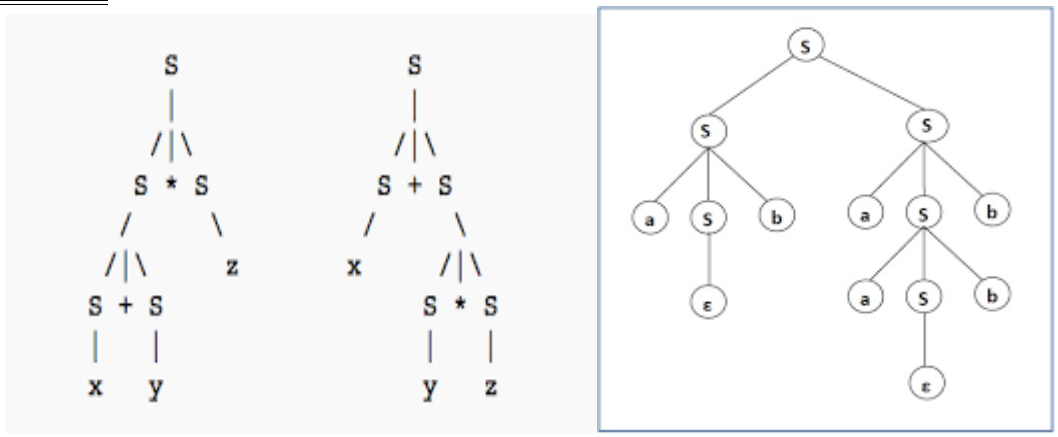
Unit-III: Context Free Grammar, Push Down Automata and Turing Machines 12hrs
Context Free Grammar: (ACD)

Context Free Grammar:



A context-free grammar (CFG) is a formal grammar consisting of a set of production rules that describe the syntax of a language. It is called "context-free" because the production rules are applied without considering the context or surrounding symbols.

A CFG consists of the following components:

1. Terminal Symbols: These are the basic symbols or tokens that appear in the language. They cannot be further divided. For example, in a programming language, terminal symbols can represent keywords, operators, and literals.
2. Non-Terminal Symbols: These symbols represent syntactic categories or abstract entities. Non-terminal symbols can be further expanded or replaced by a sequence of terminals and non-terminals. They are often denoted by uppercase letters.
3. Start Symbol: It represents the initial non-terminal from which the derivation of the strings begins.
4. Production Rules: These rules specify how the non-terminal symbols can be replaced or expanded into a sequence of terminals and non-terminals. Each production rule has the form A -> α, where A is a non-terminal symbol and α is a string of terminals and non-terminals.
5. Derivation: It is the process of applying production rules to generate valid strings in the language. Starting with the start symbol, the production rules are repeatedly applied until a string consisting only of terminal symbols is obtained.
6. Language: The set of all strings that can be generated by the CFG is called the language of the grammar.

For example, let's consider a simple CFG for a mathematical expression language:

a. Terminal Symbols: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /, (,)}
b. Non-Terminal Symbols: {expression, term, factor}
c. Start Symbol: expression
d. Production Rules:

- expression -> expression + term
- expression -> expression - term
- expression -> term
- term -> term * factor
- term -> term / factor
- term -> factor
- factor -> (expression)
- factor -> digit
- digit -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Using this CFG, we can generate valid mathematical expressions by applying the production rules. For example, starting with the expression non-terminal, we can derive the following expression:

expression -> expression + term -> term + term -> factor + term -> digit + term -> 3 + term -> 3 + factor -> 3 + (expression) -> 3 + (term) -> 3 + factor -> 3 + digit -> 3 + 8

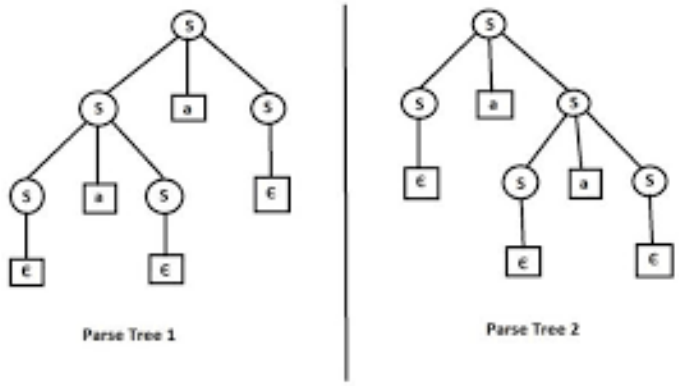
The resulting string "3 + 8" is a valid expression in the language described by the CFG.

