Automata & Complexity Theory

Institute of Technology | Faculty of Computing and Software Engineering

Anteneh Tilahun

4/1/23

CoSc 3101

Course Objectives

By the end of this course, students will be able to:

- Introduce concepts in automata theory and theory of computation
- Study the central concepts of automata theory
- Acquire insights into the relationship among formal languages, formal grammars, and automata.
- Identify different formal language classes and their relationships
- Design grammars and recognizer for different formal languages
- Explain Models of Computation, resources (time and space), algorithms, computability, and complexity.
- Understand Complexity classes, P/NP/PSPACE, reductions, hardness, completeness, hierarchy, relationships between complexity classes.
- Learn Randomized computation and complexity; Logical characterizations, incompleteness and approximately

Contents

Chapter 1

Introduction

Automata theory and complexity theory are two closely related branches of computer science that deal with the study of formal languages, automata, and computational complexity.

Automata theory is concerned with the study of abstract computing devices, called automata, which are used to recognize patterns or languages. Automata come in different forms, such as finite automata, pushdown automata, and Turing machines, and are used to study formal languages, which are sets of strings generated by a set of rules.

Formal languages are studied in the context of formal grammars, which are sets of production rules for generating strings in a language. There are different types of formal grammars, such as regular grammars, context-free grammars, and context-sensitive grammars, each with its own expressive power.

Computational complexity theory, on the other hand, deals with the study of the resources required to solve computational problems. It is concerned with classifying problems according to their level of difficulty, and with identifying efficient algorithms for solving them. Complexity theory also deals with the study of complexity classes, which are sets of decision problems with similar levels of difficulty.

The most famous complexity class is P, which contains problems that can be solved in polynomial time. There are also other complexity classes, such as NP, which contains problems that can be verified in polynomial time, and PSPACE, which contains problems that can be solved in polynomial space.

The study of automata and complexity theory has important applications in various fields, such as computer science, mathematics, linguistics, and philosophy.

Alphabets and strings

In automata theory, an alphabet is a non-empty, finite set of symbols. These symbols are the building blocks used to form strings, which are finite sequences of symbols from the alphabet. The empty string, denoted by ε, is a string with length 0.

Formally, an alphabet Σ is a non-empty, finite set of symbols, and a string over Σ is a finite sequence of symbols from Σ. The length of a string is the number of symbols in the sequence. For example, if $\Sigma = \{0, 1\}$, then 0110 is a string over Σ with length 4.

Automata are mathematical models of computing devices that accept or reject strings of symbols from an alphabet. Different types of automata, such as finite automata, pushdown automata, and Turing machines, correspond to different levels of computational power.

A finite automaton is a type of automaton that reads a string of symbols from an alphabet and either accepts or rejects the string based on a set of rules called transitions. A finite automaton consists of a finite set of states, a start state, a set of accepting states, and a set of transitions. The

transitions specify how the automaton moves from one state to another based on the symbols read from the input string.

A regular language is a language that can be recognized by a finite automaton. Regular languages have many applications, such as in pattern matching, lexical analysis, and text processing.

In summary, an alphabet is a finite set of symbols used to form strings, and strings are finite sequences of symbols from the alphabet. Automata are mathematical models of computing devices that accept or reject strings of symbols from an alphabet, and regular languages are languages that can be recognized by finite automata.

Languages and Grammars

In automata theory, a language is a set of strings over an alphabet. Formally, a language L is a subset of Σ^*, where Σ is an alphabet and Σ^* is the set of all possible strings over Σ. For example, if $\Sigma = \{0, 1\}$, then the language $L = \{0, 1, 00, 01, 10, 11\}$ is a language over Σ.

Languages can be described using formal grammars, which are sets of rules for generating strings in a language. A formal grammar consists of a set of nonterminal symbols, a set of terminal symbols (which are the symbols in the alphabet Σ), a start symbol, and a set of production rules that describe how nonterminal symbols can be replaced by sequences of terminal and nonterminal symbols.

There are several types of formal grammars, each with its own expressive power. The most common types are regular grammars, context-free grammars, and context-sensitive grammars.

A regular grammar is a formal grammar that generates a regular language. Regular grammars have production rules of the form $A \rightarrow aB$ or $A \rightarrow a$, where A and B are nonterminal symbols, a is a terminal symbol, and ε is the empty string.

A context-free grammar is a formal grammar that generates a context-free language. Context-free grammars have production rules of the form $A \rightarrow \alpha$, where A is a nonterminal symbol and α is a string of terminals and non-terminals.

A context-sensitive grammar is a formal grammar that generates a context-sensitive language. Context-sensitive grammars have production rules of the form $\alpha B\beta \rightarrow \alpha\gamma\beta$, where α and β are strings of terminals and non-terminals, B is a nonterminal symbol, and γ is a string of terminals and non-terminals such that γ is not the empty string.

Languages can also be recognized by automata, such as finite automata, pushdown automata, and Turing machines. Regular languages can be recognized by finite automata, while context-free languages can be recognized by pushdown automata. Turing machines can recognize recursively enumerable languages, which are languages that can be generated by a type-0 grammar.

In summary, in automata theory, a language is a set of strings over an alphabet, and formal grammars are sets of rules for generating strings in a language. Different types of formal grammars correspond to different types of languages, and different types of automata can recognize different types of languages.

Automata

In automata theory, an automaton is a mathematical model of a computing device that accepts or rejects strings of symbols from an alphabet. Automata come in different forms, such as finite automata, pushdown automata, and Turing machines, and are used to study formal languages, which are sets of strings generated by a set of rules.

A finite automaton is a type of automaton that reads a string of symbols from an alphabet and either accepts or rejects the string based on a set of rules called transitions. A finite automaton consists of a finite set of states, a start state, a set of accepting states, and a set of transitions. The transitions specify how the automaton moves from one state to another based on the symbols read from the input string.

A pushdown automaton is a type of automaton that reads a string of symbols from an alphabet and either accepts or rejects the string based on a stack, which is a data structure that stores symbols. A pushdown automaton consists of a finite set of states, a start state, a set of accepting states, a stack alphabet, and a set of transitions. The transitions specify how the automaton moves from one state to another and how it interacts with the stack based on the symbols read from the input string.

A Turing machine is a type of automaton that reads a string of symbols from an alphabet and either accepts or rejects the string based on a tape, which is a one-dimensional array of cells that can store symbols. A Turing machine consists of a finite set of states, a start state, a set of accepting states, a tape alphabet, and a set of transitions. The transitions specify how the machine moves from one state to another and how it interacts with the tape based on the symbols read from the input string.

Automata are used to study formal languages, which are sets of strings generated by a set of rules. Different types of automata correspond to different types of formal languages. For example, regular languages can be recognized by finite automata, while context-free languages can be recognized by pushdown automata. Turing machines can recognize recursively enumerable languages, which are languages that can be generated by a type-0 grammar.

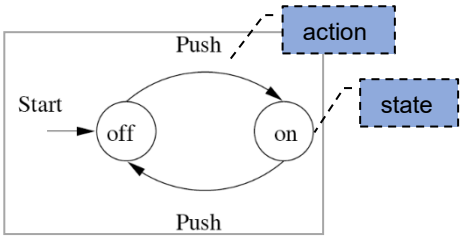
In summary, automata are mathematical models of computing devices that accept or reject strings of symbols from an alphabet. Different types of automata correspond to different types of formal languages, and automata are used to study the properties of these languages.

Finite automata, Deterministic and Non-deterministic finite automata

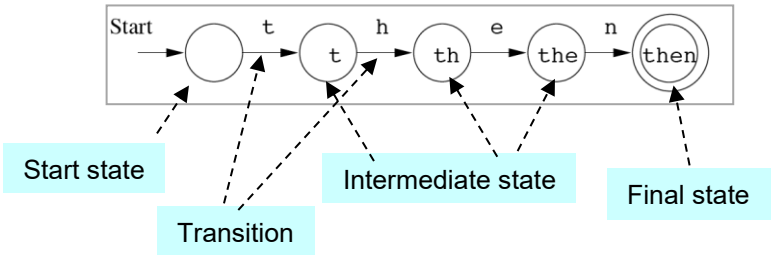
A finite automaton (FA), also called a finite state machine (FSM), is a type of automaton that recognizes languages generated by a regular grammar. It has a finite number of states and reads input symbols from an input alphabet. The automaton transitions from state to state based on the input symbols and the transitions are determined by a set of rules.

Example

- On/Off switch



- Modeling recognition of the word "*then*"

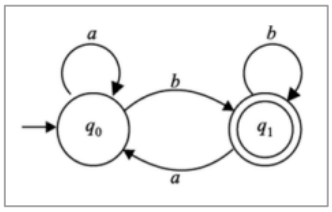


Transition Table (Transition Matrix):

A transition table is a table that shows the transitions of a finite automaton from one state to another based on the input symbols. The table has one row for each state and one column for each input symbol in the alphabet.

Here's an example of a transition table for a finite automaton:

Input/ State	a	b
q_0	q_0	q1
q1	q_0	q1



Transition Function:

The transition function, denoted by δ, is a mathematical function that defines the behavior of a finite automaton. It maps a pair consisting of a state and an input symbol to a new state.

Formally, we can define the transition function as follows:

$$\delta : Q \times \Sigma \rightarrow Q$$

where Q is the set of states and Σ is the input alphabet.

For example, let's consider a finite automaton M with two states, q0 and q1, and input alphabet Σ = {0, 1}. The transition function δ for this automaton can be defined as follows:

- δ(q0, 0) = q1
- δ(q0, 1) = q0
- δ(q1, 0) = q0
- δ(q1, 1) = q1

This transition function maps each pair consisting of a state and an input symbol to a new state. For example, if the automaton is in state q0 and receives an input symbol 0, it transitions to state q1, because δ(q0, 0) = q1.

The transition function can be used to determine the sequence of states that the automaton goes through for a given input string. For example, if the input string is "101", we start in state q0, transition to q1 on the first input symbol 1, transition to q0 on the second input symbol 0, and finally transition to q1 on the third input symbol 1. Since q1 is a final state, the automaton accepts the input string.

Example: Design a FA which checks whether the given binary number is even.

Even: any binary number ends with 0

Ex: 2 => 001<u>0</u>, 4 => 010<u>0</u>, 6 => 011<u>0</u>, 8 => 100<u>0</u>

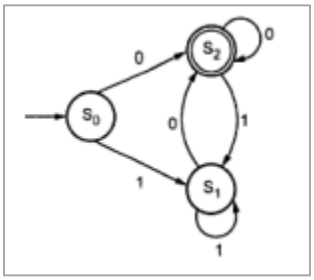
Odd: any binary number ends with 1

Ex: 1 => 000<u>1</u>, 3 => 001<u>1</u>, 5 => 010<u>1</u>, 7 => 011<u>1</u>

While designing FA, assume one start state, One state ending with 0 and another one ends with 1. we want to check whether the given binary number is even or odd so fix the final state for 0 value.

Let take 2 sample inputs one odd and one even:

Case 1: For even number	Case 2: For odd number
1000 => $1S_1000$	1011 => $1\ S_1011$
=> $10S_200$	=> $10S_211$
=> $100S_20$	=> $101S_11$
=> $1000S_2$	=> $1011S_1$



In case1: now at the end of input we are at final or accept state so its even number

In case2: now at the end of input we are at S1 state not in final state so its not even (odd) number

Exercise

1. Design a FA which accepts only those strings which starts with 1 and ends with 0
2. Design FA which accepts even number of 0's and even number of 1's
3. Consider the given transition table and Find the language accepted by the finite automata

States	Inputs	
	0	1
q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Types of Finite Automata

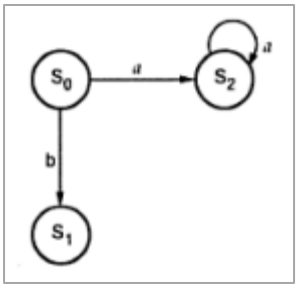
There are two types of finite automata: deterministic finite automata (DFA) and non-deterministic finite automata (NFA).

Deterministic finite automaton (DFA)

A deterministic finite automaton (DFA) is a type of finite automaton where, given the current state and the next input symbol, it can transition to only one next state (*there is only one path for a specific input from current state to next state*). A DFA can be represented by a 5-tuple, (Q, Σ, δ, q0, F),

where:

- Q is a finite set of states
- Σ is a finite input alphabet
- δ is a transition function, δ: Q × Σ → Q, which maps each state and input symbol to the next state
- q0 is the start state
- F is a set of accepting or final states



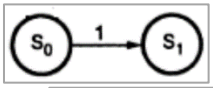
Example

From the given example transition diagram

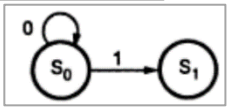
From state S_0 for input 'a' there is only one path going to state S_2
From state S_0 for input 'b' there is only one path going to state S_1
From state S_2 for input 'a' there is only one path going to state S_2

Example: Design a DFA for accepting all strings of L = $\{0^m1^n \mid m \geq 0 \text{ and } n \geq 1\}$

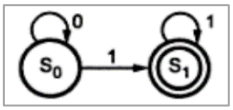
Step 1: m=0 and n=1 => 1



Step 2: m=1 and n=1 => 01



Step 3: m=1 and n=2 => 011



Input State	0	1
S0	S0	S1
S1	-	S1

Non-deterministic Finite Automata (NFA)

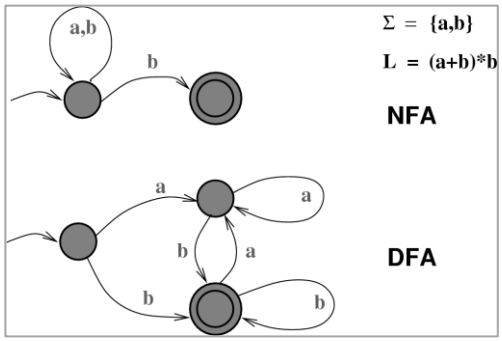
A non-deterministic finite automaton (NFA) is a type of finite automaton where, given the current state and the next input symbol, it can transition to one or more next states. In other words, the transition function of an NFA is not deterministic (*Many paths for a specific input from current state to next state*). An NFA can be represented by a 5-tuple, (Q, Σ, δ, q0, F),

where:

- Q is a finite set of states
- Σ is a finite input alphabet
- δ is a transition function, δ: Q × Σ → P(Q), which maps each state and input symbol to a set of next states
- q0 is the start state

- F is a set of accepting or final states

Example: A NFA and a DFA accepting the same language.



Nondeterministic finite automata are more powerful than deterministic finite automata in the sense that they can recognize some languages that cannot be recognized by DFAs. However, NFAs are more complex to work with than DFAs.

NFA with ε Transition:

A nondeterministic finite automaton (NFA) with ε transitions is a type of finite automaton where the state transition function can include ε transitions, which allow the automaton to move from one state to another without consuming any input symbol.

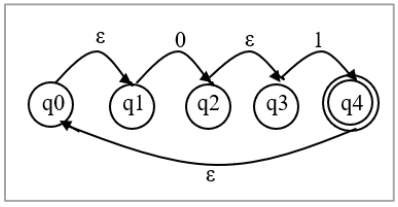
Formally, an NFA with ε transitions can be defined as a 5-tuple (Q, Σ, δ, q0, F),

where:

- Q is a finite set of states
- Σ is a finite input alphabet
- δ is a state transition function that maps Q × (Σ ∪ {ε}) to the power set of Q (i.e., a subset of Q)
- q0 is the initial state
- F is a set of final states

The ε transition represents a null or empty transition, where the automaton can move from one state to another without reading any input symbol. This allows the automaton to make non-deterministic choices and explore different paths in the input string.

Here's an example of an NFA with ε transitions:



In this example, the automaton has five states: q0, q1, q2, q3, and q4. The input alphabet is Σ = {0, 1}. The transition function δ is defined as follows:

- δ(q0, ε) = {q1}
- δ(q1, 0) = {q2}
- δ(q2, ε) = {q3}

- $\delta(q3, 1) = \{q4\}$
- $\delta(q4, \varepsilon) = \{q0\}$

The initial state is q0, and the final state is q4.

