5** − For input 'b' and STACK alphabet 'a', then  Pop one 'a' from STACK: (b,a/ε) and state will be q1.  **Step 6** − For input 'b' and STACK alphabet 'a' and state q1, then  Pop one 'a' from STACK: (b,a/ε) and state will remain q1  **Step 7** − For input 'b' and STACK alphabet 'a', then  Pop one 'a' from STACK: (b,a/ε) and state will be q1  **Step 8** − For input 'b' and STACK alphabet 'a' and state q1, then  Pop one 'a' from STACK: (b,a/ε) and state will remain q1  **Step 9** − We reached end of the string, for input ε and STACK alphabet Z,  Go to final state(qf): (ε, Z/Z) |

1. Design a Turing Machine that accepts **palindromes** over the alphabet {a, b}.

## Algorithm

**Step 1 -** If there is no input, reach the final state and halt.

**Step 2 -** If the input = “a?, then traverse forward to process the last symbol = “a?. Convert both a?s to B?.

**Step 3 -** Move left to read the next symbol.

**Step 4 -** If the input = “b?, replace it by B and move right to process its equivalent “B? at the rightmost end.

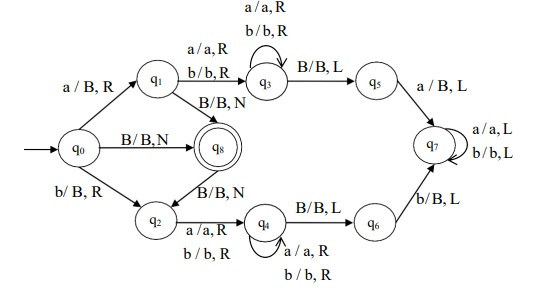
**Step 5 -** Convert the last ’b’ to ‘B’.

**Step 6 -** Move left and process step 2 – 5 until there are no more inputs to process.

**Step 7 -** If the machine reaches the final state after processing the entire input string, then the string is a palindrome that halts the machine.

## Turing Machine

The turing machine is as follows −

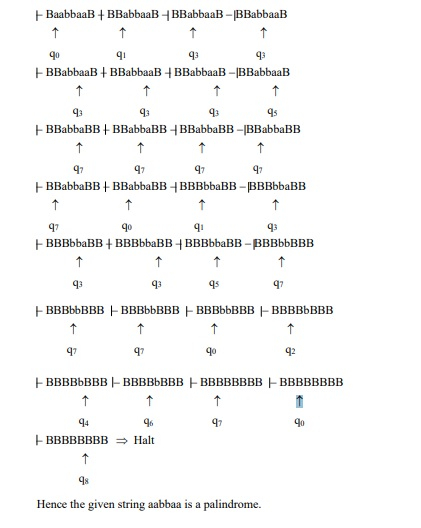


The Turing machine, M is given by M = (Q, Σ, Γ, δ, q0, B, F)

Where,

* Q = {q0, q1, q2, q3, q4, q5, q6, q7, q8}
* Σ = {a, b}
* Γ = {a, b, B}
* δ ⇒ Given by the above mentioned transition diagram,
* q0 = {q0}
* B = {B}
* F = {q8}

Consider a string aabbaa, as shown below −



1. Summarize in detail about how the tokens are specified by the compiler with a suitable example.

* A **token** is the smallest unit in a source program that has **meaning** to the compiler.
* Example token categories:
  + **Keywords** (if, while, int, …)
  + **Identifiers** (names given by user, e.g., x, sum, value1)
  + **Constants** (numeric literals like 10, 3.14, 'A')
  + **Operators** (+, -, \*, =, ==)
  + **Delimiters / punctuation** (;, ,, {, })

## How Tokens are Specified

The compiler specifies tokens using **patterns** and **lexemes**:

1. **Lexeme**
   * A lexeme is the **actual character sequence** in the source program that matches a token pattern.
   * Example: in the statement int x = 10; → the lexeme int is a keyword.
2. **Token**
   * A token is a **pair**: <token-name, attribute-value>
   * Example: <identifier, x> or <number, 10>
3. **Patterns**
   * Each token has a **pattern**, usually described by **regular expressions**.
   * Examples:
     + Keyword: fixed set (if|else|while|for|int|float)
     + Identifier: [A-Za-z][A-Za-z0-9]\*
     + Number: [0-9]+
     + Whitespace: [ \t\n]+