Ambiguous Grammars

An ambiguous grammar is a type of context-free grammar (CFG) in which there exist multiple derivation trees or parse trees for a single string in the language. In other words, there can be more than one way to derive the same string using the production rules of the grammar.

The ambiguity arises when the production rules of a grammar allow for different interpretations or parsing decisions. This can lead to multiple possible meanings or structures for a given input string. Ambiguous grammars can make parsing and

understanding the language more challenging, as the same string can be parsed in different ways, potentially resulting in different interpretations or behaviors.



Ambiguity in grammars can occur due to various reasons, including:
1. Ambiguous Production Rules: Some production rules may have multiple alternatives that can be applied to the same non-terminal symbol. This ambiguity can result in different parse trees for a string.
2. Left Recursion: Left-recursive production rules can introduce ambiguity. A left-recursive rule is one where the non-terminal symbol appears as the leftmost symbol on the right-hand side of the rule. Left recursion can lead to infinite loops during parsing and ambiguity in the resulting parse trees.
3. Operator Precedence: If the grammar does not specify clear rules for operator precedence and associativity, it can result in ambiguity when parsing expressions. For example, in an arithmetic expression grammar, the order of operations may not be well-defined, leading to multiple valid interpretations.
4. Ambiguous Expressions: Certain language constructs or expressions may inherently have multiple valid interpretations. For example, ambiguous grammars can arise in natural languages due to homonyms or phrases with multiple meanings.

It is important to note that not all grammars are ambiguous. Many grammars are designed to be unambiguous, meaning that each input string has a unique and well-defined parse tree.

Resolving ambiguity in grammars is a crucial task in language design and parsing. Techniques such as left-factoring, left-recursion elimination, and precedence rules can be employed to transform ambiguous grammars into unambiguous ones. Additionally, specifying explicit rules for operator precedence and associativity can help disambiguate expressions.
By eliminating or resolving ambiguity in grammars, we can ensure that the parsing process produces a unique and consistent interpretation of the language's syntax.

Simplification of Context Free Grammars :

Simplification of context-free grammars involves reducing their complexity and size without changing the language they generate. This simplification can improve readability, reduce parsing complexity, and make the grammar more manageable for analysis or implementation.

Here are some common techniques for simplifying context-free grammars:

1) Removal of Useless Symbols: Useless symbols are those that cannot be reached from the start symbol or cannot derive any terminal symbols. These symbols do not contribute to the language generated by the grammar and can be safely removed. This includes eliminating non-terminals and productions that are unreachable or non-productive.
2) Elimination of ε-Productions: An ε-production is a production rule that derives the empty string (ε). These productions can introduce complexity in parsing and lead to ambiguity. They can be removed by either replacing them with other productions or by introducing new non-terminals to represent optional parts of the grammar.
3) Removal of Unit Productions: Unit productions are production rules where a non-terminal directly derives another non-terminal. These productions can be eliminated by merging the non-terminals and their productions. This simplifies the grammar by reducing unnecessary layers of indirection.
4) Factoring and Combining Productions: Factoring is the process of identifying common prefixes or suffixes in productions and grouping them together. It reduces redundancy and improves readability. Combining productions involves merging similar or related productions into a single rule, reducing the number of rules in the grammar.
5) Left-Recursion Elimination: Left-recursive production rules can cause issues in parsing and lead to infinite loops. They can be eliminated by transforming the grammar to be right-recursive or by introducing additional non-terminals to handle recursion indirectly.
6) Simplifying Precedence and Associativity: If the grammar includes operators or expressions with complex precedence and associativity rules, simplifying them can make the grammar more manageable. This involves explicitly defining precedence and associativity rules to remove ambiguity and ensure a unique parse tree for each expression.

7) Normalization: Normalizing a grammar involves standardizing the grammar's representation by applying a set of transformation rules. This can make the grammar easier to analyze and compare with other grammars. Common normalization techniques include removing useless symbols, eliminating ε-productions and unit productions, and resolving left-recursion.

