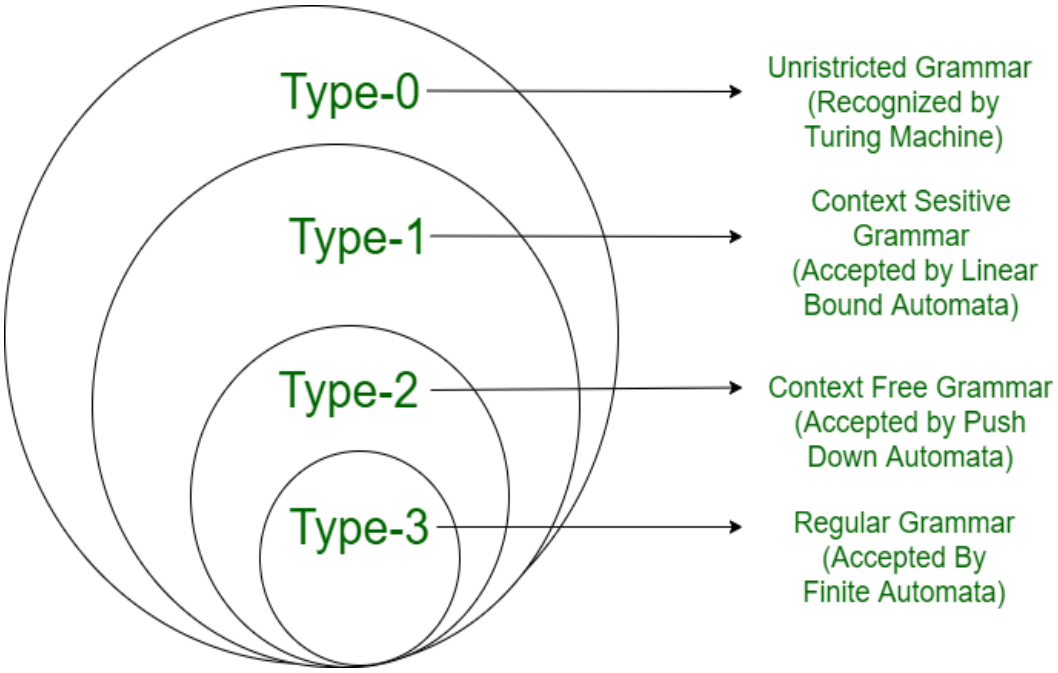
UNIT-2,3 & UNIT 4TH Important Theory Questions

Q: Explain Chomsky Hierarchy in detail.

According to Chomsky hierarchy, grammar is divided into 4 types as follows:

Type 0 is known as unrestricted grammar.
Type 1 is known as context-sensitive grammar.
Type 2 is known as context-free grammar.
Type 3 Regular Grammar.



Type 0: Unrestricted Grammar:

Type-0 grammars include all formal grammar. Type 0 grammar languages are recognized by turing machine. These languages are also known as the Recursively Enumerable languages.

Grammar Production in the form of **a->b** where

“a”is (V + T)* V (V + T)*

V : Variables

T : Terminals.

“b” is (V + T)*.

In type 0 there must be at least one variable on the Left side of production.

For example:

Sab --> ba

A --> S

Variables are S, A, and Terminals a, b.

Type 1: Context-Sensitive Grammar

Type-1 grammars generate context-sensitive languages. The language generated by the grammar is recognized by the Linear Bounded Automata

In Type 1

- First of all Type 1 grammar should be Type 0.
- Grammar Production in the form of

a->b

|a|<=|b|

That is the count of symbol in “a”is less than or equal to “b”

Also ? ? (V + T)+

i.e. ? can not be ?

For Example:

S --> AB

AB --> abc

`B --> b`

Type 2: Context-Free Grammar:

Type-2 grammars generate context-free languages. The language generated by the grammar is recognized by a Pushdown automaton. In Type 2:

- First of all, it should be Type 1.
- The left-hand side of production can have only one variable and there is no restriction on “b”

|a|=1

For example:

S --> AB

A --> a

B --> b

Type 3: Regular Grammar:

Type-3 grammars generate regular languages. These languages are exactly all languages that can be accepted by a finite-state automaton. Type 3 is the most restricted form of grammar.

Type 3 should be in the given form only :

V --> VT / T (left-regular grammar)

(or)

V --> TV /T (right-regular grammar)

For example:

S --> a

The above form is called strictly regular grammar.