## Example

Consider the input line:

sum = a + b \* 10;

The lexical analyzer scans and classifies tokens:

| **Lexeme** | **Token Form (type, value)** | **Pattern** |
| --- | --- | --- |
| sum | <id, sum> | [A-Za-z][A-Za-z0-9]\* |
| = | <assign\_op> | = |
| a | <id, a> | [A-Za-z][A-Za-z0-9]\* |
| + | <plus\_op> | + |
| b | <id, b> | [A-Za-z][A-Za-z0-9]\* |
| \* | <mult\_op> | \* |
| 10 | <num, 10> | [0-9]+ |
| ; | <semicolon> | ; |

So the sequence of tokens is:

<id, sum> <assign\_op> <id, a> <plus\_op> <id, b> <mult\_op> <num, 10> <semicolon>

## Process of Token Specification

1. **Lexical rules** are written (using regex or grammars).
2. A tool like **Lex/Flex** generates the **scanner (lexer)**.
3. The lexer reads the input stream character by character, groups them into lexemes, matches them to patterns, and produces tokens.
4. These tokens are passed to the **parser** for syntax analysis.
5. Explain in detail the phases of the compiler for the source program **Position= initial + rate \* 60.**

# 1. **Lexical Analysis (Scanning)**

* **Task:** Convert the stream of characters into **tokens**.
* **Input:** Raw program text.
* **Output:** Sequence of tokens.

From the statement:

| **Lexeme** | **Token (type, value)** |
| --- | --- |
| Position | <id, Position> |
| = | <assign\_op> |
| initial | <id, initial> |
| + | <plus\_op> |
| rate | <id, rate> |
| \* | <mult\_op> |
| 60 | <num, 60> |
| ; | <semicolon> |

Tokens are passed to the parser.

# 🔹 2. **Syntax Analysis (Parsing)**

* **Task:** Check tokens against **grammar rules** of the language and build a **parse tree / syntax tree**.
* **Grammar rule (simplified):**
* stmt → id = expr ;
* expr → expr + term | term
* term → term \* factor | factor
* factor → id | num

**Syntax Tree:**

=

/ \

Position +

/ \

initial \*

/ \

rate 60

# 🔹 3. **Semantic Analysis**

* **Task:** Check for **semantic correctness**:
  + Type checking
  + Scope checking
  + Consistency

**Examples in this case:**

* Position, initial, and rate must be declared identifiers.
* rate \* 60 must be type-compatible (if rate is float, 60 promoted to float).
* Assignment type: type(Position) = type(initial + rate \* 60).

If mismatched, compiler reports **semantic error**.

# 🔹 4. **Intermediate Code Generation (ICG)**

* **Task:** Translate into an **intermediate representation (IR)**, often in **Three-Address Code (TAC)**.

Example IR (TAC):

t1 = rate \* 60

t2 = initial + t1

Position = t2

This form is independent of machine but easier to optimize.

# 🔹 5. **Code Optimization**

* **Task:** Improve IR without changing meaning.

**Possible optimizations:**

* Constant folding: if rate known, may simplify at compile time.
* Strength reduction, dead-code elimination, register allocation.

Example optimization (if 60 is constant):  
t1 = rate \* 60 might be transformed into efficient machine-level multiplication (or shift+add if possible).

# 🔹 6. **Target Code Generation**

* **Task:** Translate optimized IR into **assembly/machine code** for the target machine.

Example (x86-style assembly, simplified):

MOV R1, rate

MUL R1, 60

ADD R1, initial

MOV Position, R1

# 🔹 7. **Symbol Table Management** (runs throughout)

* The compiler builds and maintains a **symbol table** storing info about identifiers:
  + Position, initial, rate → names, types, scope, memory location.