The ε transitions allow the automaton to move from q0 to q1 without reading any input symbol, and from q2 to q3 without reading any input symbol. Therefore, the automaton can accept input strings such as "0", "10", "100", "1001", and so on.

In summary, finite automata are mathematical models of computing devices that recognize languages generated by regular grammars. There are two types of finite automata: deterministic finite automata (DFA) and non-deterministic finite automata (NFA). DFAs have a deterministic transition function, while NFAs have a non-deterministic transition function.

Definition of ε – closure:

The ε-closure (epsilon-closure) of a state q in a nondeterministic finite automaton (NFA) with ε transitions is defined as the set of all states that can be reached from q by following ε transitions, including q itself.

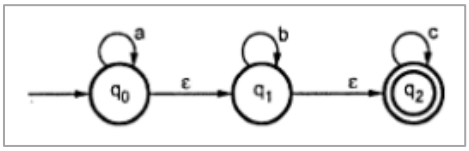
Formally, we can define the ε-closure of a state q as follows:

$$\varepsilon\text{-closure}(q) = \{q\} \cup \{p \mid \text{there exists an } \varepsilon\text{-transition from } q \text{ to } p\}$$

In other words, the ε-closure of a state q includes q itself, as well as all states that can be reached from q by following ε transitions.

The ε-closure is an important concept in the theory of NFAs with ε transitions, because it allows us to determine all the possible states that the automaton can be in after reading a sequence of ε transitions. This is useful when converting an NFA with ε transitions to an equivalent deterministic finite automaton (DFA), because we need to determine the set of states that the DFA can be in after reading a sequence of input symbols and ε transitions.

Example: Find the ε – closure for the following NFA with ε



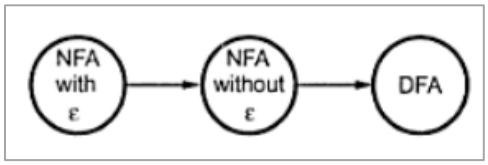
$\varepsilon - \text{closure}(q_0) = \{q_0, q_1, q_2\}$ means self state + ε – reachable states

$\varepsilon - \text{closure}(q_1) = \{ q_1, q_2\}$ q_1 is a self state + q_2 is a state obtained from q_1 with ε input.

$\varepsilon - \text{closure}(q_2) = \{ q_2\}$

Conversions and Equivalence:

NFA with ε can be converted to NFA without ε and this NFA without ε can be converted to DFA. Here we are going to discuss these conversions and will find them equivalent to each other's



Conversion from NFA with ε to NFA without ε:

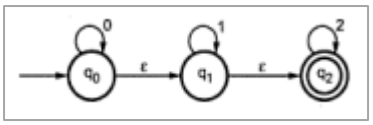
Step1: Find out all the ε transition from each state from Q. (find ε – closure for each states from Q)

Step2: Find δ' transition (means ε – closure on δ with ε moves)

Step3: repeat step2 for each input symbol and for each state

Step4: From the resultant transition table can construct the NFA without ε

Example: Convert the given NFA with ε to NFA without ε



Step1: Find ε – closure for each states:

ε – closure(q_0) = {q_0, q_1, q_2}

ε – closure(q_1) = {q_1, q_2}

ε – closure(q_2) = {q_2}

Step2: Find δ' transition for each state on each input symbol: and ***Step3***

δ'(q_0,0) = ε – closure (δ(δ(q_0, ε),0))
= ε – closure (δ(ε – closure (q_0),0))
= ε – closure (δ(q_0, q_1, q_2),0)
= ε – closure(δ(q_0,0) U δ(q_1,0) U δ(q_2,0))
= ε – closure(q_0 U ø U ø)
= ε – closure(q_0)
δ'(q_0,0) = {q_0, q_1, q_2}

δ'(q_1,0) = ε – closure (δ(δ(q_1, ε),0))
= ε – closure (δ(q_1, q_2),0)
= ε – closure(δ(q_1,0) U δ(q_2,0))
= ε – closure(ø U ø)
= ε – closure(ø)
δ'(q_1,0) = ø

δ'(q_2,0) = ε – closure (δ(δ(q_2, ε),0))
= ε – closure (δ(q_2,0))
= ε – closure(ø)
δ'(q_2,0) = ø

δ'(q_0,2) = ε – closure (δ(δ(q_0, ε),2))
= ε – closure (δ(q_0, q_1, q_2),2)
= ε – closure(δ(q_0,2) U δ(q_1,2) U δ(q_2,2))
= ε – closure(ø U ø U q_2)
= ε – closure(q_2)
δ'(q_0,2) = {q_2}

δ'(q_0,1) = ε – closure (δ(δ(q_0, ε),1))
= ε – closure (δ(q_0, q_1, q_2),1)
= ε – closure(δ(q_0,1) U δ(q_1,1) U δ(q_2,1))
= ε – closure(ø U q_1 U ø)
= ε – closure(q_1)
δ'(q_0,1) = {q_1, q_2}

δ'(q_1,1) = ε – closure (δ(δ(q_1, ε),1))
= ε – closure (δ(q_1, q_2),1)
= ε – closure(δ(q_1,1) U δ(q_2,1))
= ε – closure(q_1 U ø)
= ε – closure(q_1)
δ'(q_1,1) = {q_1, q_2}

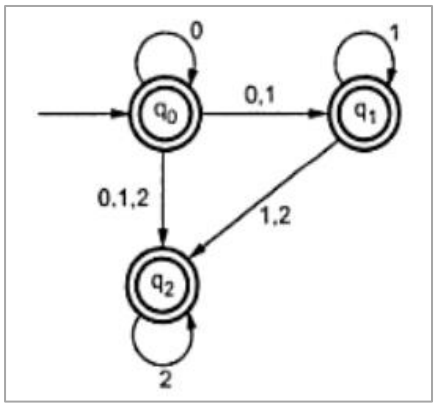
δ'(q_2,1) = ε – closure (δ(δ(q_2, ε),1))
= ε – closure (δ(q_2),1)
= ε – closure(ø)
δ'(q_2,1) = ø

δ'(q_1,2) = ε – closure (δ(δ(q_1, ε),2))
= ε – closure (δ(q_1, q_2),2)
= ε – closure(δ(q_1,2) U δ(q_2,2))
= ε – closure(ø U q_2)
= ε – closure(q_2)
δ'(q_1,2) = {q_2}

δ'(q_2,2) = ε – closure (δ(δ(q_2, ε),2)) => ε – closure (δ(q_2,2)) => ε – closure(q_2) => δ'(q_2,2) = {q_2}

Step4: Transition table and NFA

State \ Input	0	1	2
q_0	$\{q_0,q_1,q_2\}$	$\{q_1,q_2\}$	$\{q_2\}$
q_1	ø	$\{q_1,q_2\}$	$\{q_2\}$
q_2	ø	ø	$\{q_2\}$



Here q_0, q_1 and q_2 is a final state because ε – closure (q_0), ε – closure (q_1) and ε – closure (q_2) contains final state q_2.

Exercise 1

1. Which of the following is a correct statement about automata theory?

   a) It is a branch of computer science that studies algorithms for solving complex problems
   b) It is a branch of mathematics that studies the behavior of computing devices
   c) It is a branch of linguistics that studies the structure of natural languages
   d) It is a branch of physics that studies the behavior of atoms

2. What is an alphabet in automata theory?

   a) A set of rules for generating strings in a language
   b) A set of symbols used to form strings
   c) A set of states in a finite automaton
   d) A set of transitions in a pushdown automaton

3. Which of the following is an example of a regular language?

   a) The set of all palindromes over an alphabet
   b) The set of all context-free grammars
   c) The set of all strings that contain an even number of 0s and an odd number of 1s
   d) The set of all strings that start with 1 and end with 0

4. What is a finite automaton in automata theory?

   a) A mathematical model of a computing device that recognizes languages generated by context-free grammars
   b) A mathematical model of a computing device that recognizes languages generated by regular grammars
   c) A mathematical model of a computing device that recognizes any language
   d) A mathematical model of a computing device that recognizes languages generated by context-sensitive grammars

5. What is the difference between a deterministic finite automaton (DFA) and a non-deterministic finite automaton (NFA)?

   a) DFAs can recognize more languages than NFAs
   b) DFAs have a non-deterministic transition function, while NFAs have a deterministic transition function
   c) DFAs have a deterministic transition function, while NFAs have a non-deterministic transition function
   d) There is no difference between DFAs and NFAs

6. Which of the following is a correct statement about regular grammars?

   a) They can generate any language
   b) They can only generate regular languages
   c) They can generate context-free languages

d) They can generate context-sensitive languages

7. Which of the following is an example of a context-free grammar?

a) $S \rightarrow aSb \mid \varepsilon$
b) $A \rightarrow aB \mid bA \mid \varepsilon$, $B \rightarrow b \mid A$
c) $S \rightarrow aS \mid Sb \mid \varepsilon$
d) $S \rightarrow aSb \mid bSa \mid \varepsilon$

8. What is a pushdown automaton in automata theory?

a) A mathematical model of a computing device that recognizes languages generated by context-sensitive grammars
b) A mathematical model of a computing device that recognizes languages generated by context-free grammars
c) A mathematical model of a computing device that recognizes languages generated by regular grammars
d) A mathematical model of a computing device that recognizes any language

9. Which of the following is a correct statement about Turing machines?

a) They can recognize any language
b) They can only recognize regular languages
c) They can only recognize context-free languages
d) They can recognize recursively enumerable languages

10. Which of the following is a correct statement about formal languages?

a) They are sets of rules for generating strings in an alphabet
b) They are sets of strings generated by a set of rules
c) They are sets of nonterminal symbols used to form strings
d) They are sets of terminal symbols used to form strings

Chapter 2

Regular Expression and Regular languages

In automata theory, regular expressions are a notation for describing regular languages. A regular expression is a string of symbols that represents a pattern or a set of strings. Regular expressions can be used to define regular languages, which are sets of strings that can be generated by a regular grammar.

Regular expressions have a specific syntax that allows the construction of complex patterns from simpler ones. Some of the basic operations that can be used to build regular expressions include:

- Concatenation: combining two regular expressions to form a new one
- Alternation: creating a new regular expression that matches either of two given regular expressions
- Kleene star: indicating that a regular expression can be repeated zero or more times
- Grouping: enclosing a part of a regular expression in parentheses to create a subexpression

For example, the regular expression (0|1)* represents the set of all binary strings, where each symbol can be either 0 or 1, and can occur any number of times (including zero times). Another example is the regular expression (a|b)*abb, which represents the set of all strings that end with the substring "abb".

Regular languages are a class of formal languages that can be generated by regular grammars or recognized by finite automata. Regular languages include simple patterns such as the set of all binary strings, the set of all strings that start with a certain symbol or substring, and the set of all strings that contain a certain symbol or substring.

Regular languages have several useful properties, such as closure under union, intersection, and complementation, which means that if two regular languages are combined using these operations, the resulting language is still regular. Regular languages can also be recognized in linear time, which makes them efficient to process.

In summary, regular expressions are a notation for describing regular languages, which are sets of strings that can be generated by a regular grammar or recognized by a finite automaton. Regular expressions can be used to construct complex patterns from simpler ones by using operations such as concatenation, alternation, Kleene star, and grouping. Regular languages have several useful properties, such as closure under union, intersection, and complementation, and can be recognized in linear time.

Regular expressions

In automata theory, a regular expression is a sequence of characters that defines a pattern. It is used to describe a set of strings that can be generated by a regular grammar or recognized by a finite automaton. Regular expressions provide a concise and powerful way to specify patterns and are widely used in programming languages, text editors, and search engines.

Regular expressions are typically written using a special syntax that includes a set of operators and special characters. Some of the basic operators that can be used in regular expressions include:

- Concatenation: The concatenation operator is used to combine two or more regular expressions to form a new regular expression. For example, the regular expression "ab" is the concatenation of "a" and "b".
- Alternation: The alternation operator is used to specify a choice between two or more regular expressions. For example, the regular expression "a|b" matches either "a" or "b".
- Kleene star: The Kleene star operator is used to specify that the preceding regular expression can be repeated zero or more times. For example, the regular expression "a*" matches any number of occurrences of the letter "a".
- Grouping: Parentheses can be used to group regular expressions together and specify the order of operations. For example, the regular expression "(ab)*" matches any number of occurrences of the string "ab".
- Character classes: Character classes are used to match any character from a specified set. For example, the regular expression "[a-z]" matches any lowercase letter.

Regular expressions can be used to match patterns in strings, such as email addresses, phone numbers, or URLs. They can also be used to extract specific parts of a string, such as a date or a price.

For example, the regular expression "\b\d{3}-\d{2}-\d{4}\b" matches a social security number in the format of "XXX-XX-XXXX", where X is a digit. The "\b" anchors the regular expression to word boundaries, and "\d" matches any digit.

Example: Write the regular expression for the language accepting all combination of a's over the input set ∑ = {a}

Regular Expression R = a* (kleen closure)

All combination of a's means: 'a' may be zero times, one times, two times and three time etc.,

L = {ε , a, aa, aaa, aaaa, ….}

Example: Write the regular expression for the language accepting all combination of a's except the null string over the input set ∑ = {a}

Regular Expression R = a + (positive closure)

All combination of a's means: 'a' may be one times, two times and three time etc.,

L = {a, aa, aaa, aaaa, ….}

Example: Design regular expression for all the strings containing any number of a's and b's

R = (a+b)*

Any combination of a and b means any combination of a and b even null string

L = {ε, a, b, aa, ab, ba, bb, aba,…. }

Example: Construct the RE for language accepting all the stings which are ending with 00 over the input set ∑ = {0, 1}

R = (any combination of 0's and 1's) 00
R = (0 + 1)* 00
L = {00, 000, 100, 1000,0100 …. String ending with 00}

Example: If L = {stating and ending with 'a' and having any combination of b's in between } what is R?

R = a b* a

Example: Construct the RE for language accepting all the stings which are having any number of a's is followed by any number b's is followed by any number of c's over the input set ∑ = {a, b, c}

R = (any number of a) (any number of b) (any number of c)
R = a* b* c*

Example: write a RE to denote the language L over ∑ ={a, b} such that all the strings do not contain the substring ab

L = {ε, a, b, bb, ba, baa, …..} R = b* a*

Regular expressions are supported by many programming languages, such as Perl, Python, Java, and JavaScript. They are also used in Unix utilities, such as grep and sed, to search and manipulate text files.

In summary, regular expressions are a powerful tool for specifying patterns in strings. They consist of a sequence of characters and operators that define a set of strings that can be generated by a regular grammar or recognized by a finite automaton. Regular expressions are widely used in programming languages, text editors, and search engines to match and manipulate strings.

Relationship between regular expression and finite automata

Regular expressions and finite automata are two different but equivalent ways of describing regular languages. A regular expression is a string that describes a set of strings, while a finite automaton is a mathematical model that recognizes or generates a language.

The relationship between regular expressions and finite automata can be summarized as follows:

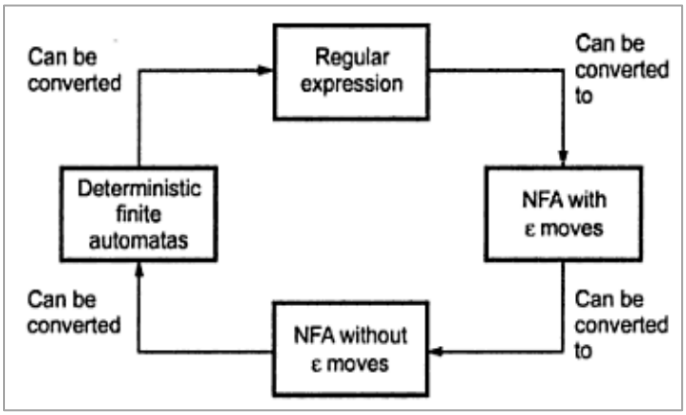
1. Every regular expression corresponds to a finite automaton that recognizes the language described by the regular expression. This is known as the "regular expression to NFA" conversion.
2. Every finite automaton corresponds to a regular expression that describes the language recognized by the automaton. This is known as the "NFA to regular expression" conversion.
3. Every regular language can be described by a regular expression and recognized by a finite automaton. This is known as the "regular expression-finite automaton equivalence".