Normal Forms- Chomsky Normal Form

Chomsky Normal Form (CNF) is a specific form that a context-free grammar (CFG) can be transformed into. In CNF, every production rule in the grammar has one of two forms:
1. A -> BC, where A, B, and C are non-terminal symbols.
2. A -> a, where A is a non-terminal symbol and a is a terminal symbol.

CNF has the following properties:
1. No ε-Productions: CNF does not allow ε-productions, except for the start symbol if it is allowed to produce ε. An ε-production is a production rule that derives the empty string (ε).
2. No Unit Productions: CNF does not allow unit productions, which are production rules where a non-terminal symbol directly derives another non-terminal symbol.

The process of transforming a context-free grammar into Chomsky Normal Form involves several steps:
1. Eliminating ε-Productions: Remove or replace any ε-productions in the grammar. An ε-production can be removed by deleting the rule or by introducing new productions to handle the case where the non-terminal symbol derives ε.
2. Eliminating Unit Productions: Merge non-terminals that are connected by unit productions. If a non-terminal A directly derives another non-terminal B, replace all occurrences of B with A in the grammar.
3. Introducing New Non-Terminals: For each terminal symbol that appears in the right-hand side of a production, introduce a new non-terminal symbol and replace the terminal symbol with the new non-terminal.
4. Breaking Down Long Productions: If a production rule has more than two non-terminal symbols on the right-hand side, introduce new non-terminals to break down the rule into smaller productions. Each new non-terminal represents a part of the original production.
5. Removing Unreachable Symbols: Remove any non-terminals and productions that cannot be reached from the start symbol.

The resulting grammar in Chomsky Normal Form has the advantage of having a simple and regular structure. It is easier to analyze and process, and it allows for efficient parsing algorithms such as the CYK algorithm.

Chomsky Normal Form provides a standardized representation for context-free grammars and facilitates the study of their properties and behavior. However, it's important to note that transforming a grammar into CNF can increase the number of productions and potentially affect the readability of the grammar.

Normal Forms-Griebach Normal Form

Greibach Normal Form (GNF) is a specific form that a context-free grammar (CFG) can be transformed into. In GNF, every production rule in the grammar has the form:
A -> aα
where A is a non-terminal symbol, a is a terminal symbol, and α is a (possibly empty) string of non-terminal symbols.

GNF has the following properties:
1. Allows ε-Productions: GNF allows ε-productions, which are production rules that derive the empty string (ε). However, ε-productions are only allowed for non-terminals other than the start symbol.
2. Eliminates Unit Productions: GNF does not allow unit productions, which are production rules where a non-terminal directly derives another non-terminal.

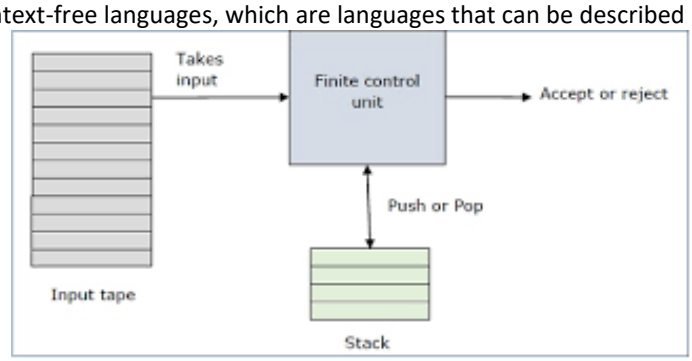
The process of transforming a context-free grammar into Greibach Normal Form involves several steps:
1. Eliminating ε-Productions: Remove or replace any ε-productions in the grammar. Similar to the transformation to Chomsky Normal Form, ε-productions can be removed by deleting the rule or by introducing new productions to handle the case where the non-terminal symbol derives ε.
2. Eliminating Unit Productions: Merge non-terminals that are connected by unit productions. If a non-terminal A directly derives another non-terminal B, replace all occurrences of B with A in the grammar.
3. Introducing New Non-Terminals: For each terminal symbol that appears in the right-hand side of a production, introduce a new non-terminal symbol and replace the terminal symbol with the new non-terminal.
4. Eliminating Left Recursion: If the grammar contains left-recursive rules, eliminate the left recursion by introducing new non-terminals and rewriting the rules.
5. Rearranging Productions: Finally, rearrange the productions to have the desired form, where the non-terminal appears on the left side, followed by a terminal symbol and then a (possibly empty) string of non-terminals.