# 🔹 8. **Error Handling** (cross-cutting)

* At each stage:
  + **Lexical:** illegal character → “Unknown symbol @”
  + **Syntax:** missing ; → “Syntax error near token”
  + **Semantic:** undeclared variable → “Undeclared identifier: rate”

# Design a TM to compute addition of two unary numbers

The unary input number n is represented with a symbol 0 n – times.

## Example

* 4 → 0000
* 1 → 0
* 5 → 00000

The separation symbol, „#? (any other special character) shall be used to distinguish between two or more inputs.

For Example: 5, 2 are the inputs represented by 00000 # 00.

## Algorithm

**Step 1 -** Read the symbols of the first input with no replacements and move right.

**Step 2 -** When the symbol = ‘#’, replace it by ‘0’ and move right.

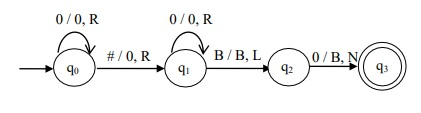
**Step 3  -** Traverse right side until the rightmost ‘0’ (left to B – last symbol)

**Step 4 -** Replace the rightmost ‘0’ by B

**Step 5 -** Stop the machine.

## Turing Machine

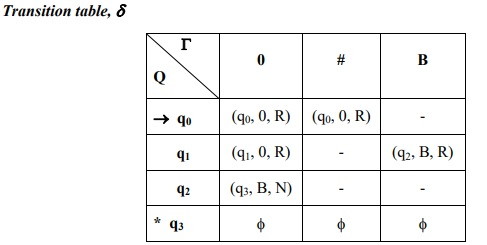
The Turing Machine (TM) is as follows −



The Turing machine, M is given by M = (Q, ∑, Γ, δ, q0, B, F)

Where,

* Q = {q0, q1, q2, q3}
* Σ = {0, #}
* Γ = {0, #, B}
* δ ⇒ Given by the above transition diagram q
* 0 = {q0}
* B = {B}
* F = {q3}



1. Design a Turing Mahine for f(w)=w^R (i.e., reverses the input string) over Σ={a,b}

## Example

**Input** − aabbab

**Output** − babbaa

## Algorithm

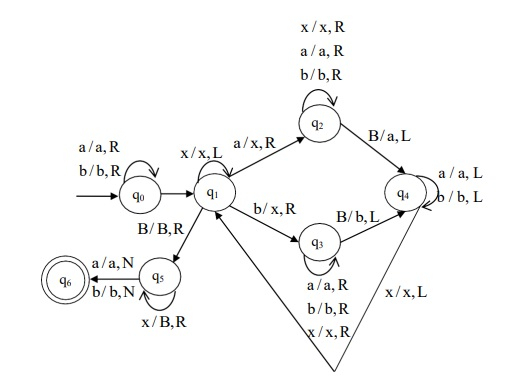
Step 1: Move to the last symbol, replace x for a or x for b and move right to convert the corresponding B to          „a? or „b? accordingly.

Step 2: Move left until the symbol left to x is reached.

Step 3: Perform step 1 and step 2 until „B? is reached while traversing left.

Step 4: Replace every x to B to make the cells empty since the reverse of the string is performed by the             previous steps.

The transition diagram for Turing Machine (TM) is as follows −

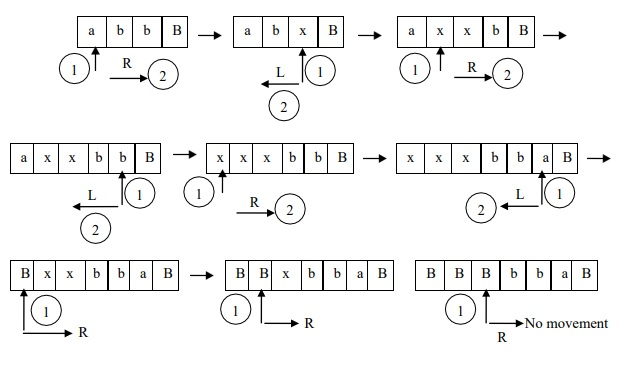


**Let us consider an example to check whether the given input is getting reverse or not**−

**Input string -** abb.

**Expected output -** {bba}

Therefore, the TM is as follows −



1. Prove that **L = {anbnci | i<n},** is not Context-Free Language.

### umping-lemma setup

Assume, for contradiction, that LLL is context-free. Then there is a pumping length p≥1p\ge1p≥1 such that any string s∈Ls\in Ls∈L with ∣s∣≥p|s|\ge p∣s∣≥p can be written