There is another form of regular grammar called extended regular grammar. In this form:

V --> VT* / T*. (extended left-regular grammar)

(or)

V --> T*V /T* (extended right-regular grammar)

For example :

S --> ab.

Q: Explain Closure Properties of Regular Grammar?

In an automata theory, there are different closure properties for regular languages. They are as follows −

- Union
- Intersection
- concatenation
- Kleene closure
- Complement

Let see one by one with an example

Union

If L1 and If L2 are two regular languages, their union L1 U L2 will also be regular.

Example

L1 = {a^n | n > O} and L2 = {b^n | n > O}

L3 = L1 U L2 = {an U bn | n > O} is also regular.

Intersection

If L1 and If L2 are two regular languages, their intersection L1 ∩ L2 will also be regular.

Example

L1= {a^m b^n | n > 0 and m > O} and

L2= {a^m b^n U b^n a^m | n > 0 and m > O}

L3 = L1 ∩ L2 = {a^m b^n | n > 0 and m > O} are also regular.

Concatenation

If L1 and If L2 are two regular languages, their concatenation L1.L2 will also be regular.

Example

L1 = {a^n | n > 0} and L2 = {b^n | n > O}

L3 = L1.L2 = {a^m . b^n | m > 0 and n > O} is also regular.

Kleene Closure

If L1 is a regular language, its Kleene closure L1* will also be regular.

Example

L1 = (a U b)

L1* = (a U b)*

Complement

If L(G) is a regular language, its complement L'(G) will also be regular. Complement of a language can be found by subtracting strings which are in L(G) from all possible strings.

Example

L(G) = {a^n | n > 3} L'(G) = {a^n | n <= 3}

Note − Two regular expressions are equivalent, if languages generated by them are the same. For example, (a+b*)* and (a+b)* generate the same language. Every string which is generated by (a+b*)* is also generated by (a+b)* and vice versa.

Reverse Operator

Given language L, L^R is the set of strings whose reversal is in L. Example: L = {0, 01, 100};L^R ={0, 10, 001}. **Proof:** Let E be a regular expression for L. We show how to reverse E, to provide a regular expression E^R for L^R.

Set Difference operator:

If L and M are regular languages, then so is L – M = strings in L but not M. **Proof:** Let A and B be DFA's whose languages are L and M, respectively. Construct C, the product automaton of A and B make the final states of C be the pairs, where A-state is final but B-state is not.

Homomorphism: A homomorphism on an alphabet is a function that gives a string for each symbol in that alphabet. Example: h(0) = ab; h(1) = E

Extend to strings by h(a1…an) =h(a1)…h(an). Example: h(01010) = abab. If L is a regular language, and h is a homomorphism on its alphabet, then h(L)= {h(w) | w is in L} is also a regular language. **Proof:** Let E be a regular expression for L. Apply h to each symbol in E. Language of resulting R, E is h(L).

Inverse Homomorphism : Let h be a homomorphism and L a language whose alphabet is the output language of h. h^-1 (L) = {w | h(w) is in L}.

Q:Explain the decidability problem in Regular Language.

What is a Decidability Problem?

A decidability problem is about figuring out whether we can create a method (an algorithm) that always gives us a correct yes-or-no answer to a specific question, no matter what input we give it. If we can, the problem is "decidable." If we can't, the problem is "undecidable."

Decidability in Regular Languages

For regular languages, there are several important questions we might ask, and we want to know if there are algorithms that can always answer these questions. Here are some common decidability problems for regular languages:

1. **Membership Problem:** Given a string and a regular language, is the string part of the language?
    - Example: Does the string "101" belong to the language described by the regular expression "1*01*"?

    - Decidable: Yes, we can create an algorithm that always gives us the correct answer.
2. **Emptiness Problem:** Given a regular language, is the language empty (does it contain no strings at all)?
    - Example: Is the language described by the regular expression "a*|b*" empty?
    - Decidable: Yes, we can create an algorithm to check this.
3. **Finiteness Problem:** Given a regular language, is the language finite (does it contain only a limited number of strings)?
    - Example: Does the language described by the regular expression "a(b|c)*" contain a limited number of strings?
    - Decidable: Yes, we can determine this using an algorithm.
4. **Equivalence Problem:** Given two regular languages, are they the same (do they contain exactly the same strings)?
    - Example: Do the languages described by the regular expressions "a*" and "(a|aa)*" contain the same strings?
    - Decidable: Yes, there is an algorithm to check if two regular languages are equivalent.