The resulting grammar in Greibach Normal Form is more restricted than Chomsky Normal Form, but it still allows for efficient top-down parsing algorithms such as recursive descent parsing. Greibach Normal Form is named after the mathematician Sheila Greibach.

Greibach Normal Form provides a standardized representation for context-free grammars and facilitates the study of their properties and behavior. However, similar to Chomsky Normal Form, transforming a grammar into GNF can increase the number of productions and potentially affect the readability of the grammar.

Push Down Automata (PDA):

A Pushdown Automaton (PDA) is a type of automaton that extends the capabilities of a finite-state machine (FSM) by adding a stack. It is used to recognize context-free languages, which are languages that can be described by context-free grammars.



A PDA consists of the following components:

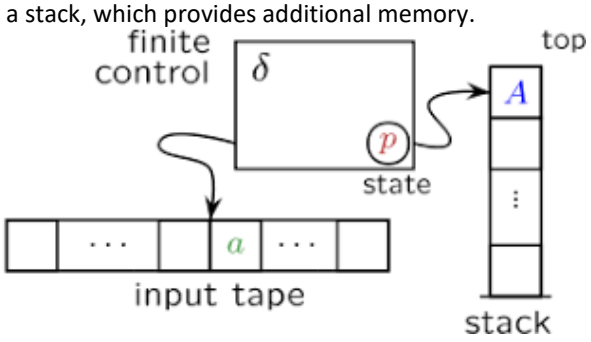
1. Finite Set of States: Similar to an FSM, a PDA has a finite set of states. These states represent different configurations of the PDA during its operation.
2. Input Alphabet: The input alphabet is a finite set of symbols that the PDA can read as input. These symbols can be from any finite set, including both terminal and non-terminal symbols.
3. Stack Alphabet: The stack alphabet is a finite set of symbols that can be pushed onto and popped from the stack. These symbols are typically taken from the same set as the input alphabet, but they can also be different.
4. Transition Function: The transition function defines how the PDA can change its state, read an input symbol, and manipulate the stack. It takes the current state, the input symbol, and the symbol on the top of the stack as input and determines the next state, the symbol to be pushed onto the stack, and whether to pop a symbol from the stack.
5. Start State: The start state is the initial state of the PDA. It represents the beginning of the input processing.
6. Accepting States: The accepting states are the states in which the PDA halts and accepts the input string. These states indicate that the input string is recognized by the PDA.

During the operation of a PDA, it reads symbols from the input, performs state transitions based on the current state and the input symbol, and manipulates the stack by pushing and popping symbols. The stack allows the PDA to keep track of non-terminal symbols encountered during the input processing.

A PDA accepts an input string if, after processing the entire input, it reaches an accepting state and the stack is empty. This indicates that the input string is recognized by the PDA according to the specified language.

Model

A Pushdown Automaton (PDA) is a theoretical computational model that consists of a finite set of states, an input alphabet, a stack alphabet, a start state, a stack, a set of accepting states, and a transition function. It extends the capabilities of a finite-state machine (FSM) by incorporating a stack, which provides additional memory.



- The stack in a PDA is a last-in, first-out (LIFO) data structure that can store symbols from the stack alphabet. The stack operations include push (add a symbol to the top of the stack) and pop (remove the top symbol from the stack). The PDA can also read symbols from the input alphabet and change its state based on the current state, the input symbol, and the top symbol of the stack.
- The transition function of a PDA determines the state transitions and stack operations based on the current state, the input symbol, and the top symbol of the stack. It specifies the next state, the symbol to be pushed onto the stack (or whether to leave the stack unchanged), and the symbol to be popped from the stack (or whether to leave the stack unchanged).