s=u v w x ys = u\,v\,w\,x\,ys=uvwxy

with

1. ∣vwx∣≤p|vwx|\le p∣vwx∣≤p,
2. ∣vx∣≥1|vx|\ge1∣vx∣≥1,
3. ∀t≥0,  u vt w xt y∈L\forall t\ge0,\; u\,v^{t}\,w\,x^{t}\,y \in L∀t≥0,uvtwxty∈L.

Choose the specific string

s=ap bp c p−1.s = a^{p}\,b^{p}\,c^{\,p-1}.s=apbpcp−1.

Note s∈Ls\in Ls∈L because i=p−1<n=pi=p-1<n=pi=p−1<n=p, and ∣s∣=3p−1≥p|s|=3p-1\ge p∣s∣=3p−1≥p.

Because ∣vwx∣≤p|vwx|\le p∣vwx∣≤p, the substring vwxvwxvwx cannot stretch over all three regions apa^{p}ap, bpb^{p}bp, and cp−1c^{p-1}cp−1. It must lie entirely within one of these regions or at most cross one boundary (i.e., be inside apa^pap, inside bpb^pbp, inside cp−1c^{p-1}cp−1, across the a ⁣− ⁣ba\!-\!ba−b boundary, or across the b ⁣− ⁣cb\!-\!cb−c boundary). Consider these cases.

### Case 1 — vwxvwxvwx lies entirely inside the apa^{p}ap block

Then vvv and/or xxx consist only of a's. Pick t=0t=0t=0. The pumped string is

u w y=ap−ta bp cp−1u\,w\,y = a^{p-t\_a}\,b^{p}\,c^{p-1}uwy=ap−ta​bpcp−1

for some ta≥1t\_a\ge1ta​≥1. Now the number of a's is p−tap-t\_ap−ta​ but the number of b's remains ppp. So a and b counts are unequal; the string cannot be of the form anbncia^{n}b^{n}c^{i}anbnci. Thus uwy∉Lu w y \notin Luwy∈/L, contradicting the pumping lemma.

### Case 2 — vwxvwxvwx lies entirely inside the bpb^{p}bp block

Analogous: removing vxvxvx (take t=0t=0t=0) yields fewer b's while a's stay ppp. So counts of a and b differ and the result is not in LLL. Contradiction.

### Case 3 — vwxvwxvwx lies entirely inside the cp−1c^{p-1}cp−1 block

Here vxvxvx is some nonempty string of c's. Take t=2t=2t=2. Then the pumped string has p−1+tcp-1 + t\_cp−1+tc​ c's with tc≥1t\_c\ge1tc​≥1, so number of c's becomes ≥p\ge p≥p. But the a/b counts remain ppp, so we get apbpci′a^{p}b^{p}c^{i'}apbpci′ with i′≥pi'\ge pi′≥p, violating the condition i′<n=pi'<n=pi′<n=p. Hence the pumped string ∉L\notin L∈/L. Contradiction.

### Case 4 — vwxvwxvwx crosses the a ⁣− ⁣ba\!-\!ba−b boundary

Then vwxvwxvwx contains some trailing a's and some leading b's. Pumping (either t=0t=0t=0 or t=2t=2t=2) changes the numbers of a's and b's by different amounts (you remove or add characters that belong to a and to b simultaneously, but not the same number of each). Concretely, if you remove vxvxvx you decrease the total of a+b in that local region but the decrease in a count is not guaranteed to equal the decrease in b count; therefore after pumping the number of a's and b's will no longer be equal. Thus the pumped string cannot be of the form anbncia^{n}b^{n}c^{i}anbnci. Contradiction.