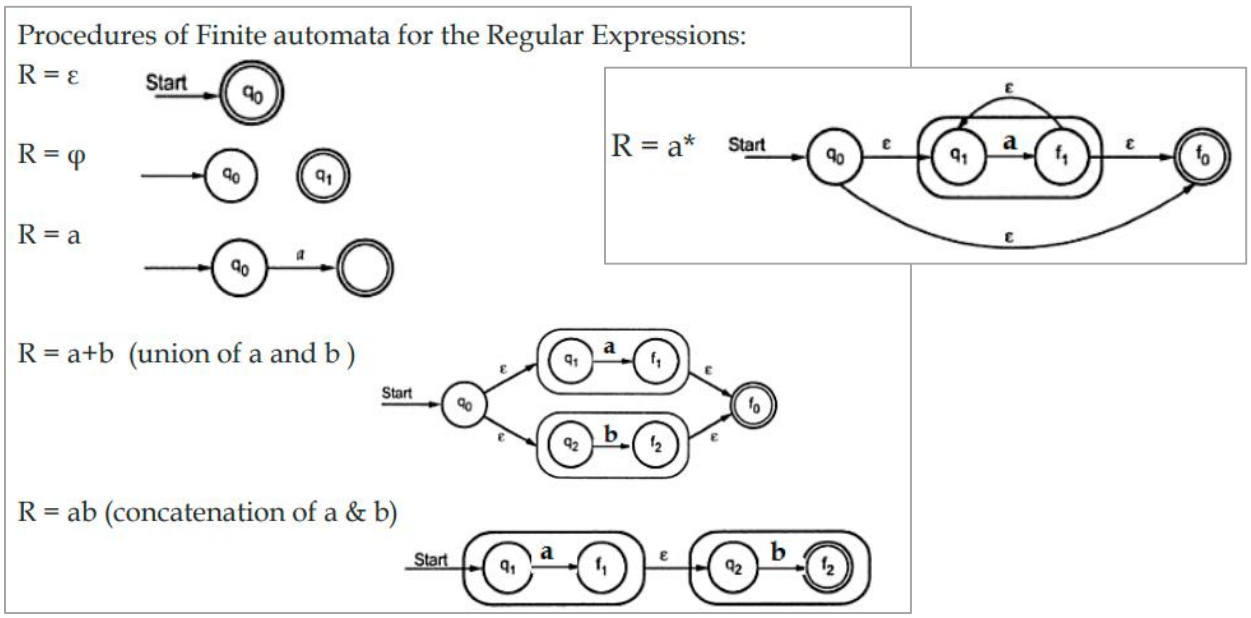
The regular expression to NFA conversion involves constructing an NFA that recognizes the language described by the regular expression. This is typically done by recursively constructing NFAs for subexpressions of the regular expression and then combining them using union, concatenation, and Kleene star operations.

The NFA to regular expression conversion involves eliminating states from the NFA until only the initial and final states remain, and constructing a regular expression that describes the language accepted by the remaining states.

The regular expression-finite automaton equivalence is a fundamental result in the theory of regular languages. It means that any regular language can be described by a regular expression, which in turn can be recognized by a finite automaton. This equivalence is important because it allows us to use regular expressions or finite automata interchangeably when working with regular languages.

Generally, regular expressions and finite automata are two different but equivalent ways of describing regular languages. Regular expressions can be converted to NFAs and vice versa, and every regular language can be described by a regular expression and recognized by a finite automaton.





Connection between regular expression and regular languages

Regular expressions and regular languages are closely connected in automata theory. A regular expression is a compact way of specifying a regular language, which is a set of strings that can be generated by a regular grammar or recognized by a finite automaton.

In other words, a regular expression defines a pattern that can be used to match strings in a regular language. For example, the regular expression "a*b*" matches any string that consists of zero or more occurrences of the letter "a" followed by zero or more occurrences of the letter "b". This regular expression defines the regular language {ε, a, b, ab, aab, abb, aaab, aabb, ...}.

Conversely, any regular language can be expressed as a regular expression. This is known as the Kleene's theorem, which states that every regular language can be generated by a regular grammar, recognized by a finite automaton, or expressed as a regular expression.

For example, the regular language {0, 1, 00, 01, 10, 11, 000, 001, ...} can be expressed as the regular expression "(0|1)(0|1)*", which matches any string that starts with either "0" or "1" followed by zero or more occurrences of "0" or "1".

Regular expressions provide a concise and intuitive way of specifying regular languages. They are widely used in programming languages, text editors, and search engines to search, match, and manipulate strings. Regular languages, on the other hand, have several useful properties, such as closure under union, intersection, and complementation, which make them efficient to process.

In summary, regular expressions and regular languages are intimately connected in automata theory. Regular expressions provide a compact way of specifying regular languages, and any regular language can be expressed as a regular expression. Regular languages have several useful properties and can be efficiently recognized by finite automata or generated by regular grammars.

Regular Grammar

In automata theory, a regular grammar is a formal grammar that generates a regular language. A regular language is a set of strings that can be recognized by a finite automaton, or equivalently, generated by a regular expression.

A regular grammar is defined by a set of production rules, each of which takes the form of a nonterminal symbol followed by a terminal symbol or a nonterminal symbol. The nonterminal symbols represent variables or placeholders, while the terminal symbols represent the actual symbols of the language.

The production rules of a regular grammar can be classified into three types:

1. Start rule: This is a single rule that specifies the starting symbol of the grammar. The starting symbol is usually denoted by the symbol S.
2. Terminal rule: These are rules that map a nonterminal symbol to a single terminal symbol. For example, the rule A → a maps the nonterminal symbol A to the terminal symbol a.
3. Nonterminal rule: These are rules that map a nonterminal symbol to a concatenation of a terminal symbol and a nonterminal symbol. For example, the rule A → aB maps the

nonterminal symbol A to the concatenation of the terminal symbol a and the nonterminal symbol B.

Regular grammars can be used to generate or recognize regular languages. For example, the regular grammar with the start rule S → 0A | 1B, terminal rules A → 0A | 1B | ε and B → 0A | 1B | ε generates the regular language {0, 1, 00, 01, 10, 11, 000, 001, ...}, which consists of all strings of 0s and 1s. This language can also be recognized by a deterministic finite automaton with two states.

Alternative way of specifying languages

Definition: G = (N, T, P, S)

N – set of non terminals

T – set of terminals

P – production rules

S – start symbol

Example: **G = (N, T, P, S)**

N = {S, A}

S → Start symbol

P = {S →0S | S →1A | A →ε}

T = {0, 1}

Regular grammars have several useful properties, such as closure under union, concatenation, and Kleene star, which means that if two regular languages are combined using these operations, the resulting language is still regular. Regular grammars can also be transformed into finite automata or regular expressions, which makes them useful for practical applications.

In summary, a regular grammar is a formal grammar that generates a regular language. It is defined by a set of production rules that map nonterminal symbols to terminal symbols or concatenations of terminal symbols and nonterminal symbols. Regular grammars have several useful properties and can be transformed into finite automata or regular expressions.

Types

Right – linear grammar

If the non-terminal symbol appears as rightmost symbol in each production rules of the regular grammar.

A → aB --------- rule 1

A → bA --------- rule 2

A → ε --------- rule 3

In rule1 non terminal B is appeared in rightmost

In rule2 non terminal A is appeared in rightmost

Left – linear grammar

If the non-terminal symbol appears as leftmost symbol in each production rules of the regular grammar.

A → Ba --------- rule 1

A → Ab --------- rule 2

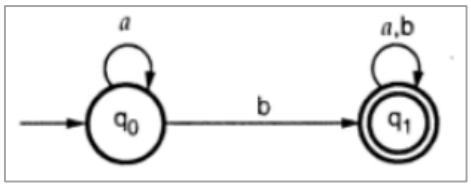
A → ε --------- rule 3

In rule1 non terminal B is appeared in leftmost

In rule2 non terminal A is appeared in leftmost

Regular grammar for Finite Automata

Example: construct the regular grammar for the given FA.



Consider the outgoing transitions for production rule

Grammar equivalent for the given above DFA

G = (N, T, P, S)

In the above DFA two states available, for that two states fix the non-terminals so,

N = {A0, A1}

T = {a, b}

S = A0

Transition from q0

Transition q0 to q0 - A0→ a A0

Transition q0 to q1 - A0→ a A1

- A0→ a (final state)

Transition from q1

Transition q1 to q1 - A1→ a A1

- A1→ a (final state)

Transition q1 to q1 - A1→ b A1

- A1→ b (final state)

finally P = {A0→ a A0, A0→ a A1, A0→ a, A1→ a A1, A1→ a A1→ b A1, A1→ b}

Finite Automata for Regular grammar

Example: construct a FA recognizing L(G) where G is grammar

S → aS | bA | b

A → aA |bS | a

Finite Automata M = ({q0, q1, qf }, {a, b}, δ, q0, {qf})

Where q0 and q1 corresponds to S and A

qf is final state

S → aS - transition from q0 to q0 with label a

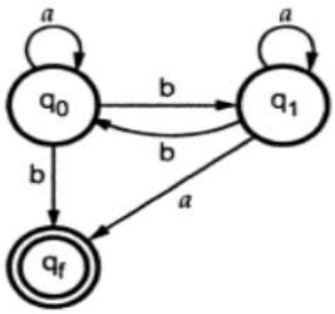
S → bA - transition from $q0$ to $q1$ with label b

S → b - transition from $q0$ to qf with label b

A → aA - transition from q1 to q1 with label a

A → bS - transition from q1 to q0 with label b

A → a - transition from q1 to qf with label a



Pumping lemma and non-regular language grammars

The pumping lemma for regular languages is a powerful tool used to prove that a given language is not regular. It states that for any regular language L, there exists a pumping length p such that any string s in L of length at least p can be divided into three parts, s = xyz, such that:

1. The length of y is greater than 0.
2. The length of xy is less than or equal to p.
3. For any non-negative integer n, the string xy^n z is also in L.

In other words, if a language is regular, then there exists a pumping length p such that any sufficiently long string in the language can be pumped (i.e., repeated) any number of times and still remain in the language.

To use the pumping lemma to prove that a language is not regular, one needs to assume that the language is regular and then derive a contradiction by showing that there exists a string in the language that cannot be pumped. This typically involves choosing a specific string in the language and showing that no matter how it is divided into three parts, pumping it will result in a string that is not in the language.

There are many languages that are not regular and therefore cannot be generated by regular grammars. Some examples of non-regular languages include:

1. The language {a^n b^n c^n | n ≥ 0}, which consists of all strings of the form a^n b^n c^n. This language is not regular because it cannot be recognized by a finite automaton.
2. The language {ww | w ∈ {0, 1}*}, which consists of all strings that are of the form ww, where w is any string of 0s and 1s. This language is not regular because it requires counting the length of the first half of the string and comparing it to the second half, which cannot be done by a finite automaton.
3. The language {a^n b^m | n ≠ m}, which consists of all strings of the form a^n b^m where n is not equal to m. This language is not regular because it requires comparing the number of occurrences of two different symbols, which cannot be done by a finite automaton.

To generate these non-regular languages, one needs to use more powerful grammar formalisms, such as context-free grammars or context-sensitive grammars. These grammars allow for more complex rules that enable the generation of non-regular languages.

Context-free grammars are a more powerful grammar formalism than regular grammars. In a context-free grammar, the production rules have the form $A \rightarrow \alpha$, where A is a nonterminal symbol and α is a string of terminals and nonterminals. The left-hand side of the rule specifies the nonterminal symbol that is being replaced, and the right-hand side specifies the replacement string.

Context-free grammars can be used to generate context-free languages, which are a more general class of languages than regular languages. Context-free languages can be recognized by pushdown automata, which are a type of automaton that has a stack for storing information.

Some examples of context-free languages include:

1. The language of balanced parentheses, which consists of all strings of parentheses that are properly balanced. This language is generated by the context-free grammar $S \rightarrow (S)S \mid \varepsilon$.
2. The language of arithmetic expressions, which consists of all valid expressions in a programming language. This language is generated by a context-free grammar that includes rules for operators, parentheses, and operands.
3. The language of nested if-then-else statements, which consists of all valid nested if-then-else statements in a programming language. This language is generated by a context-free grammar that includes rules for if-then-else statements, conditions, and statements.

Context-sensitive grammars are even more powerful than context-free grammars. In a context-sensitive grammar, the production rules have the form $\alpha A \beta \rightarrow \alpha \gamma \beta$, where A is a nonterminal symbol, α and β are strings of terminals and non-terminals, and γ is a nonempty string. The left-hand side of the rule specifies a context in which the nonterminal symbol A can be replaced, and the right-hand side specifies the replacement string.

Context-sensitive grammars can be used to generate context-sensitive languages, which are an even more general class of languages than context-free languages. Context-sensitive languages can be recognized by linear-bounded automata, which are a type of automaton that has a tape that is bounded by a linear function of the input length.

In summary, regular grammars can generate regular languages, while context-free grammars can generate context-free languages, and context-sensitive grammars can generate context-sensitive languages. Non-regular languages require more powerful grammar formalisms, such as context-free or context-sensitive grammars.

Exercise 2

1. Which of the following is a regular language?

A. {a^n b^n c^n | n ≥ 0}
B. {ww | w ∈ {0, 1}*}
C. {a^n b^m | n ≠ m}
D. {0, 1, 00, 01, 10, 11, 000, 001, ...}

2. Which of the following is a basic operator used in regular expressions?

A. Union
B. Intersection
C. Complementation
D. Concatenation

3. Which of the following statements is true about the connection between regular expression and regular languages?

A. Every regular language can be generated by a regular expression.
B. Every regular expression can be recognized by a finite automaton.
C. Every context-free language can be generated by a regular expression.
D. Every non-regular language can be generated by a regular expression.

4. Which of the following is a production rule in a regular grammar?

A. $A \rightarrow \alpha B \beta$
B. $A \rightarrow \alpha \mid \beta$
C. $S \rightarrow \varepsilon$
D. $S \rightarrow AB$

5. Which of the following statements is true about the pumping lemma for regular languages?

A. It is used to prove that every language is regular.
B. It states that every regular language can be recognized by a pushdown automaton.
C. It can be used to prove that a language is not regular.
D. It applies only to context-free languages.

6. Which of the following languages is not regular?

A. {a^n b^n | n ≥ 0}
B. {a^n b^m c^n | n, m ≥ 0}
C. {ww^R | w ∈ {0, 1}*}
D. {a^n b^n c^n d^n | n ≥ 0}

7. Which of the following grammar formalisms generates context-free languages?

A. Regular grammars
B. Context-free grammars
C. Context-sensitive grammars

D. Turing machines

8. Which of the following is a basic property of regular languages?

A. Closure under intersection
B. Closure under complementation
C. Closure under Kleene star
D. Closure under concatenation of any number of languages

9. Which of the following is a non-regular language?

A. {a^n b^n | n ≥ 0}
B. {a^n b^n c^n | n ≥ 0}
C. {ww | w ∈ {0, 1}*}
D. {a^n b^m | n = m}

10. Which of the following is a consequence of the pumping lemma for regular languages?

A. Every regular language can be recognized by a pushdown automaton.
B. Every regular language can be generated by a context-free grammar.
C. Every non-regular language can be recognized by a Turing machine.
D. Every regular language has an infinite number of strings.

Chapter 3

Context free languages

In automata theory and complexity theory, context-free languages are an important class of formal languages that can be generated by context-free grammars. These languages are a more general class of languages than regular languages and can be recognized by pushdown automata, which are a type of automaton that has a stack for storing information.

A context-free grammar is a set of production rules that can be used to generate a context-free language. The rules have the form $A \rightarrow \alpha$, where A is a nonterminal symbol and α is a string of terminals and non-terminals. The left-hand side of the rule specifies the nonterminal symbol that is being replaced, and the right-hand side specifies the replacement string.

Context-free languages have many important properties and applications in computer science. Some of these are:

1. Parsing: Context-free languages are used to describe the syntax of programming languages and other formal languages, and can be used to parse (i.e., analyze and understand) such languages. This is done using techniques such as top-down parsing and bottom-up parsing.
2. Compiler design: Context-free languages are used in the design and implementation of compilers, which are programs that translate source code written in a high-level language into machine code that can be executed by a computer.
3. Natural language processing: Context-free grammars can be used to model the structure of natural language sentences, and can be used in applications such as speech recognition, machine translation, and text analysis.
4. Formal language theory: Context-free languages are a fundamental concept in the study of formal languages and automata theory, and have many important theoretical properties and applications.