- During the operation of a PDA, it reads symbols from the input, performs state transitions, and manipulates the stack. The stack allows the PDA to remember and process information about the input symbols encountered so far. The PDA can accept an input string if, after processing the entire input, it reaches an accepting state and the stack is empty.
- Pushdown Automata are used to recognize context-free languages, which are languages that can be generated by context-free grammars. They provide a more powerful computational model than FSMs and can handle nested structures and non-deterministic behaviors. PDAs are fundamental in the theory of computation, formal languages, parsing algorithms, and compiler design.

Desgin of PDA

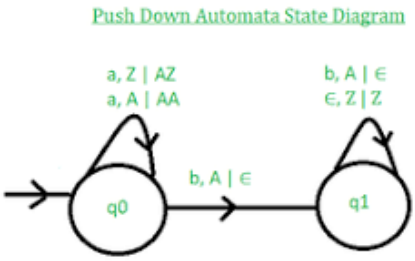
Designing a Pushdown Automaton (PDA) involves specifying its components, including the finite set of states, input and stack alphabets, start state, accepting states, and the transition function. Here's a step-by-step guide to designing a PDA:

1. Define the Input Alphabet: Determine the set of symbols that the PDA will read as input. These symbols can be from any finite set and represent the valid input for the language being recognized.
2. Define the Stack Alphabet: Determine the set of symbols that the PDA will use to populate its stack. These symbols can be from the same set as the input alphabet or a different set.
3. Determine the Set of States: Identify the finite set of states that the PDA will have. These states represent different configurations or conditions of the PDA during its operation. At minimum, there should be a start state and one or more accepting states.
4. Determine the Start State: Specify the initial state of the PDA from which the input processing begins. This state represents the starting point of the computation.
5. Determine the Accepting States: Identify the states in which the PDA will halt and accept the input string. These states indicate that the input string is recognized by the PDA.
6. Define the Transition Function: Specify the transition function, which defines how the PDA transitions from one state to another based on the current state, the input symbol being read, and the symbol on the top of the stack. The transition function determines the next state, the symbol to be pushed onto the stack, and whether to pop a symbol from the stack.
7. Determine the Stack Operations: Decide on the stack operations that will be performed by the PDA during state transitions. These operations include pushing symbols onto the stack, popping symbols from the stack, or leaving the stack unchanged.
8. Validate the Design: Ensure that the PDA design is consistent with the intended language. Check that the transition function covers all possible scenarios and that the PDA will correctly recognize the desired language.

Once the design is complete, the PDA can be implemented using various programming languages or simulation tools. The design can be represented through a state transition diagram or a state transition table, which visually or systematically display the state transitions and stack operations of the PDA

Deterministic PDA

A Deterministic Pushdown Automaton (DPDA) is a type of Pushdown Automaton (PDA) in which for each configuration, there is at most one possible transition. In other words, given the current state, input symbol, and top symbol of the stack, a DPDA always has a unique next configuration.



The key difference between a DPDA and a non-deterministic PDA is that in a DPDA, the transition function is deterministic, whereas in a non-deterministic PDA, there can be multiple possible transitions for a given configuration.
The deterministic property of a DPDA simplifies its operation and analysis compared to a non-deterministic PDA. It allows for more straightforward implementation and guarantees that the recognition process will always follow a unique path.

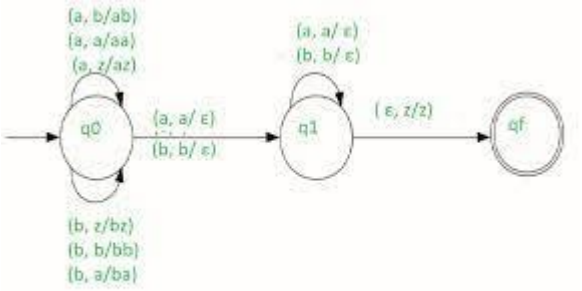
To design a deterministic PDA, the following considerations need to be taken into account:
1. Deterministic Transition Function: The transition function of a DPDA should be defined in such a way that for every possible combination of current state, input symbol, and stack top symbol, there is at most one valid transition.
2. Deterministic Stack Operations: The stack operations (push and pop) in a DPDA should be deterministic. That means, for a given configuration, there should be a unique stack operation associated with the transition.
3. Deterministic Accepting States: In a DPDA, the set of accepting states should be well-defined and deterministic. That is, a unique accepting state should be specified to indicate successful recognition of the input string.