(If you prefer a short numeric argument: let vvv contain rar\_ara​ a's and rbr\_brb​ b's with ra+rb≥1r\_a+r\_b\ge1ra​+rb​≥1. Pumping changes a by Δa\Delta\_aΔa​ and b by Δb\Delta\_bΔb​ where Δa≠Δb\Delta\_a\neq\Delta\_bΔa​=Δb​ in general, so equality breaks.)

### Case 5 — vwxvwxvwx crosses the b ⁣− ⁣cb\!-\!cb−c boundary

Same reasoning as Case 4: pumping changes b-count and c-count in a way that destroys the required relationship. Either a and b counts become unequal (if b changed but a didn't), or c becomes ≥n\ge n≥n (if c increased), or some other violation of the form anbncia^{n}b^{n}c^{i}anbnci with i<ni<ni<n. So the pumped string ∉L\notin L∈/L. Contradiction.

In every possible location for vwxvwxvwx, pumping yields a string not in LLL, contradicting the pumping lemma requirement that all pumped strings remain in LLL. Therefore our assumption that LLL is context-free is false.

### Conclusion

L={anbnci∣i<n} is not a context-free language.\boxed{L=\{a^{n}b^{n}c^{i}\mid i<n\}\ \text{is \textbf{not} a context-free language.}}L={anbnci∣i<n} is not a context-free language.​

1. Design a PDA which accepts L = {wCwr|w (a+b)\*}

*L = {aa, bb, abba, aabbaa, abaaba, ......}*

## ****Explanation****

In this type of input string, one input has more than one transition states, hence it is called non-deterministic PDA, and the input string contain any order of 'a' and 'b'. Each input alphabet has more than one possibility to move next state and finally when the stack is empty then the string is accepted by the NPDA. In this NPDA we used some symbols which are given below:

*Γ = {a, b, z}*

Where, Γ = set of all the stack alphabet   
z = stack start symbol   
a = input alphabet   
b = input alphabet

### **Approach**

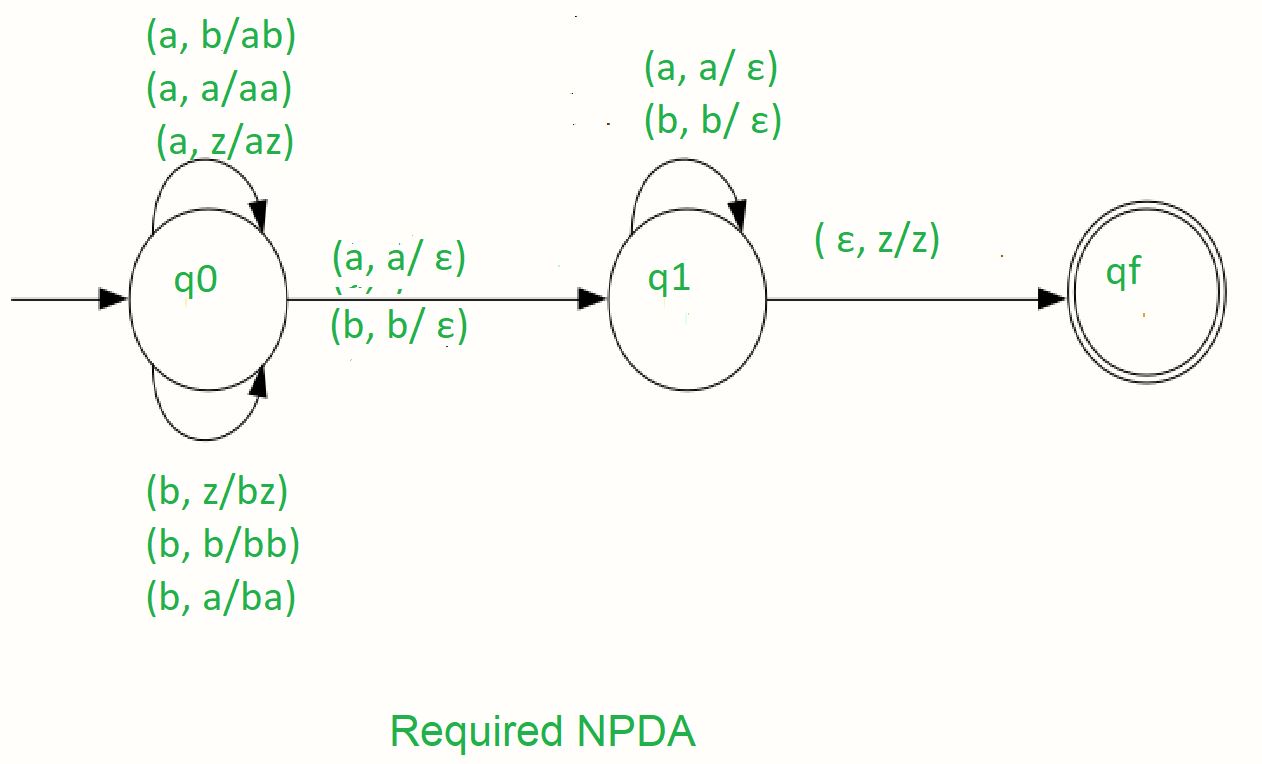
As we want to design an NPDA, thus every times 'a' or 'b' comes then either push into the stack or move into the next state. It is dependent on a string. When we see the input alphabet which is equal to the top of the stack then that time pop operation applies on the stack and move to the next step.