Q:Explain Context-Free Grammar with suitable example.

A Context-Free Grammar is a set of rules that defines the structure of a language. These rules describe how strings in the language can be generated. Each rule in a CFG tells us how a single symbol can be replaced by a combination of other symbols.

A CFG consists of:

1. **Terminals**: The basic symbols from which strings are formed (like letters or digits).
2. **Non-terminals**: Symbols that can be replaced by groups of terminals and non-terminals (like placeholders for patterns).
3. **Production Rules**: The rules that describe how non-terminals can be replaced by combinations of terminals and non-terminals.
4. **Start Symbol**: A special non-terminal symbol from which the generation of strings begins.

Example of a Context-Free Grammar

Let's consider a simple example of a CFG for a language that consists of balanced parentheses. This language includes strings like "", "()", "(())", "(()())", etc.

Here is the CFG for this language:

1. **Terminals**: `(`, `)`
2. **Non-terminals**: `S`
3. **Production Rules**:
    - `S -> SS`
    - `S -> (S)`
    - `S -> ε` (where `ε` represents the empty string)
4. **Start Symbol**: `S`

Explanation of Production Rules

1. `S -> SS`: This rule means that a string formed by `S` can be split into two parts, each of which is also a valid `S`.
    - Example: `(()())` can be split into `(())` and `()`, both of which are balanced.
2. `S -> (S)`: This rule means that a string formed by `S` can be a pair of parentheses with another valid `S` inside.
    - Example: `((()))` can be seen as `(` followed by `(((()))` and then `)`.
3. `S -> ε`: This rule means that `S` can be replaced by an empty string, representing the base case of having no parentheses.

Generating Strings with CFG

Let's see how we can generate a string using these rules.

1. Start with the start symbol: `S`

2. Apply `S -> (S)`: we get `(S)`
3. Apply `S -> SS` to the inner `S`: we get `(SS)`
4. Apply `S -> ε` to both inner `S`: we get `(εε)`
5. Replace `ε` with empty string: we get `()`

So, we have generated the string `()`, which is a valid string in our language of balanced parentheses.

Another Example

Generate the string `(()())`:

1. Start with `S`
2. Apply `S -> (S)`: we get `(S)`
3. Apply `S -> SS` to the inner `S`: we get `(SS)`
4. Apply `S -> (S)` to the first inner `S`: we get `((S)S)`
5. Apply `S -> ε` to the first inner `S`: we get `(()S)`
6. Apply `S -> (S)` to the remaining `S`: we get `(()(S))`
7. Apply `S -> ε` to the inner `S`: we get `(()())`

So, we have generated the string `(()())`.

Q:Discuss basic closure properties of Context free Languages.

Closure Properties of Context-Free Languages

1. **Union**
    - **Property**: If L1 and L2 are CFLs, then L1 ∪ L2 is also a CFL.
    - **Explanation**: We can construct a context-free grammar (CFG) for the union of two CFLs by creating a new start symbol and adding production rules that allow deriving strings from either of the original grammars.

Example: If L1={a^n b^n | n≥0} and L2={a^nb^2n | n≥0}, the union L1 ∪ L2 is also a CFL.

2. **Concatenation**
    - **Property**: If L1 and L2are CFLs, then L1 · L2(the concatenation of L1 and L2) is also a CFL.
    - **Explanation**: We can construct a CFG for the concatenation by creating a new start symbol that derives a string from L1 followed by a string from L2
    - **Example**: If L1={a^n b^n | n≥0} and L2={c^n d^n | n≥0}, the concatenation L1 · L2 is also a CFL.
3. **Kleene Star**
    - **Property**: If L is a CFL, then L^* (the Kleene star of L) is also a CFL.
    - **Explanation**: We can construct a CFG for L^* by creating a new start symbol that can produce either an empty string or a string from L followed by another string from L^*
    - **Example**: If L={a^nb^n | n≥0} then L^* (which includes strings like ϵ, a^nb^n, a^n b^n a^m b^m,… is also a CFL.
4. **Intersection with Regular Language**
    - **Property**: If Lis a CFL and R is a regular language, then L∩R is also a CFL.
    - **Explanation**: We can use a method called "product construction" involving a pushdown automaton for L and a finite automaton for RRR to create a pushdown automaton that recognizes L∩R
    - **Example**: If L={a^nb^n | n≥0} and R=(a | b)*b then L∩RL is also a CFL.