In complexity theory, context-free languages are classified according to their computational complexity, which is a measure of the resources required to recognize or generate the language. Context-free languages can be either deterministic or nondeterministic, and can be further classified as being in one of several complexity classes, such as P, NP, or PSPACE.

In summary, context-free languages are an important class of formal languages that can be generated by context-free grammars and recognized by pushdown automata. They have many important applications in computer science, and are classified according to their computational complexity in complexity theory.

Parsing and ambiguity

In automata theory and complexity theory, parsing is the process of analyzing a string of symbols according to the rules of a formal grammar. The goal of parsing is to determine whether the string belongs to the language generated by the grammar, and if so, to identify the structure of the string according to the grammar's rules.

Parsing is an important process in many areas of computer science, including compiler design, natural language processing, and computer vision. In compiler design, parsing is used to analyze the syntax of a program written in a high-level language, and to translate it into an equivalent program in a lower-level language. In natural language processing, parsing is used to analyze the structure of sentences in a natural language, and to extract meaning from them.

Ambiguity is a common problem in parsing, and occurs when a string can be parsed in more than one way according to the grammar's rules. For example, consider the grammar:

S → aSb | ab

This grammar generates the language consisting of all strings of the form a^n b^n or ab, where n is a nonnegative integer. However, the string "aabb" can be parsed in two different ways:

S → aSb → aaSbb → aabb

S → ab

This ambiguity can lead to problems in parsing, such as incorrect interpretations of program code or natural language sentences.

One way to resolve ambiguity in parsing is to use a technique called disambiguation. Disambiguation involves adding additional rules or constraints to the grammar to eliminate the ambiguity. For example, in natural language processing, disambiguation may involve using additional contextual information, such as the meaning of surrounding words, to determine the correct interpretation of a sentence.

Another way to resolve ambiguity is to use a parsing algorithm that is guaranteed to produce a unique parse tree for any string in the language. One such algorithm is the CYK algorithm, which uses dynamic programming to parse a string in O(n^3) time, where n is the length of the string. The CYK algorithm can be used to parse any context-free language, and can be extended to handle ambiguous grammars by producing all possible parse trees for a given string.

In summary, parsing is the process of analyzing a string according to the rules of a formal grammar. Ambiguity is a common problem in parsing, and can be resolved using techniques such as disambiguation or unique parsing algorithms. Parsing has many important applications in computer science, including compiler design, natural language processing, and computer vision.

Sentential forms

In automata theory, a sentential form is a string of symbols that can be derived from the start symbol of a grammar by applying a sequence of production rules. A sentential form may contain both terminal and nonterminal symbols, and may be used to represent a possible sequence of symbols that can be generated by the grammar.

For example, consider the following context-free grammar:

S → aSb | ε

Starting with the start symbol S, we can apply the first rule to obtain the sentential form aSb, and then apply the second rule to obtain the sentential form ab. Thus, ab is a valid sentential form of this grammar.

Sentential forms are important in the process of parsing a string according to a grammar. In bottom-up parsing, for example, the parser builds the sentential form of the input string from the bottom up, by repeatedly applying the production rules of the grammar to the input symbols. In top-down parsing, the parser starts with the start symbol of the grammar and applies the production rules in a left-to-right manner until it derives the input string.

Let's consider an example of generating sentential forms from a context-free grammar. Suppose we have the following grammar:

$S \rightarrow A \mid B$

$A \rightarrow aA \mid a$

$B \rightarrow bB \mid b$

Starting with the start symbol S, we can generate a number of different sentential forms by repeatedly applying the production rules of the grammar. For example:

1. $S \rightarrow A \rightarrow aA \rightarrow aaA \rightarrow aaa$
2. $S \rightarrow A \rightarrow aA \rightarrow aaA \rightarrow aabB \rightarrow aabbB \rightarrow aabbbB \rightarrow aabbbb$
3. $S \rightarrow B \rightarrow bB \rightarrow bbB \rightarrow bbb$

Each of these is a valid sentential form of the grammar, and represents a possible sequence of symbols that can be generated by the grammar.

In the first example, we apply the production rules $S \rightarrow A$ and $A \rightarrow aA$ repeatedly until we reach the sentential form aaa, which consists entirely of terminal symbols.

In the second example, we apply the production rules $S \rightarrow A$ and $A \rightarrow aA$ repeatedly to generate the prefix aaa, and then apply the production rules $A \rightarrow a$ and $B \rightarrow bB$ repeatedly to generate the suffix aabbbb. Note that we can also generate the same sentential form using the production rules $S \rightarrow B$ and $B \rightarrow bB$ repeatedly.

In the third example, we apply the production rules $S \rightarrow B$ and $B \rightarrow bB$ repeatedly to generate the sentential form bbb, which consists entirely of terminal symbols.

In addition to being used in parsing, sentential forms have many other important applications in automata theory and computer science. For example, they can be used to represent the behavior of a program or system, and to analyze its properties.

Sentential forms can also be used to study the properties of context-free grammars, such as their ambiguity and their ability to generate languages with certain properties. In particular, the number of distinct sentential forms that can be derived from a grammar can be used to determine its complexity and its ability to generate a large number of distinct strings.

In summary, a sentential form is a string of symbols that can be derived from the start symbol of a grammar by applying a sequence of production rules. Sentential forms are important in parsing, program analysis, and the study of context-free grammars.

Derivation tree or parse tree

In automata theory, a derivation tree (also known as a parse tree or syntax tree) is a graphical representation of the process of deriving a string from the start symbol of a grammar by applying a sequence of production rules. The nodes of the tree represent the nonterminal symbols in the derivation, while the edges represent the production rules used to derive the symbols.

Derivation trees are important in the process of parsing a string according to a grammar, as they provide a way to visualize the structure of the string and the way in which it was derived from the grammar. In bottom-up parsing, for example, the parser builds the derivation tree of the input string from the bottom up, starting with the input symbols and working backwards through the production rules of the grammar. In top-down parsing, the parser builds the derivation tree of the input string from the top down, starting with the start symbol of the grammar and applying the production rules in a left-to-right manner until it derives the input string.

Derivation trees can also be used to study the properties of context-free grammars, such as their ambiguity and their ability to generate languages with certain properties. In particular, the structure of the derivation tree can be used to determine the complexity of the grammar and its ability to generate a large number of distinct strings.

The following is an example of a derivation tree for the string "aabb" generated by the grammar:

$$S \rightarrow aSb \rightarrow aaSbb \rightarrow aabBbb \rightarrow aabb$$

In this tree, the nonterminal symbols are represented by the nodes, and the edges represent the production rules used to derive the symbols. The root node represents the start symbol S, and the leaf nodes represent the terminal symbols of the string. The tree shows the sequence of production rules used to derive the string "aabb" from the start symbol S.

Let's consider an example of a derivation tree for a string generated by a context-free grammar. Suppose we have the following grammar:

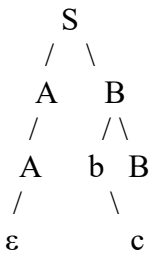
$$S \rightarrow AB$$
$$A \rightarrow aA \mid \varepsilon$$
$$B \rightarrow bB \mid c$$

We will generate a derivation tree for the string "aacbb" using this grammar. Starting with the start symbol S, we can use the production rules of the grammar to derive the string as follows:

$$S \rightarrow AB \rightarrow aAB \rightarrow aaAB \rightarrow aacBb \rightarrow aacbBb \rightarrow aacbb$$

To represent this derivation as a tree, we start with a single node representing the start symbol S. We then add children nodes for each nonterminal symbol in the derivation, and connect them with

edges representing the production rules used to generate them. The resulting derivation tree for the string "aacbb" is shown below:



In this tree, the root node represents the start symbol S, and has two children representing the nonterminal symbols A and B. The node for A has two children representing the nonterminal symbol A and the empty string ε, while the node for B has two children representing the terminal symbols b and c. The leaf nodes of the tree represent the individual symbols of the string "aacbb", with each symbol appearing as a child of a nonterminal symbol that generates it.

The derivation tree provides a visual representation of the process of deriving the string "aacbb" from the start symbol S, and allows us to see the structure of the string according to the grammar's rules.

Left most and right most derivations

In automata theory, a leftmost derivation and a rightmost derivation are two types of derivations that can be used to generate a string from a context-free grammar. Both types of derivations use the production rules of the grammar to replace nonterminal symbols with sequences of terminal and nonterminal symbols, but they differ in the order in which the replacements are made.

A leftmost derivation is a derivation in which the leftmost nonterminal symbol in the current sentential form is always replaced in each step. That is, at each step of the derivation, the leftmost nonterminal symbol in the current sentential form is replaced by a sequence of terminal and nonterminal symbols according to a production rule.

Let's consider an example of leftmost and rightmost derivations for a string generated by a context-free grammar. Suppose we have the following grammar:

$S \rightarrow AB$

$A \rightarrow aA \mid \varepsilon$

$B \rightarrow bB \mid c$

We will use a leftmost derivation and a rightmost derivation to generate the string "aacbb" using this grammar.

Leftmost derivation:

$S \rightarrow AB \rightarrow aAB \rightarrow aaAB \rightarrow aacBb \rightarrow aacbBb \rightarrow aacbb$

In each step of the leftmost derivation, we replace the leftmost nonterminal symbol in the current sentential form with a sequence of terminal and nonterminal symbols according to a production rule.

Rightmost derivation:
S → AB → ABb → ACbb → Aacbb → aABCbb → aaACbbb → aacBbbb → aacbBb → aacbb

In each step of the rightmost derivation, we replace the rightmost nonterminal symbol in the current sentential form with a sequence of terminal and nonterminal symbols according to a production rule.

Note: that while the leftmost derivation and the rightmost derivation generate the same string, they use different sequences of production rules and generate different intermediate sentential forms. The leftmost derivation generates sentential forms in a left-to-right manner, while the rightmost derivation generates sentential forms in a right-to-left manner.

Both leftmost and rightmost derivations can be used in parsing a string according to a grammar. In top-down parsing, for example, the parser uses a leftmost derivation to generate the sentential form of the input string, while in bottom-up parsing, the parser uses a rightmost derivation. The choice of derivation depends on the parsing algorithm used and the properties of the grammar being parsed.

In summary, a derivation tree (or parse tree or syntax tree) is a graphical representation of the process of deriving a string from the start symbol of a grammar by applying a sequence of production rules. Derivation trees are important in parsing and in the study of context-free grammars.

Simplification of context free grammar

In automata theory, simplification of a context-free grammar involves removing unnecessary symbols and production rules from the grammar without changing the language it generates. Simplification can make a grammar easier to understand and analyze, and can also make it easier to implement a parser for the grammar.

There are several techniques that can be used to simplify a context-free grammar:

1. Removing unreachable symbols: Symbols that cannot be reached from the start symbol of the grammar should be removed. This includes nonterminal symbols that do not appear in any derivation of a terminal string.
2. Removing unproductive symbols: Symbols that cannot generate any terminal string should be removed. This includes nonterminal symbols that do not appear in any derivation of a terminal string, as well as production rules that do not contribute to the generation of any terminal string.
3. Removing useless productions: Productions that do not contribute to the generation of any terminal string should be removed. This includes productions that have a nonterminal symbol on the left-hand side that does not appear in any derivation of a terminal string, as well as productions that have a nonterminal symbol on the right-hand side that cannot generate any terminal string.
4. Removing epsilon productions: Epsilon productions are productions of the form A → ε, where A is a nonterminal symbol. They can be removed by replacing each occurrence of A

in a production with the right-hand side of the production, except when A appears at the beginning of a production.

5. Removing unit productions: Unit productions are productions of the form A → B, where A and B are nonterminal symbols. They can be removed by replacing each occurrence of A in a production with the right-hand side of a production that has B on the left-hand side.

By applying these techniques, a context-free grammar can be simplified while preserving the language it generates. This can make the grammar easier to work with and can help to identify properties of the language it generates.

Let's continue with an **example** of simplifying a context-free grammar. Suppose we have the following grammar:

S → AaB | ε
A → B | a
B → Ab | ε

We can simplify this grammar using the following steps:

1. Removing unreachable symbols: Since S is the start symbol, all symbols are reachable from it.
2. Removing unproductive symbols: The nonterminal symbol A is unproductive since it does not appear in any derivation of a terminal string. We can remove it from the grammar.
3. Removing useless productions: The production A → B is useless since B can generate ε, and thus A can be replaced by ε. We can remove this production from the grammar.
4. Removing epsilon productions: The grammar has two epsilon productions: S → ε and B → ε. We can remove them by replacing each occurrence of the nonterminal symbol with the right-hand side of the production, except when the nonterminal symbol appears at the beginning of a production. Thus, we get:

   S → AaB | AB | aB | a
   A → B | a
   B → Ab

5. Removing unit productions: There are no unit productions in this grammar.

The resulting simplified grammar is:

S → AaB | AB | aB | a
A → Ab | a
B → Ab

This grammar generates the same language as the original grammar, but is simpler and easier to understand.

Methods for transforming grammars

In automata theory, there are several methods for transforming grammars that can be used to generate equivalent grammars with different properties. These methods include:

1. Elimination of left recursion: A grammar is said to be left-recursive if it has a nonterminal symbol A that can derive a string starting with A itself. For example, the grammar A → Ab | c is left-recursive because it can derive the string Ab, which starts with A. Left recursion can cause problems in parsing and can be eliminated by rewriting the grammar to remove the left-recursive productions.

2. Factoring: A grammar is said to be factored if it has a nonterminal symbol A that can derive two or more distinct strings that start with the same symbol. For example, the grammar A → aB | aC can be factored as A → aD, D → B | C. Factoring can make a grammar more efficient to parse and can reduce the size of the grammar.

3. Chomsky normal form: A grammar is said to be in Chomsky normal form if all of its productions are either of the form A → BC or A → a, where A, B, and C are nonterminal symbols and a is a terminal symbol. Any context-free grammar can be transformed into an equivalent grammar in Chomsky normal form.

4. Greibach normal form: A grammar is said to be in Greibach normal form if all of its productions are of the form A → aα, where A is a nonterminal symbol, a is a terminal symbol, and α is a string of nonterminal symbols. Any context-free grammar can be transformed into an equivalent grammar in Greibach normal form.

These methods can be used to simplify and optimize context-free grammars, to make them easier to parse or to analyze. The choice of method depends on the properties of the grammar and the requirements of the parsing algorithm being used.

Let's continue with an example of transforming a grammar. Suppose we have the following grammar:

S → Sa | Sb | a

This grammar is left-recursive, which can cause problems in parsing. We can eliminate left recursion by introducing a new nonterminal symbol S' and rewriting the grammar as follows:

S → aS'

S' → aS' | bS' | ε

The resulting grammar is equivalent to the original grammar but does not have any left-recursive productions.

Now let's consider an example of transforming a grammar to Chomsky normal form. Suppose we have the following grammar:

S → AB | BC | a | b

A → BA | a

B → CC | b

C → AB | a

To transform this grammar into Chomsky normal form, we can follow these steps:

1. Introduce a new start symbol S' and add a production S' → S.
2. Eliminate the ε-production by adding new productions for each nonterminal symbol that can derive ε. In this case, C can derive ε, so we add the productions C → ε and A → ε.
3. Replace each production that generates a single terminal symbol with a new production that generates that terminal symbol. In this case, we replace the production S → a with a new production S → A, and the production S → b with a new production S → B.
4. Replace each production that generates a string of length greater than 2 with a sequence of productions that generate only two nonterminal symbols. In this case, we replace the production S → AB with the productions S → XY, X → AB, and Y → BC.

The resulting grammar in Chomsky normal form is:

S' → S

S → XY | A | B

X → AB

Y → BC

A → BA | a | ε

B → CC | b

C → AB | a | ε

This grammar is now in Chomsky normal form and can be useful for analyzing and parsing the language it generates.