So, in the end, if the stack becomes empty then we can say that the string is accepted by the PDA.

### **Stack Transition Functions**

*δδ(q0, a, z) ⊢⊢ (q0, az)  
δδ(q0, a, a) ⊢⊢ (q0, aa)  
δδ(q0, b, z) ⊢⊢ (q0, bz)  
δδ(q0, b, b) ⊢⊢ (q0, bb)  
δδ(q0, a, b) ⊢⊢ (q0, ab)  
δδ(q0, b, a) ⊢⊢ (q0, ba)  
δδ(q0, a, a) ⊢⊢ (q1, ∈)  
δδ(q0, b, b) ⊢⊢ (q1, ∈)  
δδ(q1, a, a) ⊢⊢ (q1, ∈)  
δδ(q1, b, b) ⊢⊢ (q1, ∈)  
δδ(q1, ∈, z) ⊢⊢ (qf, z)*

Where, q0 = Initial state   
qf = Final state   
∈ = indicates pop operation



So, this is our required non-deterministic PDA for accepting the language L = {wwR |w ∈ (a, b)+ }

### Example

We will take one input string: "abbbba".

1. Scan string from left to right
2. The first input is 'a' and follows the rule:
3. On input 'a' and STACK alphabet Z, push the 'a's into STACK as: (a, Z/aZ) and state will be q0
4. On input 'b' and STACK alphabet 'a', push the 'b' into STACK as: (b, a/ba) and state will be q0
5. On input 'b' and STACK alphabet 'b', push the 'b' into STACK as: (b, b/bb) and state will be q0
6. On input 'b' and STACK alphabet 'b' (state is q1), pop one 'b' from STACK as: (b, b/∈) and state will be q1
7. On input 'b' and STACK alphabet 'b' (state is q1), pop one 'b' from STACK as: (b, b/∈) and state will be q1
8. On input 'a' and STACK alphabet 'a' and state q1, pop one 'a' from STACK as: (a, a/∈) and state will remain q1
9. On input ∈ and STACK alphabet Z, go to the final state (qf) as : (∈, Z/Z)

So, at the end the stack becomes empty then we can say that the string is accepted by the PDA.

**Note:** This DPDA will not accept the empty language.

Let us consider another problem which contains the odd length palindrome

## ****Problem****

**10 Design a deterministic PDA for accepting the language L = { wcwR w ∈ (a, b)+}, i.e. ,**

*{aca, bcb, abcba, abacaba, aacaa, bbcbb, .......}*

**In each string, the substring which is present on the right side of c is the reverse of the substring which is the present left side of c.**

### Explanation

Here we need to maintain string in such a way that, the substring which is present on the left side of c is exactly the reverse substring which is the right side of c. For doing this we used a stack. In string 'a' and 'b' are present any order and 'c' come only one time. When 'c' comes then the pop operation is started into the stack. And when a stack is empty then language is accepted.