Non-Closure Properties

It's also useful to know some operations for which CFLs are not closed:

1.Intersection

Property: If L1 and L2 are CFLs, L1∩L2 is not necessarily a CFL.

Example: L1={a^nb^nc^m | n,m≥0} and L2={a^mb^nc^n | m,n≥0}. Their intersection L1∩L2={a^nb^nc^n | n≥0} is not a CFL.

2 Complement

Property: If L is a CFL, its complement L is not necessarily a CFL.

Example: If L={a^nb^n | n≥0}, then its complement (strings not of the form a^nb^na^n b^na^nb^n) is not a CFL.

CFLs are closed under operations like union, concatenation, and Kleene star, and intersection with regular languages. However, they are not closed under intersection or complement. Understanding these closure properties helps in knowing what kinds of operations we can perform on CFLs while still remaining within the realm of context-free languages.

Q:Define the yield of a parse tree.

Parse : It means to resolve (a sentence) into its component parts and describe their syntactic roles or simply it is an act of parsing a string or a text.

Tree: A tree may be a widely used abstract data type that simulates a hierarchical tree structure, with a root value and sub-trees of youngsters with a parent node, represented as a group of linked nodes.

Parse Tree:

- Parse tree is the hierarchical representation of terminals or non-terminals.
- These symbols (terminals or non-terminals) represent the derivation of the grammar to yield input strings.
- In parsing, the string springs using the beginning symbol.
- The starting symbol of the grammar must be used as the root of the Parse Tree.
- Leaves of parse tree represent terminals.
- Each interior node represents productions of a grammar.

Rules to Draw a Parse Tree:

1. All leaf nodes need to be terminals.
2. All interior nodes need to be non-terminals.

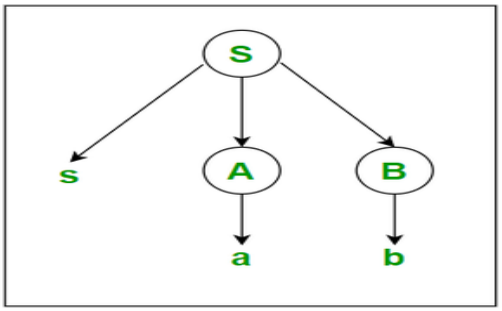
3. In-order traversal gives the original input string.

Let us take an example of Grammar (Production Rules).

S -> sAB

A -> a

B -> b



Q:Explain the ambiguity in CFG.

Ambiguity in context-free grammars (CFGs) occurs when a single string (sentence) can be generated by the grammar in more than one way, leading to different parse trees or interpretations. This means there is no unique way to parse the sentence according to the rules of the grammar.

Q:Discuss the Normal form in CFG. Why do we need it?

Normal forms in Context-Free Grammars (CFG) are standardized ways of writing grammars to simplify parsing and theoretical analysis. The two most commonly discussed normal forms are Chomsky Normal Form (CNF) and Greibach Normal Form (GNF). Here's an overview of each and why they are useful:

Chomsky Normal Form (CNF)

A CFG is in Chomsky Normal Form if all production rules are of the form:

1. $A \rightarrow BC$
2. $A \rightarrow aA$
3. $S \rightarrow \epsilon$

Where A,B, and C are non-terminal symbols, a is a terminal symbol, S is the start symbol, and epsilon ϵ is the empty string.

Why CNF is useful:

1. **Simplified Parsing Algorithms**: Algorithms like the CYK (Cocke-Younger-Kasami) algorithm for parsing are easier to implement and prove correct with CNF because of its restricted rule format.
2. **Proofs and Theoretical Work**: Many theoretical properties of CFGs and their languages are easier to demonstrate when the grammar is in CNF. This includes proofs of decidability and complexity.
3. **Consistency and Standardization:** Having a standard form makes it easier to compare different grammars and their properties, facilitating better understanding and communication of formal language theory.

Greibach Normal Form (GNF)

A CFG is in Greibach Normal Form if all production rules are of the form:

1. $A \rightarrow aA1A2$

   $A \rightarrow a$

   $S \rightarrow \epsilon$

Where A1,A2… is a non-terminal, a is a terminal, and Epsilon is a (possibly empty) string of non-terminals.