Designing a DPDA involves carefully defining the transition function, stack operations, and accepting states to ensure deterministic behavior. By maintaining determinism, the DPDA guarantees that it will always follow a unique path during the input processing, simplifying the analysis and implementation of the PDA.

Non-deterministic PDA

A Non-deterministic Pushdown Automaton (NPDA) is a type of Pushdown Automaton (PDA) that allows for multiple possible transitions for a given configuration. In other words, given the current state, input symbol, and top symbol of the stack, an NPDA can have multiple choices for the next configuration.
The key characteristic of an NPDA is that its transition function is non-deterministic. It means that for a particular configuration, there can be more than one valid transition, and the NPDA can choose any of these transitions to continue its computation.

The non-deterministic property of an NPDA allows for greater flexibility and expressive power compared to a deterministic PDA. It can explore different paths during the recognition process, which can be beneficial for certain language recognition problems.

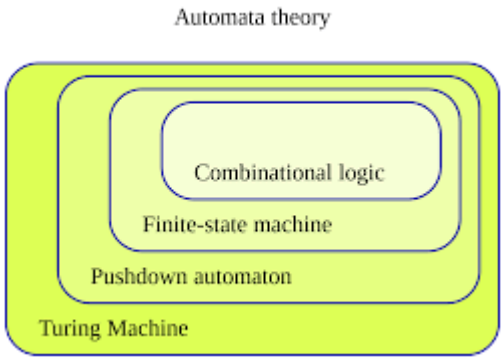


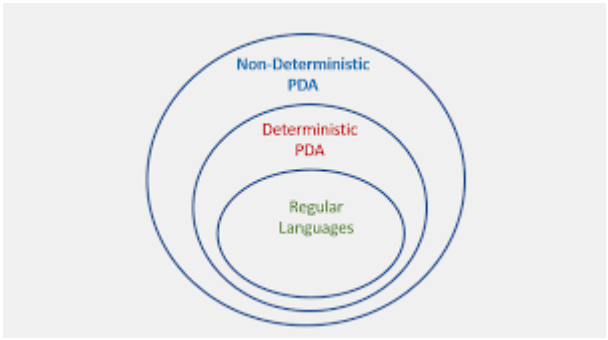
Required NPDA

To design a non-deterministic PDA, the following considerations need to be taken into account:

1. Non-deterministic Transition Function: The transition function of an NPDA allows for multiple possible transitions for a given combination of current state, input symbol, and stack top symbol. These transitions can be defined explicitly, indicating all the possible choices, or they can be defined using non-deterministic rules.
2. Non-deterministic Stack Operations: The stack operations (push and pop) in an NPDA can also be non-deterministic. For a given configuration, there can be multiple stack operations associated with different transitions.
3. Non-deterministic Accepting States: In an NPDA, the set of accepting states can be non-deterministic. That is, there can be multiple accepting states indicating successful recognition of the input string.

Designing an NPDA involves defining the transition function, stack operations, and accepting states to allow for non-deterministic behavior. The non-determinism in the NPDA enables it to explore different computation paths and potentially recognize a broader class of languages compared to a deterministic PDA.





Equivalence of PDA and Context Free Grammars :

Pushdown Automata (PDAs) and Context-Free Grammars (CFGs) are two different formal models used in the study of formal languages and computation. While PDAs and CFGs are related, they are not exactly equivalent in terms of expressive power. However, there exists a strong connection between PDAs and CFGs through language recognition.

The key relationship between PDAs and CFGs is as follows:

1. Every Context-Free Grammar can be recognized by a PDA:
   - For any language generated by a CFG, there exists a PDA that can recognize that language.
   - This means that every language generated by a CFG is also a language recognized by a PDA.
2. Every PDA can be simulated by a Context-Free Grammar:
   - For any language recognized by a PDA, there exists a CFG that can generate that language.
   - This means that every language recognized by a PDA is also a language generated by a CFG.

These connections establish that the class of languages recognized by PDAs (called the class of context-free languages) is equivalent to the class of languages generated by CFGs. Therefore, the expressive power of PDAs and CFGs is equivalent in terms of language recognition.