*Γ = {a, b, z}*

Where, Γ = set of all the stack alphabet   
z = stack start symbol   
a = input alphabet   
b = input alphabet

### Approach

As we want to design PDA In every time when 'a' or 'b' comes we push into the stack and stay on the same state q0 and when 'c' comes then we move to the next state q1 without pushing 'c' into the stack.

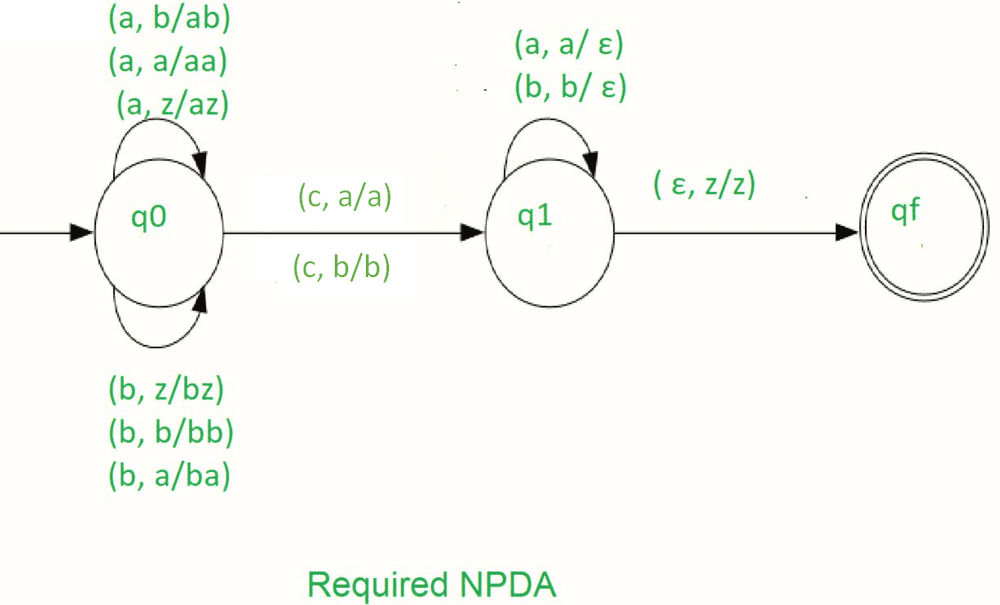
And after when comes an input which is the same as the top of the stack then pop from the stack and stay on the same state. POP operation is performed until the input string is ended.

Finally when the input is ∈, then move to the final state qf. When if the stack will become empty then the language is accepted.   
  
Where, q0 = Initial state   
qf = Final state   
z = stack start symbol   
∈ = indicates pop operation

### Stack Transition Functions

*δδ(q0, a, z) ⊢⊢ (q0, az)  
δδ(q0, a, a) ⊢⊢ (q0, aa)  
δδ(q0, b, z) ⊢⊢ (q0, bz)  
δδ(q0, b, b) ⊢⊢ (q0, bb)  
δδ(q0, a, b) ⊢⊢ (q0, ab)  
δδ(q0, b, a) ⊢⊢ (q0, ba)  
δδ(q0, c, a) ⊢⊢ (q1, a)  
δδ(q0, c, b) ⊢⊢ (q1, b)  
δδ(q1, a, a) ⊢⊢ (q1, ∈)  
δδ(q1, b, b) ⊢⊢ (q1, ∈)  
δδ(q1, ∈, z) ⊢⊢ (qf, z)*

Where, q0 = Initial state   
qf = Final state   
∈ = indicates pop operation 



So, this is our required deterministic PDA for accepting the language,

*L = { wcwR | w ∈ (a, b)+}*

11 Discuss how finite automata is used to represent tokens and Perform lexical analysis with examples.

12. Explain in detail the specifications of tokens.

13. Explain the role of Lex with a suitable example.

14. What is the role of a **lexical analyzer** in a compiler?  
Mention at least four important functions it perform

15. Explain the language processing system with a neat diagram.

16. Design a transition diagram to accept only unsigned integer constants. Modify your diagram to reject inputs with leading zeros (except for 0 itself).