Let's continue with an example of transforming a grammar to Greibach normal form. Suppose we have the following grammar:

S → aSb | ε

To transform this grammar into Greibach normal form, we can follow these steps:

1. Introduce a new start symbol S' and add a production S' → S.

2. Replace each occurrence of a terminal symbol in a production with a new nonterminal symbol that generates that terminal symbol. In this case, we replace a with A and b with B.

3. Replace each production that generates a string of length greater than 1 with a sequence of productions that generate a string of length 1. In this case, we replace the production S → aSb with the productions S → AS' and S' → SB.

4. Rewrite each production so that the right-hand side starts with a terminal symbol. In this case, we rewrite the production S' → SB as S' → BS.

5. Remove any ε-productions that generate the empty string. In this case, there are no ε-productions.

6. Remove any unit productions that have a single nonterminal symbol on the right-hand side. In this case, there are no unit productions.

The resulting grammar in Greibach normal form is:

$S' \rightarrow S$

$S \rightarrow AS'$

$S' \rightarrow BS$

$A \rightarrow a$

$B \rightarrow b$

This grammar is now in Greibach normal form and can be useful for analyzing and parsing the language it generates.

Chomsky's hierarchy of grammars

Chomsky's hierarchy of grammars is a classification of formal grammars into four types, based on the types of production rules used in the grammar. The hierarchy was proposed by Noam Chomsky in the 1950s and has become an important concept in the study of formal languages and automata theory.

The four types of grammars in Chomsky's hierarchy are:

1. Type-0 (unrestricted or phrase-structure grammars): These grammars have no restrictions on the form of their production rules. They can generate any language that can be recognized by a Turing machine, including non-context-free languages.

2. Type-1 (context-sensitive grammars): These grammars have production rules of the form $\alpha A\beta \rightarrow \alpha\gamma\beta$, where A is a nonterminal symbol, α and β are strings of terminal and nonterminal symbols, and γ is a nonempty string of symbols. They can generate all context-sensitive languages.

3. Type-2 (context-free grammars): These grammars have production rules of the form $A \rightarrow \alpha$, where A is a nonterminal symbol and α is a string of terminal and nonterminal symbols. They can generate all context-free languages.

4. Type-3 (regular grammars): These grammars have production rules of the form $A \rightarrow aB$ or $A \rightarrow a$, where A and B are nonterminal symbols, a is a terminal symbol, and ε is allowed as a right-hand side. They can generate all regular languages.

The grammars in each type of Chomsky's hierarchy are a strict subset of the grammars in the higher type. In other words, a grammar in a higher type can generate a language that a grammar in a lower type cannot generate. The hierarchy provides a way of comparing the expressive power of different types of grammars and the complexity of the languages they can generate.

Chomsky's hierarchy of grammars has been influential in the development of formal language theory and has applications in the design and analysis of programming languages, compilers, and natural language processing systems.

Let's continue with some examples of languages that can be generated by each type of grammar in Chomsky's hierarchy:

1. Type-0: Any language that can be recognized by a Turing machine, including non-context-free languages such as the language {0^n1^n2^n | n ≥ 0}.

2. Type-1: Context-sensitive languages such as the language {0^n1^n2^n3^n | n ≥ 0}.

3. Type-2: Context-free languages such as the language of properly nested parentheses, {w | w is a string of balanced parentheses}.

4. Type-3: Regular languages such as the language of all strings over the alphabet {0, 1} that contain an even number of 0s, and the language of all strings over the alphabet {a, b} that start with an 'a'.

In practice, context-free grammars and regular grammars are the most commonly used types of grammars, and they can be used to generate a wide variety of languages. The Chomsky hierarchy provides a useful framework for understanding the expressive power and limitations of different types of grammars, and for comparing the complexity of different languages.

How to convert CFG to CNF?

Step1: *Eliminate start symbol from RHS.*

If start symbol S is at the RHS of any production in the grammar, create a new production as:

S0 ->S | where S0 is the new start symbol.

Step2: *Eliminate null, unit and useless productions.*

If CFG contains null, unit or useless production rules, eliminate them.

Step3: *Eliminate terminals from RHS if they exist with other terminals or non-terminals*.

E.g. production rule X->xY can be decomposed as: X->ZY, Z->x

Step4: *Eliminate RHS with more than two non-terminals*.

E.g. production rule X->XYZ can be decomposed as: X->PZ, P->XY

Procedure for converting to CNF:

Simplify the CFG | *Eliminate non-terminal*, if more than two non-terminal presented in the production, eliminate it and make new rule

Example: S → ABA

In CNF form: P1 → BA rule 1
S → AP1 rule 2

See the rule 1 and 2, only two non-terminals only available in right of the production.

Eliminate the terminal, if more than one terminal is presented in the production, eliminate it and make new rule

Example: S → aa

In CNF form:
P1 → a rule1
P2 → a rule2
S → P1P2 rule3

Now see the rule1 and 2 are containing only one terminal. Also rule3 is containing two non-terminals only

Example: Convert the given CFG to CNF

S → aSa | bSb | a | b

Write the rules separately
S → aSa rule1
S → bSb rule2
S → a rule3
S → b rule4

Start with adding new symbol for the terminals

A → a new rule1
B → b new rule2

Apply new rule1 in rule1

S → ASA in this rule three non-terminal available so eliminate the non-terminals
S → AP1 new rule3
P1 → SA new rule4

Apply new rule2 in rule2

S → BSB in this rule three non-terminal available so eliminate the non-terminals
S → BP2 new rule5
P2 → SB new rule6

Now consider the rule3 and rule4, these two rules are already in CNF format so no need to change it.

Finally the CFG in CNF

A → a new rule1
B → b new rule2
S → AP1 new rule3
P1 → SA new rule4
S → BP2 new rule5
P2 → SB new rule6
S → a new rule7
S → b new rule8

Exercise 3

1. Which of the following languages is context-free?

a) {a^n b^n c^n d^n | n ≥ 0}
b) {ww^R | w is a string over {0, 1}*}
c) {0^n 1^m 2^k | n = m or m = k}
d) {a^n b^m c^k | n + m = k}

2. Which of the following statements is true about parsing and ambiguity?

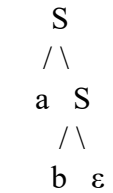
a) Ambiguous grammars always have a unique parse tree.
b) Parsing is the process of converting a parse tree into a string.
c) Ambiguity can be resolved by removing all epsilon productions from a grammar.
d) Ambiguity occurs when a grammar generates more than one parse tree for a given string.

3. Which of the following is a sentential form of the grammar S → aSb | ε?

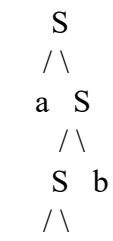
a) S
b) aSb
c) aaSbb
d) aab

4. Which of the following is a parse tree for the string "aabb" generated by the grammar S → aSb | ε?

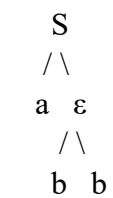
a)



b)



c)



d)

```
   S
  / \
 a   b
    / \
   a   b
```

5. Which of the following is a simplified grammar equivalent to the grammar S → AaB | ε, A → B | a, B → Ab | ε?

a) S → AaB | a, A → a, B → Ab
b) S → AaB | AB | a, A → a, B → Ab
c) S → AaB | AB | a, A → B | a, B → Ab
d) S → AaB | AB | a, A → B, B → Ab

6. Which of the following is not a method for simplifying a context-free grammar?

a) Eliminating left recursion
b) Removing unit productions
c) Factoring out common prefixes
d) Introducing new nonterminal symbols

7. Which type of grammar in Chomsky's hierarchy can generate all context-free languages?

a) Type-0
b) Type-1
c) Type-2
d) Type-3

8. Which of the following is a context-free grammar for the language of arithmetic expressions with addition and multiplication, using the symbols +, *, (, and)?

a) E → E + T | T, T → T * F | F, F → (E) | id
b) E → T + E | T, T → F * T | F, F → (E) | id
c) E → T + E | T, T → F T' | F, T' → * F T' | ε, F → (E) | id
d) E → E + T | T, T → F * T | F, F → (E) | id

9. Which of the following statements is true about a pushdown automaton?

a) It accepts all context-sensitive languages.
b) Its input tape is infinite in both directions.
c) It has a finite control and an unbounded stack.
d) It can recognize non-context-free languages.

10. Which of the following is a parsing algorithm that uses a bottom-up strategy?

a) LL parser
b) LR parser
c) Recursive descent parser

d) Top-down parser

11. Which of the following is a language that is not context-free?

a) {a^n b^n c^n | n ≥ 0}
b) {0^n 1^m 0^n | n,m ≥ 0}
c) {ww | w is a string over {0, 1}*}
d) {a^n b^m c^k | n + m = k}

12. Which of the following statements about parse trees is false?

a) A parse tree represents the derivation of a string from a grammar.
b) The leaves of a parse tree correspond to the terminals in the string being parsed.
c) A parse tree can be used to determine if a string is in the language generated by a grammar.
d) A parse tree can be uniquely constructed for any string generated by a grammar.

13. Which of the following is not a step in simplifying a context-free grammar?

a) Eliminating epsilon productions
b) Removing useless symbols
c) Converting the grammar to regular form
d) Eliminating immediate left recursion

14. Which of the following is a context-free grammar for the language of all palindromes over the alphabet {0, 1}?

a) S → 0S0 | 1S1 | ε
b) S → 0S | 1S | S0 | S1 | ε
c) S → 0S0 | 1S1 | 0 | 1 | ε
d) S → 0S | 1S | 0 | 1 | ε

15. Which of the following is an example of an ambiguous grammar?

a) S → aSb | bSa | ε
b) S → AB, A → a, B → b
c) S → aSbS | bSaS | ε, S → SSS
d) S → AB | BA, A → aB | ε, B → bA | ε

16. Which of the following parsing algorithms uses a top-down strategy?

a) LL parser
b) LR parser
c) Earley parser
d) CYK parser

17. Which of the following is a context-free grammar for the language of all strings over the alphabet {a, b} that contain at least one 'a'?

a) S → aS | bS | ε
b) S → A | B, A → aB | a, B → bB | ε

c) S → aA | bB, A → aA | ε, B → bB | ε
d) S → aA | B, A → aA | ε, B → bB | ε

18. Which of the following statements about context-free grammars is false?

a) Every regular language is a context-free language.
b) Every context-free language is generated by a context-free grammar.
c) Every context-free language is also a regular language.
d) Two context-free grammars can generate the same language.

19. Which of the following is a language that can be generated by a context-sensitive grammar but not by a context-free grammar?

a) {a^n b^n c^n | n ≥ 0}
b) {0^n 1^m 0^n | n,m ≥ 0}
c) {ww | w is a string over {0, 1}*}
d) {a^n b^m c^k | n + m = k}

20. Which of the following is a simplification of the context-free grammar S → aSb | aS | ε?

a) S → aS | ε
b) S → aSb | aS
c) S → aSb | aS | b
d) S → aSb | aS | bS

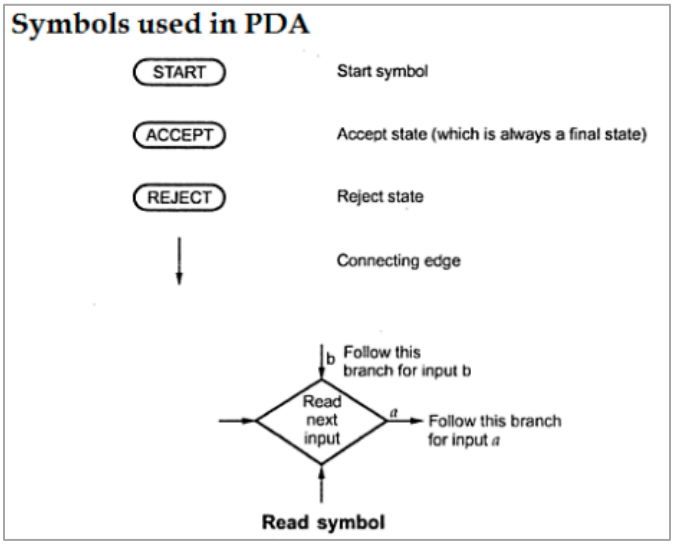
Chapter 4

Pushdown Automata

A pushdown automaton (PDA) is a type of automaton, which is a mathematical model for computation. PDAs are similar to finite state machines (FSMs) but also have a stack, which allows them to recognize context-free languages.

At a high level, a PDA consists of:

1. An input tape, which is read from left to right.
2. A stack, which can hold a finite number of symbols.
3. A finite set of states.
4. A start state.
5. A set of accept states.
6. A set of transitions, which define how the PDA moves between states based on the input symbol and the top symbol of the stack.



During operation, the PDA reads symbols from the input tape and uses the stack to keep track of information about the input that it has seen so far. The PDA can push symbols onto the stack, pop symbols off the stack, or leave the stack unchanged. The stack allows the PDA to keep track of context or nesting, which makes it more powerful than a finite state machine.

A PDA can accept or reject an input string based on whether it can reach an accept state after reading the entire input string. If it reaches an accept state, the PDA accepts the string; otherwise, it rejects the string.

A PDA can recognize a language if there exists a sequence of transitions that leads it from the start state to an accept state when reading any string in the language. The set of languages that can be recognized by PDAs is called the class of context-free languages.

PDAs can be modeled in different ways, including using a transition table or a transition diagram. One common way to represent a PDA is using a state-transition table, which shows the possible transitions the PDA can take based on the input symbol and the top symbol of the stack. Another way to represent a PDA is using a state-transition diagram, which is a graphical representation of the PDA's states and transitions.

PDAs have a close relationship with context-free grammars (CFGs). In fact, every CFG can be transformed into an equivalent PDA, and vice versa. This relationship is known as the Chomsky–Schützenberger theorem.

PDAs have many practical applications, especially in the field of programming language theory and compiler design. They are used to parse and analyze programming languages, and to ensure

that the syntax of the language is correct. Additionally, PDAs are used in natural language processing to analyze and understand the structure of sentences in a language.

PDAs have several variants, including non-deterministic PDAs (NPDA) and deterministic PDAs (DPDA). In an NPDA, there can be multiple possible transitions from a given state and input symbol combination, while in a DPDA, there is at most one possible transition from a given state and input symbol combination. NPDAs are more powerful than DPDAs, but they are also more difficult to design and analyze.

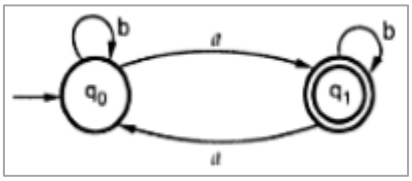
Another variant of PDAs is the two-way PDA, which can read the input tape in both directions, allowing it to recognize languages that are not context-free, such as palindromes.

Finally, it is worth noting that PDAs are a specific type of automaton, and there are other types of automata, such as Turing machines, which are even more powerful and can recognize more languages. Turing machines are equivalent in power to the class of recursive languages, which includes all computable languages.

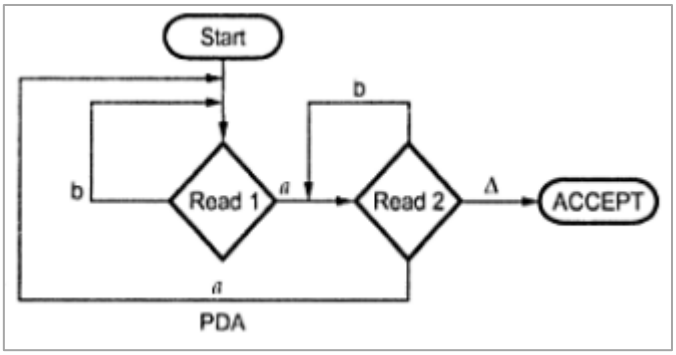
Overall, PDAs are an important concept in computer science and theoretical computer science, and they provide a useful framework for studying the properties of context-free languages and their relationship to other classes of languages.

Example: Design PDA which accepts only odd number of a's over Σ = {a, b}

Draw the Finite State automata for the above problem from that you can draw the PDA easily



=>



Non-deterministic pushdown automata

Non-deterministic pushdown automata (NPDAs) are a variant of pushdown automata (PDAs) that allow for multiple possible transitions from a given state and input symbol combination. In an NPDA, the next transition is not uniquely determined by the current state and input symbol. Instead, the NPDA can guess which transition to take, and then it can verify whether the guess was correct by simulating all possible paths.