Why GNF is useful:

1. **Top-Down Parsing:** GNF is particularly useful for top-down parsing techniques, such as recursive descent parsing, because it ensures that each production expands to a terminal followed by non-terminals.
2. **Predictability**: Since each production starts with a terminal symbol, it is easier to predict and manage parsing steps, aiding in the construction of efficient parsers.
3. **Theoretical Analysis**: Similar to CNF, GNF simplifies theoretical analyses and proofs, especially in demonstrating properties related to deterministic parsing and parsing strategies.

Why We Need Normal Forms

1. **Simplification(Easier Understanding):** Normal forms reduce the complexity of the grammar rules, making it easier to work with the grammars both manually and algorithmically.

2. **Algorithm Design:** Many parsing and language-processing algorithms require or perform better with grammars in a specific normal form.
3. **Theoretical Understanding:** Normal forms help in the formal study and classification of languages. They make it easier to understand the structure and behavior of languages generated by CFGs.
4. **Interoperability**: Converting grammars to a normal form can make it easier to translate between different types of grammars and formal language tools, facilitating interoperability between different systems and frameworks.
5. **Proof of Equivalence:** It helps in proving the equivalence between different CFGs, showing that they generate the same language even if they look different.
6. **Error Detection**
7. **Optimization**
8. **Eliminating Redundancies:** Remove unnecessary rules & symbols do not contribute to generate language.

Q:Differentiate DPDA and NPDA.

Deterministic Pushdown Automata (DPDA) and Non-deterministic Pushdown Automata (NPDA) are two types of Pushdown Automata (PDA) used to recognize context-free languages (CFLs). While they share many characteristics, their key difference lies in the nature of their transitions and the types of languages they can recognize. Here's a detailed comparison:

Deterministic Pushdown Automata (DPDA)

Definition

A DPDA is a PDA where, for any given input symbol and stack symbol, there is at most one possible transition. This means the automaton's behavior is entirely predictable and does not involve any choice or guesswork.

Formal Definition

A DPDA can be defined as a 7-tuple (Q,Σ,Γ,δ,q0,Z0,F) q_0, Z_0, F)(Q,Σ,Γ,δ,q0,Z0,F) with the constraint that for each (q,a,X) in Q×(Σ∪{ϵ}) there is at most one element in the set δ(q,a,X)

Characteristics

- **Uniqueness:** For each state, input symbol, and stack symbol, there is at most one possible move.
- **Language Recognition:** DPDAs recognize a proper subset of context-free languages called deterministic context-free languages (DCFLs).

- **Acceptance:** A DPDA can accept input either by reaching an accept state or by emptying its stack.

Example

Consider a DPDA that accepts the language L={a^nb^n | n≥0}

- States: Q={q0,q1,q2}
- Input Alphabet: Σ={a,b}
- Stack Alphabet: Γ={Z,A}
- Start State: q0q_0q0
- Initial Stack Symbol: Z
- Accept States: F={q2}

Transition Function

δ(q0,a,Z)=(q0,AZ)

δ(q0,a,A)=(q0,AA)

δ(q0,b,A)=(q1,ϵ)

δ(q1,b,A)=(q1,ϵ)

δ(q1,ϵ,Z)=(q2,Z)

Non-deterministic Pushdown Automata (NPDA)

Definition

An NPDA is a PDA where, for a given input symbol and stack symbol, there can be multiple possible transitions. This means the automaton can explore multiple computation paths simultaneously.

Formal Definition

An NPDA can be defined as a 7-tuple (Q,Σ,Γ,δ,q0,Z0,F)F) where the transition function δ\deltaδ maps Q×(Σ∪{ϵ})×ΓQ to subsets of Q×Γ∗

Characteristics

- **Non-determinism:** For each state, input symbol, and stack symbol, there can be multiple possible moves.
- **Language Recognition:** NPDAs recognize all context-free languages (CFLs).
- **Acceptance:** An NPDA can accept input either by reaching an accept state or by emptying its stack.