However, it's important to note that PDAs and CFGs differ in their operational aspects and represent different aspects of computation. PDAs are concerned with the recognition of languages, while CFGs focus on the generation of languages. PDAs use a stack-based memory structure, whereas CFGs use production rules to generate strings.

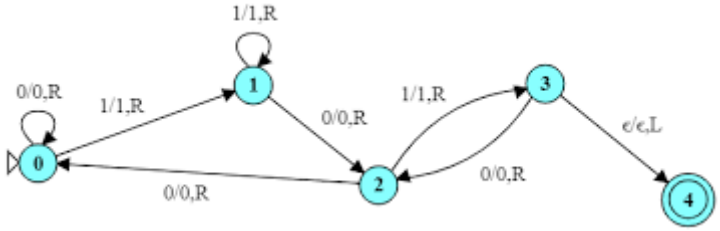
In summary, while PDAs and CFGs are not exactly equivalent in terms of their operational characteristics, they are equivalent in terms of the languages they can recognize or generate. The relationship between PDAs and CFGs provides a fundamental connection between the syntactic and computational aspects of formal languages and forms the basis for many important results and algorithms in the field of formal languages and automata theory.

Turing Machine (TM)

A Turing Machine (TM) is a theoretical computing device introduced by Alan Turing in the 1930s. It is a mathematical model that simulates a general-purpose computer and forms the foundation of the theory of computation.

A Turing Machine consists of the following components:

1. Tape: The tape is an infinite length strip divided into discrete cells. Each cell can hold a symbol from a finite alphabet, including both input symbols and special blank symbols.
2. Head: The head is a device that moves along the tape, reading and writing symbols on the cells. It can also change its internal state.
3. State Register: The state register holds the current state of the Turing Machine.
4. Transition Function: The transition function defines the behavior of the Turing Machine. It specifies how the Turing Machine can change its state, move the head, read and write symbols on the tape, and determine the next action based on the current state and the symbol under the head.
5. Initial State: The initial state is the starting state of the Turing Machine when it begins its computation.
6. Accepting and Rejecting States: The Turing Machine can have one or more accepting states, indicating that it halts and accepts the input, or it can have a single rejecting state, indicating that it halts and rejects the input.



The operation of a Turing Machine involves the following steps:

1. The Turing Machine starts in the initial state with the input provided on the tape.
2. The head reads the symbol under it and consults the transition function to determine the next action.
3. Based on the current state and the symbol under the head, the Turing Machine may change its state, write a new symbol on the tape, move the head left or right, or perform other actions specified by the transition function.
4. The process continues until the Turing Machine reaches an accepting or rejecting state. If it reaches an accepting state, it halts and accepts the input; if it reaches a rejecting state, it halts and rejects the input.

Turing Machines are known for their ability to simulate any algorithmic process and solve problems that are computationally solvable. They are used to define the concept of computability, study the limits of mechanical computation, and analyze the complexity of algorithms. The Church-Turing Thesis states that any algorithmic computation can be performed by a Turing Machine or its equivalent. This thesis is fundamental to the theory of computation and has had a profound impact on computer science and mathematics.

Model, Design of Turing Machine

A Turing Machine (TM) is a theoretical computing device that operates on an infinite tape divided into discrete cells, where each cell can hold a symbol from a finite alphabet. Designing a Turing Machine involves specifying its components and defining the behavior of the machine through a transition function. Here's a step-by-step guide to designing a Turing Machine:

1. Define the Tape Alphabet: Determine the finite set of symbols that can appear on the tape. This includes input symbols, special blank symbols, and any additional symbols required for the specific problem.
2. Define the State Set: Identify the finite set of states that the Turing Machine will have. These states represent different configurations or conditions of the Turing Machine during its computation.
3. Determine the Initial State: Specify the initial state of the Turing Machine when it begins its computation. This state represents the starting point of the machine's operation.
4. Determine the Accepting and Rejecting States: Decide on the states that indicate the acceptance or rejection of the input. The Turing Machine halts and accepts the input if it reaches an accepting state, or it halts and rejects the input if it reaches a rejecting state.
5. Design the Transition Function: The transition function defines how the Turing Machine changes its state, moves the head, reads and writes symbols on the tape, and determines the next action based on the current state and the symbol under the head.
   - For each state and symbol combination, specify the next state, the symbol to be written on the tape, the direction in which the head should move (left or right), and the next action of the Turing Machine.
   - The transition function should cover all possible combinations of states and symbols that the Turing Machine may encounter during its computation.
6. Validate the Design: Ensure that the Turing Machine design is consistent with the problem or language being addressed. Check that the transition function is well-defined and that the Turing Machine behaves correctly for different inputs.