Formally, an NPDA is defined as a 7-tuple (Q, Σ, Γ, δ, q0, Z, F), where:

- Q is a finite set of states.
- Σ is a finite input alphabet.
- Γ is a finite stack alphabet.

- δ is a transition function that maps Q × (Σ ∪ {ε}) × Γ to 2^(Q × Γ*), where ε represents an empty string and Γ* represents the set of all possible strings over the stack alphabet Γ.
- q0 is the initial state.
- Z is the initial stack symbol.
- F is a set of final states.

The transition function δ takes as input the current state, the next input symbol or an empty string (ε), and the top symbol of the stack, and returns a set of possible next states and stack symbols. If there are no possible next states and stack symbols for the given input, the transition function returns the empty set.

The NPDA accepts an input string w if there exists a sequence of transitions that leads it from the initial state and the initial stack symbol Z to a final state with an empty stack, when reading the input string w. In other words, the NPDA can push symbols onto the stack, pop symbols off the stack, or leave the stack unchanged, and it can do so in any order.

NPDAs are more powerful than deterministic pushdown automata (DPDAs), since they can recognize more languages. However, this increased power comes at a cost, since the non-determinism in NPDAs makes them more difficult to design and analyze.

The computational complexity of NPDAs is also more difficult to analyze than that of DPDAs, since the number of possible paths that the NPDA can take can grow exponentially with the length of the input string. As a result, the time and space complexity of an NPDA can be difficult to predict.

NPDAs can be modeled using transition diagrams or tables, similar to DPDAs. However, in an NPDA, there can be multiple arrows leaving a state with the same input symbol and stack symbol. These multiple arrows represent the non-determinism in the NPDA, since the NPDA can choose any of these arrows to transition to the next state.

NPDAs are useful in many applications, such as parsing context-free languages, analyzing natural language processing, and modeling cellular processes in biology. They are also used in the design of compilers and interpreters for programming languages.

One disadvantage of NPDAs is that they are more difficult to implement in hardware than DPDAs, since the non-determinism in NPDAs requires more complex circuitry. However, this is not usually a significant problem in practice, since most applications of NPDAs are in software rather than hardware.

There are several algorithms for simulating an NPDA, one of which is the subset construction algorithm. The subset construction algorithm is a standard method for converting an NPDA into a deterministic pushdown automaton (DPDA).

The subset construction algorithm works by constructing a set of states for the DPDA that corresponds to each possible subset of states for the NPDA. The transitions for the DPDA are then defined based on the transitions for the NPDA and the current subset of states.

The resulting DPDA simulates the behavior of the original NPDA, but with a potentially larger number of states. However, the DPDA is guaranteed to recognize the same language as the original NPDA, since deterministic automata are a subset of non-deterministic automata.

In summary, non-deterministic pushdown automata are a variant of pushdown automata that allow for multiple possible transitions from a given state and input symbol combination. NPDAs are more powerful than deterministic pushdown automata, but they are also more difficult to design and analyze. The subset construction algorithm can be used to convert an NPDA into a deterministic pushdown automaton.

Let's consider an example of a non-deterministic pushdown automaton that recognizes the language {0^n1^n | n ≥ 0}, which consists of all strings of 0's followed by an equal number of 1's.

The NPDA for this language can be defined as follows:

- Q = {q0, q1, q2}
- Σ = {0, 1}
- Γ = {Z, A}
- δ is defined as follows:
    - δ(q0, 0, Z) = {(q0, AZ)}
    - δ(q0, 0, A) = {(q0, AA)}
    - δ(q0, 1, A) = {(q1, ε)}
    - δ(q1, 1, A) = {(q1, ε)}
    - δ(q1, ε, Z) = {(q2, Z)}
- q0 is the initial state.
- Z is the initial stack symbol.
- F = {q2} is the set of final states.

In this NPDA, the stack initially contains only the symbol Z. Whenever the NPDA reads a 0 from the input, it pushes an A onto the stack. Whenever it reads a 1 from the input, it pops an A from the stack. If the input is valid, the stack will be empty when the NPDA reaches the final state q2.

To see how this NPDA works, let's consider the input string 0011. The NPDA starts in state q0 with the stack containing Z. It reads the first symbol 0 from the input and pushes an A onto the stack, so the stack now contains ZA. The NPDA reads the second symbol 0 from the input and pushes another A onto the stack, so the stack now contains ZAA. The NPDA reads the third symbol 1 from the input and pops an A from the stack, so the stack now contains ZA. The NPDA reads the fourth symbol 1 from the input and pops another A from the stack, so the stack now contains Z. Finally, the NPDA reaches the final state q2 with an empty stack, so it accepts the input string.

If we consider the input string 000111, the NPDA would also accept it in a similar manner. However, if we consider the input string 00110, the NPDA would reject it since there would be an extra 0 in the input with no corresponding 1 to match it.

It's worth noting that this NPDA is non-deterministic because there are multiple possible transitions for some of the input symbols and stack symbols. For example, when the NPDA is in state q0 and reads a 0 with a top-of-stack symbol of A, there are two possible transitions it can take: either it

can stay in state q0 and push another A onto the stack, or it can move to state q1 and pop an A from the stack. The non-determinism allows the NPDA to explore multiple possible paths through the transition graph in order to recognize the language.

To convert this NPDA into a deterministic pushdown automaton (DPDA), we can use the subset construction algorithm. The resulting DPDA would have a larger number of states than the original NPDA, but it would recognize the same language. The subset construction algorithm works by constructing a set of states for the DPDA that corresponds to each possible subset of states for the NPDA. The transitions for the DPDA are then defined based on the transitions for the NPDA and the current subset of states.

Sure! Here's a diagram of the non-deterministic pushdown automaton we considered for the language {0^n1^n | n ≥ 0}:

0, Z → AZ
0, A → AA
δ(q0, Z) 1, A → ε
1, A → ε
ε, Z → Z

1, Z → reject
δ(q1, A) 1, A → ε

ε, Z → accept
δ(q2, Z) ε, A → ε

In this diagram, the states are represented by circles labeled q0, q1, and q2. The arrows represent transitions between states, and they are labeled with the input symbol and the top symbol of the stack that cause the transition. The symbol ε represents an empty string on the input or the stack. The accept state is represented by a double circle around the state q2.

Note that there are multiple arrows leaving some of the states, indicating the non-determinism in the NPDA. For example, from state q0, there are two arrows labeled 0, A that lead to different states. This means that the NPDA can choose either of these arrows to transition to a new state, depending on the current input and stack symbol.

In summary, the non-deterministic pushdown automaton we considered is a simple example of an NPDA that recognizes a context-free language. The NPDA uses a stack to keep track of the number of 0's and 1's in the input, and the non-determinism allows it to explore multiple possible paths through the transition graph in order to recognize the language. The subset construction algorithm can be used to convert this NPDA into a deterministic pushdown automaton that recognizes the same language.

Push down automata and context free languages

Pushdown automata (PDA) are a powerful model of computation that are used to recognize context-free languages. Context-free languages are a class of formal languages that can be generated by context-free grammars.

A context-free language consists of strings of symbols that can be generated by a context-free grammar, which is a set of production rules that describe how to generate the strings. A context-free grammar consists of a set of non-terminal symbols, a set of terminal symbols, a start symbol, and a set of production rules. The production rules define how to replace non-terminal symbols with sequences of terminal and non-terminal symbols.

A pushdown automaton consists of a finite set of states, a stack, an input tape, and a set of transitions. The stack allows the PDA to keep track of context or nesting, which makes it more powerful than a finite state machine. The transitions define how the PDA moves between states based on the input symbol and the top symbol of the stack.

A PDA can recognize a language if there exists a sequence of transitions that leads it from the start state to an accept state when reading any string in the language. If it reaches an accept state, the PDA accepts the string; otherwise, it rejects the string.

Every context-free language can be recognized by a pushdown automaton, and every pushdown automaton can be associated with a context-free grammar. This relationship between pushdown automata and context-free grammars is known as the Chomsky-Schützenberger theorem.

Pushdown automata are an important concept in computer science and theoretical computer science, and they provide a useful framework for studying the properties of context-free languages and their relationship to other classes of languages. They are used in many applications, such as parsing and analyzing programming languages, and in natural language processing to analyze and understand the structure of sentences in a language.

There are several variants of pushdown automata, including deterministic pushdown automata (DPDA) and non-deterministic pushdown automata (NPDA). In a DPDA, there is at most one possible transition from a given state and input symbol combination, while in an NPDA, there can be multiple possible transitions from a given state and input symbol combination. NPDAs are more powerful than DPDAs, but they are also more difficult to design and analyze.

Another variant of pushdown automata is the two-way pushdown automaton, which can read the input tape in both directions, allowing it to recognize languages that are not context-free, such as palindromes.

Pushdown automata have many practical applications, especially in the field of programming language theory and compiler design. They are used to parse and analyze programming languages, and to ensure that the syntax of the language is correct. Additionally, pushdown automata are used in natural language processing to analyze and understand the structure of sentences in a language.

Overall, pushdown automata are an important concept in computer science and theoretical computer science, and they provide a useful framework for studying the properties of context-free languages and their relationship to other classes of languages.

Let's consider an example of a context-free language that can be recognized by a pushdown automaton. Consider the language {a^n b^n | n ≥ 0}, which consists of all strings of the form a^n b^n, where n is a non-negative integer.

We can define a context-free grammar for this language as follows:

- S → ε
- S → a S b

The start symbol is S, and the production rules say that we can replace S with an empty string or with the sequence a S b. This generates strings of the form a^n b^n, where n is a non-negative integer.

We can also define a pushdown automaton that recognizes this language. The PDA consists of a stack, an input tape, and a set of transitions. The stack contains a single symbol, Z, which is the initial stack symbol. The transitions are defined as follows:

- (q0, a, Z) → (q0, aZ)
- (q0, a, A) → (q0, AA)
- (q0, ε, Z) → (q1, Z)
- (q1, b, A) → (q1, ε)

In this PDA, the states are q0 and q1. The symbol A represents a non-terminal symbol, and all other symbols are terminal symbols. The PDA starts in state q0 with the stack containing the symbol Z. Whenever it reads an a from the input tape, it pushes an A onto the stack. Whenever it reads a b from the input tape, it pops an A from the stack. If the input is valid, the stack will be empty when the PDA reaches state q1.

Let's consider the input string aaabb. The PDA starts in state q0 with the stack containing Z. It reads the first symbol a from the input tape and pushes an A onto the stack, so the stack now contains ZA. The PDA reads the second symbol a from the input tape and pushes another A onto the stack, so the stack now contains ZAA. The PDA reads the third symbol a from the input tape and pushes another A onto the stack, so the stack now contains ZAAA. The PDA reads the first b from the input tape and pops an A from the stack, so the stack now contains ZAA. The PDA reads the second b from the input tape and pops another A from the stack, so the stack now contains ZA. Finally, the PDA reaches state q1 with an empty stack, so it accepts the input string.

Thus, the string aaabb is in the language {a^n b^n | n ≥ 0}, and it is recognized by the pushdown automaton we defined.

Deterministic push down automata

Deterministic pushdown automata (DPDA) are a variant of pushdown automata that are similar to non-deterministic pushdown automata (NPDA), but with one key difference: in a DPDA, there is at most one possible transition from a given state and input symbol combination. This makes DPDAs more predictable and easier to analyze than NPDAs.

Formally, a DPDA can be defined as a 7-tuple M = (Q, Σ, Γ, δ, q0, Z, F), where:

- Q is a finite set of states.
- Σ is a finite input alphabet.
- Γ is a finite stack alphabet.
- δ: Q × Σ × Γ → Q × Γ* is a transition function that maps a state, an input symbol, and a stack symbol to a new state and a string of stack symbols to replace the top of the stack.
- q0 ∈ Q is the initial state.
- Z ∈ Γ is the initial stack symbol.
- F ⊆ Q is the set of accepting/final states.

The behavior of a DPDA is similar to that of an NPDA, but with the added constraint that there can be at most one possible transition for each state and input symbol combination. When the DPDA reads an input symbol, it can read the top symbol of the stack and transition to a new state while also pushing, popping, or replacing symbols on the stack.

DPDAs recognize a subset of the languages recognized by NPDAs, but they have several advantages. For example, DPDAs are easier to implement and analyze, and they can be used to parse deterministic context-free languages. Additionally, some algorithms that rely on pushdown automata, such as the CYK algorithm for parsing context-free grammars, can be implemented more efficiently using DPDAs.

One way to construct a DPDA from an NPDA is to use the subset construction algorithm, which constructs a new DPDA that simulates the behavior of the original NPDA by maintaining a set of possible states. The resulting DPDA can recognize the same language as the original NPDA, but it may have a larger number of states.

Overall, deterministic pushdown automata are an important variant of pushdown automata that are useful for recognizing deterministic context-free languages and for implementing algorithms that rely on pushdown automata.

Let's consider an example of a deterministic pushdown automaton (DPDA) that recognizes the language {0^n1^n | n ≥ 0}, which consists of all strings of 0's followed by an equal number of 1's.

The DPDA for this language can be defined as follows:

- Q = {q0, q1, q2}
- Σ = {0, 1}
- Γ = {Z, A}
- δ is defined as follows:
    - δ(q0, 0, Z) = (q0, AZ)

  - δ(q0, 0, A) = (q0, AA)
  - δ(q0, 1, A) = (q1, ε)
  - δ(q1, 1, A) = (q1, ε)
  - δ(q1, ε, Z) = (q2, Z)
- q0 is the initial state.
- Z is the initial stack symbol.
- F = {q2} is the set of final states.

In this DPDA, the stack initially contains only the symbol Z. Whenever the DPDA reads a 0 from the input, it pushes an A onto the stack. Whenever it reads a 1 from the input, it pops an A from the stack. If the input is valid, the stack will be empty when the DPDA reaches the final state q2.

To see how this DPDA works, let's consider the input string 000111. The DPDA starts in state q0 with the stack containing Z. It reads the first symbol 0 from the input tape and pushes an A onto the stack, so the stack now contains ZA. The DPDA reads the second symbol 0 from the input tape and pushes another A onto the stack, so the stack now contains ZAA. The DPDA reads the third symbol 0 from the input tape and pushes another A onto the stack, so the stack now contains ZAAA. The DPDA reads the first symbol 1 from the input tape and pops an A from the stack, so the stack now contains ZAA. The DPDA reads the second symbol 1 from the input tape and pops another A from the stack, so the stack now contains ZA. The DPDA reads the third symbol 1 from the input tape and pops another A from the stack, so the stack is now empty. Finally, the DPDA reaches the final state q2, so it accepts the input string.

Thus, the string 000111 is in the language {0^n1^n | n ≥ 0}, and it is recognized by the DPDA we defined.

Deterministic context free languages

A deterministic context-free language (DCFL) is a context-free language that can be recognized by a deterministic pushdown automaton (DPDA). In other words, a DCFL is a language that can be generated by a context-free grammar and recognized by a deterministic pushdown automaton.

DCFLs are a proper subset of context-free languages (CFLs), which can be recognized by non-deterministic pushdown automata (NPDAs). The fact that not all CFLs are DCFLs means that there are some context-free languages that cannot be recognized by a DPDA, even though they can be recognized by an NPDA.

One example of a DCFL is the language {ww^R | w ∈ {0, 1}*}, which consists of all strings of the form ww^R, where w is any string of 0's and 1's, and w^R is the reverse of w. This language is deterministic context-free because it can be recognized by a DPDA that reads the input from left to right and simultaneously pushes the input onto the stack. When the DPDA reaches the end of the input, it reads the input again from right to left and pops symbols off the stack, checking that they match the input symbols.

Another example of a DCFL is the language {0^n1^n | n ≥ 0}, which consists of all strings of 0's followed by an equal number of 1's. This language is also deterministic context-free because it can be recognized by the DPDA we defined in the previous example.

DCFLs have some attractive properties that make them useful in practical applications. For example, they can be parsed efficiently using a bottom-up parsing algorithm such as LR parsing. Additionally, some programming languages have grammars that are DCFLs, which means that a DPDA can be used to parse and validate programs written in those languages.