Example

Consider an NPDA that accepts the language L={a^nb^n | n≥0}

- States: Q={q0,q1,q2}
- Input Alphabet: Σ={a,b}
- Stack Alphabet: Γ={Z,A}
- Start State: q0
- Initial Stack Symbol: Z
- Accept States: F={q2}

Transition Function

1. δ(q0,a,Z)={(q0,AZ)}
2. δ(q0,a,A)={(q0,AA)}
3. δ(q0,ϵ,Z)={(q1,Z)}
4. δ(q0,b,A)={(q1,ϵ)}
5. δ(q1,b,A)={(q1,ϵ)}
6. δ(q1,ϵ,Z)={(q2,Z)}

Key Differences

1. **Determinism:**
    - DPDA: Has a unique transition for each input symbol and stack symbol combination.
    - NPDA: Can have multiple possible transitions for each input symbol and stack symbol combination.
2. **Language Recognition:**
    - DPDA: Recognizes deterministic context-free languages (DCFLs), a subset of CFLs.
    - NPDA: Recognizes all context-free languages (CFLs).
3. **Complexity:**
    - DPDA: Simpler to implement due to determinism.
    - NPDA: More powerful but potentially more complex due to non-determinism.
4. **Acceptance Criteria:**
    - Both DPDAs and NPDAs can accept input by reaching an accept state or by emptying their stack, though the methods can be different due to the nature of their transitions.

Q:Define the role of stack used in PDA.

In a Pushdown Automaton (PDA), a stack plays a crucial role in maintaining the memory or "state" of the computation. Here's a breakdown of its role:

1. **Memory Storage:** The stack serves as a memory storage mechanism for the PDA. It allows the PDA to store symbols temporarily as it processes input symbols. This enables the PDA to remember previous states or configurations during its computation.
2. **Last-In-First-Out (LIFO) Structure:** The stack follows the Last-In-First-Out principle, meaning the most recently added symbol is the first one to be removed. This property is essential in simulating the behavior of a PDA, where the most recent decisions or transitions need to be undone in a reverse order.
3. **Push and Pop Operations:** The PDA can perform two fundamental operations on the stack:
    - Push: When the PDA reads an input symbol and decides to transition to a new state while potentially adding symbols to the stack. This operation adds a symbol to the top of the stack.
    - Pop: When the PDA needs to backtrack or transition back to a previous state, it can remove symbols from the top of the stack. This operation removes the top symbol from the stack.
4. **Determining Acceptance:** In PDA-based languages, the acceptance of a string often depends on the final state of the PDA and the contents of the stack after processing the entire input. The stack allows the PDA to make decisions based not only on the current input symbol but also on the history of symbols processed so far.

Q:Explain instantaneous Description with example.

An instantaneous description (ID) in the context of a Pushdown Automaton (PDA) provides a snapshot of the current configuration of the PDA during its computation. It consists of the current state of the PDA, the remaining input to be processed, and the contents of the stack.

Here's a breakdown of the components of an instantaneous description:

1. Current State: This indicates the state of the PDA at the current step of computation.
2. Remaining Input: This shows the portion of the input string that has not yet been processed.
3. Stack Contents: This displays the symbols currently stored in the stack, typically from top to bottom.

Together, these components provide a comprehensive view of the PDA's progress and the context in which it operates at any given moment during its computation.

Let's illustrate this with an example:

Consider a PDA that recognizes strings of the form a^nb^n where n is a non-negative integer. The PDA has the following transition rules:

- q0 is the initial state.
- q1 is the accepting state.

- The PDA reads an aaa from the input and pushes it onto the stack.
- The PDA pops an aaa from the stack for each bit read from the input.
- If the stack becomes empty while there is still input remaining, it means the number of b's exceeds the number of a's, and the input is rejected.

Now, let's observe the instantaneous descriptions for the input string aabbb

0. q0,aabbb,ϵ

Initially, the PDA is in state q0d the input is aabbb. The stack is empty.

1. q0,abbb,a

After reading aaa, the PDA pushes it onto the stack.

2. q0,bbb,aa

After reading another aaa, the PDA pushes it onto the stack.

3. q0,bb,aaa

After reading b, the PDA pops an a from the stack.

4. q0,b,aa

After reading another b, the PDA pops another a from the stack.

5. q0,ϵ,a

After reading the last b, the PDA pops the last afrom the stack, and the input is empty.

6. q1,ϵ,ϵ

Since the input is empty, and the stack is also empty, and the PDA is in the accepting state q1q_1q1, the input string aabbb is accepted.

Each of these instantaneous descriptions represents a specific step in the computation of the PDA, providing insight into how the PDA processes the input string and manipulates its stack.