Once the design is complete, the Turing Machine can be implemented using various programming languages or simulation tools. The design can be represented through a state transition diagram or a state transition table, which visually or systematically display the state transitions and actions of the Turing Machine.

Deterministic TM

A Deterministic Turing Machine (DTM) is a type of Turing Machine (TM) in which for each state and symbol combination, there is at most one valid transition. In other words, given the current state and the symbol under the head, a DTM always has a unique next configuration.

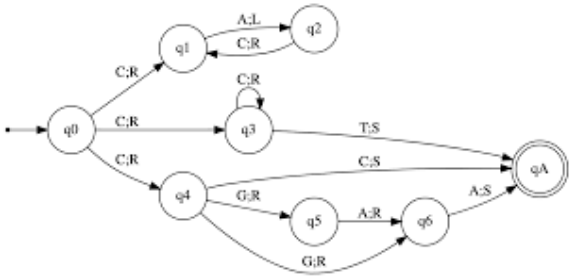
The key difference between a DTM and a non-deterministic Turing Machine (NTM) is that in a DTM, the transition function is deterministic, whereas in an NTM, there can be multiple possible transitions for a given configuration.
The deterministic property of a DTM simplifies its operation and analysis compared to an NTM. It allows for straightforward implementation and guarantees that the computation will always follow a unique path.

To design a Deterministic Turing Machine, the following considerations need to be taken into account:
1. Deterministic Transition Function: The transition function of a DTM should be defined in such a way that for every possible combination of the current state and the symbol under the head, there is at most one valid transition.
2. Deterministic Tape Operations: The tape operations (reading and writing symbols) in a DTM should also be deterministic. That means, for a given configuration, there should be a unique symbol to be written on the tape.
3. Deterministic Halting: The DTM should have a well-defined halting behavior. That is, for every input, the computation should eventually halt, either in an accepting or a rejecting state.

Designing a DTM involves carefully defining the transition function, tape operations, and halting behavior to ensure deterministic behavior. By maintaining determinism, the DTM guarantees that it will always follow a unique path during the computation, simplifying the analysis and implementation of the Turing Machine.

Non- deterministic TM

A Non-deterministic Turing Machine (NTM) is a type of Turing Machine (TM) that allows for multiple possible transitions for a given state and symbol combination. In other words, given the current state and the symbol under the head, an NTM can have multiple choices for the next configuration.



The non-deterministic property of an NTM means that it can explore different computation paths simultaneously, branching out into multiple possibilities during its operation. It can be viewed as a "guessing" machine that can try different choices and paths in parallel.

To design a Non-deterministic Turing Machine, the following considerations need to be taken into account:
1. Non-deterministic Transition Function: The transition function of an NTM allows for multiple possible transitions for a given combination of the current state and the symbol under the head. These transitions can be defined explicitly, indicating all the possible choices, or they can be defined using non-deterministic rules.
2. Non-deterministic Tape Operations: The tape operations (reading and writing symbols) in an NTM can also be non-deterministic. For a given configuration, there can be multiple symbols that the NTM can write on the tape.
3. Accepting Criteria: The NTM can have different accepting criteria based on its design. For example, it can halt and accept the input if any of its computation paths leads to an accepting state, or it can require all computation paths to reach an accepting state.

Designing an NTM involves defining the non-deterministic transition function, tape operations, and accepting criteria. The non-deterministic behavior allows the NTM to explore multiple computation paths simultaneously, potentially leading to different outcomes for different inputs.

It's important to note that non-deterministic Turing Machines do not violate the Church-Turing Thesis. Despite their ability to explore multiple paths, they are still equivalent in computational power to deterministic Turing Machines. This equivalence is known as the Church-Turing Thesis, which states that any effectively calculable function can be computed by a Turing Machine or its equivalent.