In summary, a deterministic context-free language is a context-free language that can be recognized by a deterministic pushdown automaton. Although DCFLs are a proper subset of CFLs, they have useful properties that make them important in practical applications such as parsing and programming language theory.

Let's consider an example of a deterministic context-free language (DCFL). The language {ww^R | w ∈ {0, 1}*} is a DCFL, which consists of all strings of the form ww^R, where w is any string of 0's and 1's, and w^R is the reverse of w.

We can define a context-free grammar for this language as follows:

- S → 0S0 | 1S1 | ε

The start symbol is S, and the production rules say that we can replace S with an empty string or with the sequence 0S0 or 1S1. This generates strings of the form ww^R, where w is any string of 0's and 1's.

We can also define a DPDA that recognizes this language. The DPDA consists of a stack, an input tape, and a set of transitions. The transitions are defined as follows:

- (q0, 0, Z) → (q1, 0Z)
- (q0, 1, Z) → (q2, 1Z)
- (q1, 0, 0) → (q1, 00)
- (q1, 1, 1) → (q1, 11)
- (q1, ε, Z) → (q3, Z)
- (q2, 1, 1) → (q2, 11)
- (q2, 0, 0) → (q2, 00)
- (q2, ε, Z) → (q3, Z)

In this DPDA, the states are q0, q1, q2, and q3. The symbol Z represents the initial stack symbol, and all other symbols are terminal symbols. The DPDA starts in state q0 with the stack containing the symbol Z. Whenever it reads a 0 from the input tape, it pushes a 0 onto the stack and moves to state q1. Whenever it reads a 1 from the input tape, it pushes a 1 onto the stack and moves to state q2. When it reads an epsilon symbol from the input tape, it moves to state q3 and checks whether the remaining input matches the contents of the stack by popping symbols off the stack and comparing them to the input symbols.

Let's consider the input string 10101. The DPDA starts in state q0 with the stack containing Z. It reads the first symbol 1 from the input tape and pushes a 1 onto the stack, so the stack now contains Z1. The DPDA reads the second symbol 0 from the input tape and pushes a 0 onto the stack, so the stack now contains Z10. The DPDA reads the third symbol 1 from the input tape and pushes a

1 onto the stack, so the stack now contains Z101. The DPDA reads the fourth symbol 0 from the input tape and pops a 0 from the stack, so the stack now contains Z10. The DPDA reads the fifth symbol 1 from the input tape and pops a 1 from the stack, so the stack now contains Z1. Finally, the DPDA reaches the end of the input and moves to state q3, where it pops the remaining symbols from the stack and checks that they match the input symbols. Since the stack contents and the input are the same, the DPDA accepts the input string.

Thus, the string 10101 is in the language {ww^R | w ∈ {0, 1}*}, and it is recognized by the DPDA we defined, which is a deterministic pushdown automaton.

Exercise 4

1. Which of the following is a model of computation that extends the capabilities of a finite automaton by allowing it to use a stack?

a. Deterministic finite automaton
b. Non-deterministic finite automaton
c. Pushdown automaton
d. Turing machine

2. Which of the following is a variant of a pushdown automaton where there may be multiple possible transitions from a given state and input symbol combination?

a. Deterministic pushdown automaton
b. Non-deterministic pushdown automaton
c. Deterministic finite automaton
d. Non-deterministic finite automaton

3. Which of the following is a type of formal language that can be generated by a context-free grammar and recognized by a pushdown automaton?

a. Regular language
b. Context-sensitive language
c. Context-free language
d. Chomsky language

4. Which of the following is a variant of a pushdown automaton where there is at most one possible transition from a given state and input symbol combination?

a. Deterministic pushdown automaton
b. Non-deterministic pushdown automaton
c. Deterministic finite automaton
d. Non-deterministic finite automaton

5. Which of the following is a type of context-free language that can be recognized by a deterministic pushdown automaton?

a. All context-free languages
b. All regular languages
c. All context-sensitive languages
d. A proper subset of the context-free languages

6. Which of the following is an example of a language that is recognized by a non-deterministic pushdown automaton but not by a deterministic pushdown automaton?

a. {ww^R | w ∈ {0, 1}*}
b. {0^n1^n | n ≥ 0}
c. {0^n1^m | n, m ≥ 0}
d. {a^n b^n c^n | n ≥ 0}

7. Which of the following is an example of a deterministic context-free language?

a. {ww^R | w ∈ {0, 1}*}
b. {0^n1^n | n ≥ 0}
c. {0^n1^m | n, m ≥ 0}
d. {a^n b^n c^n | n ≥ 0}

8. Which of the following is a variant of a pushdown automaton that is used to parse deterministic context-free languages?

a. Non-deterministic pushdown automaton
b. Deterministic pushdown automaton
c. Non-deterministic finite automaton
d. Deterministic finite automaton

9. Which of the following is an advantage of deterministic context-free languages over non-deterministic context-free languages?

a. Deterministic context-free languages are more expressive than non-deterministic context-free languages.
b. Deterministic context-free languages can be recognized by deterministic pushdown automata, which are easier to implement and analyze than non-deterministic pushdown automata.
c. Deterministic context-free languages can be recognized by non-deterministic pushdown automata, which are more powerful than deterministic pushdown automata.
d. Deterministic context-free languages can be parsed using top-down parsing algorithms, which are more efficient than bottom-up parsing algorithms used for non-deterministic context-free languages.

10. Which of the following algorithms can be used to convert a non-deterministic pushdown automaton to a deterministic pushdown automaton?

a. Subset construction algorithm
b. CYK algorithm
c. LR parsing algorithm
d. Earley parsing algorithm

11. Which of the following is a property of context-free languages that ensures that they can be recognized by a pushdown automaton?

a. They can be generated by a regular grammar.
b. They can be generated by a context-sensitive grammar.
c. They can be generated by a context-free grammar.
d. They can be generated by a Chomsky grammar.

12. Which of the following is a property of deterministic pushdown automata that makes them easier to analyze than non-deterministic pushdown automata?

a. They can recognize a larger set of languages than non-deterministic pushdown automata.
b. They can recognize context-sensitive languages.
c. They can recognize non-context-free languages.
d. They have at most one possible transition from a given state and input symbol combination.

Chapter 5

Turing machines

A Turing machine is a mathematical model of computation that can simulate any computer algorithm. It was invented by Alan Turing in 1936 and is named after him. A Turing machine consists of a tape, a head that can read and write symbols on the tape, and a set of states and transitions that determine the machine's behavior.

Here are some key concepts related to Turing machines:

1. Tape: A Turing machine tape is an infinite tape divided into cells that can hold symbols. The tape head can read and write symbols on the tape.
2. Head: The Turing machine head is a device that reads and writes symbols on the tape. It can move left or right along the tape and change the symbols on the tape.
3. States: The Turing machine has a finite set of states that it can be in. The machine's behavior is determined by its current state and the symbol it is currently reading from the tape.
4. Transitions: The Turing machine has a set of transitions that determine how the machine should behave in response to its current state and the symbol it is currently reading from the tape. A transition specifies the new symbol to be written on the tape, the direction the head should move (left or right), and the new state that the machine should transition to.
5. Halting: A Turing machine halts when it reaches a special halting state. When the machine halts, it outputs the contents of the tape as the result of its computation.

Turing machines can be used to define the class of computable functions. A function is said to be computable if there exists a Turing machine that can compute it. Turing machines are also used to define the class of recursively enumerable languages, which are languages that can be generated by a Turing machine.

Turing machines are a powerful theoretical tool in computer science and are used in the analysis of algorithms and the study of computational complexity. They are also used to prove theorems about the limits of computation and the existence of undecidable problems.

Let's consider an example of a Turing machine that computes the successor function, which takes an input n and outputs n+1. We can represent numbers on the Turing machine tape as a sequence of 1's separated by a single 0. For example, the number 3 would be represented on the tape as 11101.

The Turing machine for the successor function has two states, q0 and q1, and three symbols, 0, 1, and B, where B represents the blank symbol. The transitions are defined as follows:

- $(q0, 1, B) \rightarrow (q1, B, 1)$
- $(q0, 0, B) \rightarrow (q1, B, 1)$
- $(q1, 1, B) \rightarrow (q1, B, 1)$
- $(q1, 0, B) \rightarrow (q1, B, 0)$
- $(q1, B, B) \rightarrow (qf, B, B)$

In this Turing machine, the initial state is q0, and the input is represented on the tape. The machine starts by scanning the tape from left to right, looking for the rightmost 1. When it finds a 1, it replaces it with a 0 and moves the head one cell to the right. If it finds a 0, it replaces it with a 1 and halts. If it reaches the end of the tapewithout finding a 1, it adds a new 1 to the end of the tape and halts.

Let's see how the Turing machine works on an input tape containing the number 3, which is represented as 11101. The Turing machine starts in state q0 and moves the head to the right until it reaches the rightmost 1. It then replaces the 1 with a 0 and moves the head one cell to the right. The tape now contains 11100, and the Turing machine is in state q1. The Turing machine then moves the head to the left, looking for the next rightmost 1. It finds the second rightmost 1 and replaces it with a 0, then moves the head one cell to the right. The tape now contains 11010, and the Turing machine is in state q1 again. The Turing machine repeats this process until it reaches the leftmost 1, which it replaces with a 0 and halts. The final tape contents are 10000, which represents the number 4.

Thus, the Turing machine has computed the successor function on the input 3 and output the correct result, which is 4. This Turing machine can be used to compute the successor function on any input, and can be extended to compute other arithmetic functions as well. Turing machines are a powerful tool for studying the limits of computation and the theory of computation.

Standard TM

A Standard Turing Machine, also known as a single-tape Turing machine, is a type of Turing machine that has a single tape for both input and output. A Standard Turing Machine is a simple but powerful model of computation that can simulate any algorithmic process.

The Standard Turing Machine consists of a tape divided into cells, a read/write head that can move back and forth along the tape, a finite set of states, and a set of transition rules. The tape contains a string of symbols, which can be read and written by the head. The head can move left or right along the tape, and the machine can change state and write new symbols on the tape according to a set of transition rules.

The transition rules specify the behavior of the machine in response to its current state and the symbol being read by the head. A transition rule has the form:

$$(qi, a) \rightarrow (qj, b, L/R)$$

where qi and qj are states, a and b are symbols, and L/R specifies the direction in which the head should move after writing the symbol b. L stands for "left" and R stands for "right". For example, the transition rule $(q0, 1) \rightarrow (q1, 0, R)$ means that if the machine is in state q0 and is reading a 1, it should change to state q1, write a 0 on the tape, and move the head one cell to the right.

The Standard Turing Machine can be used to simulate any algorithmic process, including the execution of computer programs. It provides a theoretical framework for understanding the limits of computation and the complexity of algorithms. The Church-Turing thesis, which states that any function that can be computed by an algorithm can be computed by a Turing machine, is based on the Standard Turing Machine model.

The Standard Turing Machine is a fundamental concept in the theory of computation and has had a profound impact on computer science and mathematics. It is a simple but powerful tool for studying the limits of computation and the structure of algorithms.

Despite its simplicity, the Standard Turing Machine is capable of performing complex computations. For example, it can be used to perform arithmetic operations, search and sort algorithms, and artificial intelligence tasks such as natural language processing and computer vision.

The Standard Turing Machine has several variations, including multi-tape Turing machines, non-deterministic Turing machines, and quantum Turing machines. Multi-tape Turing machines have multiple tapes, which can be used to simplify certain computations. Non-deterministic Turing machines can have multiple possible transitions from a given state and symbol combination, allowing them to explore different computation paths simultaneously. Quantum Turing machines are based on quantum mechanics and are used to study the theoretical limits of computation.

While the Standard Turing Machine is a powerful tool for studying the limits of computation, it has its limitations. For example, it assumes an infinite tape, which is not physically possible. It also assumes that the machine has unlimited time and resources, which is not the case in practice. In addition, some problems are computationally infeasible for a Standard Turing Machine to solve, even though they can be solved by other models of computation, such as parallel computing or quantum computing.

Despite these limitations, the Standard Turing Machine remains a fundamental concept in the theory of computation and has had a profound impact on computer science and mathematics. It provides a simple but powerful model of computation that can be used to understand the limits of computation and the structure of algorithms.

The Standard Turing Machine is also used as a theoretical tool for studying the complexity of algorithms. The time complexity of an algorithm is the amount of time it takes a Turing machine to execute the algorithm on a given input. The space complexity of an algorithm is the amount of space on the tape that the Turing machine needs to execute the algorithm on a given input.

The time and space complexity of an algorithm are important measures of its efficiency. Algorithms with lower time or space complexity are generally more efficient and faster than algorithms with higher time or space complexity. The study of time and space complexity is known as computational complexity theory and is a major area of research in computer science.

In addition to its theoretical applications, the Standard Turing Machine has had practical applications in the design of computer systems. The von Neumann architecture, which is used in most modern computers, is based on the idea of a stored-program computer, in which the instructions for the computer are stored in memory along with the data. This architecture is based on the Standard Turing Machine model and has been highly successful in the design of modern computers.

In summary, the Standard Turing Machine is a simple but powerful model of computation that can simulate any algorithmic process. It provides a framework for understanding the limits of

computation and the complexity of algorithms, and has had a profound impact on computer science and mathematics.

Let's consider an example of how a Standard Turing Machine can be used to calculate the sum of two binary numbers.

Suppose we have two binary numbers, A and B, each represented on a tape as a sequence of 0's and 1's, separated by a blank symbol. For example, A = 101 and B = 011 would be represented on the tape as:

1 0 1 B 0 1 1

To compute the sum of A and B, we can use a Standard Turing Machine with the following states and transition rules:

- q0: Initial state, move to the right until the first blank symbol is reached
- q1: Add the corresponding digits in A and B and write the result on the output tape
- q2: Carry a 1 to the next column if the sum exceeds 1
- q3: Move the head back to the left until the leftmost digit is reached

The transition rules are as follows:

- (q0, 0) → (q1, 0, R)
- (q0, 1) → (q1, 1, R)
- (q0, B) → (qf, B, S)
- (q1, 0) → (q2, 0, R)
- (q1, 1) → (q2, 1, R)
- (q2, 0) → (q0, 1, L)
- (q2, 1) → (q2, 0, R)
- (q2, B) → (q3, B, L)
- (q3, 0) → (qf, 0, S)
- (q3, 1) → (q3, 1, L)

In this Turing Machine, the input numbers A and B are represented on the tape with a blank symbol between them. The Turing Machine starts in state q0 and moves to the right until it reaches the first blank symbol. It then switches to state q1 and adds the corresponding digits in A and B, writing the result on the output tape. If the sum is greater than 1, it carries a 1 to the next column by switching to state q2. If the end of the input is reached, it moves the head back to the left until the leftmost digit is reached by switching to state q3. Finally, it halts in state qf and outputs the sum of A and B on the output tape.

For example, if we run this Turing Machine on the input tape with A = 101 and B = 011, the Turing Machine will perform the following steps:

1 0 1 B 0 1 1 (Initial input tape)

1 0 1 B 0 1 1 (Move to the right)

1 0 1 B 0 1 1 (Add 1 + 0 = 1)

1 0 1 B 1 1 1 (Carry 1)

1 0 1 B 1 1 1 (Move to the left)

1 0 1 B 1 1 1 (Move to the left)

1 0 1 B 1 1 1 (Move to the left)

1 0 1 B 1 1 1 (Move to the left)

1 0 1 B 1 1 1 (Move to the left)

1 0 1 B 1 1 1 (Move to the left)

1 0 1 B 1 1 1 (Move to the left)

1 0 1 B 1 1 1 (Move to the left)

1 0 1 B 1 1 1 (Move to the left)

1 0 1 B 1 1 0 (Output the result)

Thus, the output tape now contains the sum of A and B, which is 1000 in binary, or 8 in decimal.

This example demonstrates how a Standard Turing Machine can be used to perform arithmetic operations on binary numbers. The same principles can be extended to perform other computations and algorithms.

However, it's worth noting that the Standard Turing Machine is a theoretical model of computation, and in practice, other models of computation, such as the Random Access Machine (RAM) model, are used to analyze the performance of algorithms. The RAM model is more closely related to modern computers, as it includes concepts like memory and registers, and is therefore more practical for analyzing the performance of algorithms on real-world computers.

In addition to the RAM model, other variations of the Turing Machine have been developed to study different aspects of computation. For example, the Multi-Tape Turing Machine allows for multiple tapes, which can be used to simplify certain computations. The Non-Deterministic Turing Machine allows for multiple possible transitions from a given state and symbol combination, allowing them to explore different computation paths simultaneously. The Quantum Turing Machine is based on quantum mechanics and is used to study the theoretical limits of computation.

Despite its limitations, the Standard Turing Machine remains a fundamental concept in the theory of computation and has had a profound impact on computer science and mathematics. It provides a simple but powerful model of computation that can be used to understand the limits of computation and the structure of algorithms. It has also been used as a theoretical tool for studying the complexity of algorithms and the design of computer systems.

In summary, the Standard Turing Machine is a simple but powerful model of computation that can simulate any algorithmic process. It provides a framework for understanding the limits of computation and the complexity of algorithms, and has had a profound impact on computer science and mathematics. While it has its limitations, it remains a fundamental concept in the theory of computation and is a valuable tool for analyzing the performance and structure of algorithms.

Construction of TM

Constructing a Turing Machine involves designing the states, the tape symbols, and the transition rules that define the behavior of the machine. Here are the general steps to construct a Turing Machine:

1. Define the input and output alphabets: The input alphabet is the set of symbols that can appear on the input tape, and the output alphabet is the set of symbols that the machine can output. These alphabets can be any finite set of symbols.
2. Define the tape alphabet: The tape alphabet is the set of symbols that can appear on the tape, including the input and output alphabets. It can also include special symbols like the blank symbol B.
3. Define the states: The Turing Machine has a finite set of states that it can be in. States can be used to represent the machine's internal state or to keep track of the computation. The initial state and the halting state should be defined.
4. Define the transition rules: The transition rules specify how the machine should behave in response to its current state and the symbol it is currently reading from the tape. A transition rule has the form:

   $$(qi, a) \rightarrow (qj, b, L/R)$$

   where qi and qj are states, a and b are symbols, and L/R specifies the direction in which the head should move after writing the symbol b. L stands for "left" and R stands for "right".

5. Define the behavior of the machine: The behavior of the machine should be defined for each state and symbol combination. This can be done by specifying the set of transition rules for the machine.
6. Test the Turing Machine: Once the Turing Machine is constructed, it should be tested to ensure that it behaves correctly on a range of inputs. This can be done by manually simulating the machine on a variety of inputs, or by using a software simulation tool.

Constructing a Turing Machine can be a complex process, and requires a good understanding of the problem being solved and the behavior of the machine. However, once a Turing Machine is constructed, it can be a powerful tool for analyzing the complexity of algorithms and understanding the limits of computation.

It's also worth noting that there are different approaches to constructing a Turing Machine, depending on the problem being solved and the desired behavior of the machine. Here are a few additional tips to keep in mind when constructing a Turing Machine:

- Keep the tape alphabet as small as possible: The tape alphabet should be as small as possible to simplify the behavior of the machine. For example, if the problem being solved involves only binary numbers, the tape alphabet should be limited to the symbols 0, 1, and B.
- Use states to control the behavior of the machine: States can be used to represent different stages of the computation and to control the behavior of the machine. For example, a state could be used to represent the machine's current position in the input, or to keep track of the sum of two numbers being added.
- Break the problem down into smaller parts: Complex problems can be broken down into smaller parts, each of which can be solved by a simpler Turing Machine. This approach can simplify the behavior of the machine and make it easier to construct.
- Test the Turing Machine on a range of inputs: Testing the Turing Machine on a range of inputs can help identify any errors or unexpected behavior. It's important to test the machine on inputs that cover a range of cases, including edge cases and invalid inputs.

In summary, constructing a Turing Machine involves designing the states, tape symbols, and transition rules that define the behavior of the machine. It's important to keep the tape alphabet as small as possible, use states to control the behavior of the machine, break the problem down into smaller parts, and test the machine on a range of inputs. Once a Turing Machine is constructed, it can be a useful tool for analyzing the complexity of algorithms and understanding the limits of computation.

Turing Decidable and Turing Acceptable

In the theory of computation, a decision problem is said to be Turing decidable if there exists a Turing Machine that can solve the problem for all possible inputs. In other words, a decision problem is Turing decidable if there is an algorithmic procedure that can determine whether the input belongs to the language of the problem or not.

A Turing Machine that solves a Turing decidable problem always halts on every input, and it either accepts or rejects the input. If it accepts the input, it means the input belongs to the language of the problem, and if it rejects the input, it means the input does not belong to the language of the problem. In other words, a Turing decidable problem is a problem for which we can construct an algorithm that always gives the correct answer and terminates in a finite amount of time.

On the other hand, a language is said to be Turing acceptable if there exists a Turing Machine that accepts all strings in the language and rejects all strings not in the language. The Turing Machine that accepts a Turing acceptable language may not halt on some inputs that do not belong to the language. In other words, a Turing acceptable language is a language for which we can construct an algorithm that accepts all and only the strings in the language, but may not halt on some inputs outside the language.

It's worth noting that Turing decidable problems are a proper subset of Turing acceptable languages. Every Turing decidable problem is Turing acceptable, but the converse is not necessarily true. A Turing acceptable language may not be decidable if there is no algorithm that can always determine whether an input belongs to the language or not.

In summary, a decision problem is said to be Turing decidable if there exists a Turing Machine that can solve the problem for all possible inputs, while a language is said to be Turing acceptable if there exists a Turing Machine that accepts all strings in the language and rejects all strings not in the language. Turing decidable problems always halt and give a correct answer, while Turing acceptable languages may not halt on some inputs outside the language.

Let's consider an example to illustrate the concepts of Turing decidable and Turing acceptable.

Consider the language L = {0^n1^n | n ≥ 0}, which consists of all strings of 0's followed by an equal number of 1's. For example, some strings in L are "", "01", "0011", "000111", and so on.

We can show that L is Turing decidable by constructing a Turing Machine that decides whether an input string is in L or not. The Turing Machine works as follows:

1. Start in the initial state q0, and read the input string from the tape.
2. Move the head to the right until the first 1 is encountered, marking each 0 with a special symbol X.
3. If a 1 is encountered, move the head to the left until the first X is encountered, erasing the X and marking the 1 with a special symbol Y.
4. If a blank symbol B is encountered, accept the input if all X's have been erased, and reject otherwise.
5. If a 0 or Y is encountered, reject the input.

This Turing Machine always halts on every input, and either accepts or rejects the input. Therefore, the language L is Turing decidable.

However, L is not Turing acceptable. To see why, suppose there exists a Turing Machine M that accepts L. Then, we can use M to construct another Turing Machine M' that decides the Halting

Problem, which is known to be undecidable. This is a contradiction, since we know that the Halting Problem is not Turing decidable. Therefore, L is not Turing acceptable.

In summary, the language L = {0^n1^n | n ≥ 0} is Turing decidable, since we can construct a Turing Machine that decides whether an input string is in L or not. However, L is not Turing acceptable, since there does not exist a Turing Machine that accepts all strings in L and rejects all strings not in L.

Undecidable problems

In the theory of computation, an undecidable problem is a decision problem for which there is no Turing Machine that can solve the problem for all possible inputs. In other words, an undecidable problem is a problem for which there is no algorithmic procedure that can determine whether the input belongs to the language of the problem or not.

The most famous example of an undecidable problem is the Halting Problem, which asks whether a given program and input will eventually halt or run forever. The Halting Problem is undecidable, meaning that there is no algorithmic procedure that can determine whether a given program and input will halt or run forever.

Other examples of undecidable problems include the Post Correspondence Problem, the Busy Beaver Problem, and the Entscheidungsproblem (Decision Problem). These problems have been shown to be undecidable using techniques such as diagonalization and reduction.

The undecidability of these problems has significant implications for the theory of computation and for computer science in general. It means that there are limits to what can be computed algorithmically, and that there are problems for which we cannot determine a correct answer in a finite amount of time using a computer.

In practice, many real-world problems can be approximated or solved using heuristic algorithms or other non-algorithmic techniques. However, the existence of undecidable problems shows that there are fundamental limits to what can be computed algorithmically, and that there are problems for which we may never be able to find a correct solution using a computer.

In summary, an undecidable problem is a decision problem for which there is no Turing Machine that can solve the problem for all possible inputs. The existence of undecidable problems shows that there are fundamental limits to what can be computed algorithmically, and that there are problems for which we may never be able to find a correct solution using a computer.

Exercise 5

1. Which of the following is a type of automaton that can simulate any algorithmic process?

a) Pushdown automaton
b) Finite state machine
c) Turing Machine
d) Non-deterministic automaton

2. What is a Standard Turing Machine?

a) A Turing Machine that has only one tape
b) A Turing Machine that has multiple tapes
c) A Turing Machine that uses quantum mechanics
d) A Turing Machine that has a finite tape

3. What are the general steps to construct a Turing Machine?

a) Define the input and output alphabets, define the states, and define the behavior of the machine
b) Define the input and output alphabets, define the tape alphabet, and define the behavior of the machine
c) Define the states, define the tape alphabet, and define the behavior of the machine
d) Define the states, define the transition rules, and define the behavior of the machine

4. What is the difference between a Turing Decidable problem and a Turing Acceptable language?

a) A Turing Decidable problem always halts and gives a correct answer, while a Turing Acceptable language may not halt on some inputs outside the language.
b) A Turing Decidable problem may not halt on some inputs outside the language, while a Turing Acceptable language always halts and gives a correct answer.
c) A Turing Decidable problem and a Turing Acceptable language are the same thing.
d) A Turing Decidable problem may not halt on some inputs both inside and outside the language, while a Turing Acceptable language always halts and gives a correct answer.

5. What is an undecidable problem?

a) A problem for which there is no Turing Machine that can solve the problem for all possible inputs
b) A problem for which there is at least one Turing Machine that can solve the problem for all possible inputs
c) A problem for which there is no input that belongs to the language of the problem
d) A problem for which there is no input that does not belong to the language of the problem

6. Which of the following problems is undecidable?

a) Determining whether a given regular expression describes a regular language
b) Determining whether a given context-free grammar describes a context-free language

c) Determining whether a given context-sensitive grammar describes a context-sensitive language
d) Determining whether a given Turing Machine halts on a given input

7. Which of the following is a property of a decidable problem?

a) There exists a Turing Machine that can solve the problem for all possible inputs
b) There exists a Turing Machine that can solve the problem for some inputs
c) There exists a Turing Machine that can solve the problem for most inputs
d) There exists a Turing Machine that can solve the problem for only some specific inputs

8. Which of the following statements is true regarding Turing Machines?

a) A Turing Machine can only read the input once
b) A Turing Machine can only write to the tape once
c) A Turing Machine can move the head in both directions on the tape
d) A Turing Machine can only move the head to the right on the tape

9. Which of the following problems is Turing acceptable but not Turing decidable?

a) The problem of determining whether a given context-free grammar describes a context-free language
b) The problem of determining whether a given regular expression describes a regular language
c) The problem of determining whether a given Turing Machine halts on a given input
d) The problem of determining whether a given Turing Machine accepts a given input

10. Which of the following is a technique used to prove that a problem is undecidable?

a) Reduction
b) Induction
c) Diagonalization
d) Approximation

Chapter 6

Computability

Automata and complexity theory are important areas of computer science that deal with the study of algorithms and the complexity of problems. One important concept in these areas is computability, which refers to the ability to solve a problem using an algorithm.

In computability theory, the central question is what problems can be solved algorithmically. The answer to this question is provided by the Church-Turing thesis, which states that any problem that can be solved algorithmically can be solved by a Turing machine. A Turing machine is a mathematical model of a machine that can perform any computation that can be performed by a computer.

The theory of automata is concerned with the study of abstract machines that can solve problems. These machines are typically represented as finite state machines or pushdown automata, and they are used to model the behavior of algorithms. The theory of automata is closely related to the theory of formal languages, which is concerned with the study of languages generated by formal grammars.

The theory of complexity is concerned with the study of the resources required to solve a problem, such as time and space. The complexity of a problem is typically measured in terms of its worst-case complexity, which is the maximum amount of resources required to solve the problem for any input of a given size. The theory of complexity is closely related to the theory of computability, since problems that are not computable cannot be solved in any amount of time or space.

An example of a problem that is computable is sorting a list of numbers. Given a list of n numbers, we can write an algorithm that sorts the list in O(n log n) time using a variety of sorting algorithms such as merge sort, quicksort, or heapsort.

On the other hand, an example of a problem that is not computable is the halting problem. The halting problem asks whether a given program will halt (i.e., stop running) on a given input. It has been proven that there is no algorithm that can solve the halting problem for all possible inputs, even for very simple programs.

To see why the halting problem is not computable, consider a hypothetical algorithm that solves the halting problem. This algorithm would take as input a program P and an input I, and return whether P halts on I. Now imagine a new program Q that takes as input another program R, and runs the halting problem algorithm on R and R as inputs. If the algorithm returns "yes", then Q enters an infinite loop. Otherwise, Q halts immediately. Now consider running Q with itself as input. If the algorithm returns "yes", then Q enters an infinite loop, which contradicts the assumption that the algorithm correctly solves the halting problem. On the other hand, if the algorithm returns "no", then Q halts immediately, which again contradicts the assumption that the algorithm correctly solves the halting problem. Therefore, there can be no algorithm that solves the halting problem for all possible inputs.

In summary, automata and complexity theory are important areas of computer science that are concerned with the study of algorithms, computability, and the resources required to solve

problems. These areas are fundamental to the design and analysis of algorithms and are essential for understanding the limits of computation.

Recursive functions

Automata and complexity theory are also closely related to recursive functions. In particular, the theory of computability provides a framework for understanding the limitations of recursive functions in solving certain problems, while the theory of complexity provides a framework for understanding the resources required to compute recursive functions.

In the theory of computability, the Church-Turing thesis states that any problem that can be solved algorithmically can be solved by a Turing machine, which is a mathematical model of a machine that can perform any computation that can be performed by a computer. Recursive functions are one example of a class of algorithms that can be implemented by a Turing machine, which means that any problem that can be solved by a recursive function can also be solved by a Turing machine.

However, not all problems can be solved by a recursive function or a Turing machine. The halting problem is an example of a problem that is not solvable by any recursive function or Turing machine, as discussed in the previous answer.

In the theory of complexity, recursive functions are used to analyze the time and space complexity of algorithms. For example, the time complexity of a recursive function can be analyzed using recurrence relations, which describe the amount of time required to compute a function as a function of its inputs. The space complexity of a recursive function can also be analyzed, by keeping track of the amount of memory required to store the function's variables and data structures.

In summary, recursive functions are an important tool in automata and complexity theory, as they provide a way to implement algorithms and analyze their properties. However, the theory of computability and complexity provide a framework for understanding the limitations of recursive functions, and for understanding the resources required to compute them.

Recursive languages and recursive Enumerable languages

In automata and complexity theory, recursive languages and recursively enumerable languages are two important classes of formal languages.

A language is said to be recursive if there exists a Turing machine that can decide whether a given string belongs to the language or not. In other words, a recursive language is a language for which there exists an algorithm that can always halt and give a definite answer of "yes" or "no" for any given input string. Recursive languages are also sometimes referred to as decidable languages.

On the other hand, a language is said to be recursively enumerable if there exists a Turing machine that can recognize the language, i.e., if there exists a Turing machine that can accept all strings in the language, but may not necessarily halt or give a definite answer for strings that are not in the language. In other words, a recursively enumerable language is a language for which there exists an algorithm that can always accept any string in the language, but may not halt or give a definite answer for any string not in the language. Recursively enumerable languages are also sometimes referred to as partially decidable languages.

It is important to note that every recursive language is also recursively enumerable, but the converse is not necessarily true. That is, every algorithm that can decide a language can also recognize it, but not every algorithm that can recognize a language can necessarily decide it. This is because recognizing a language only requires that the algorithm eventually halts and accepts strings in the language, whereas deciding a language requires that the algorithm halts and gives a definite answer for all input strings.

Recursive languages and recursively enumerable languages play an important role in the study of computational complexity, as they provide a framework for classifying the difficulty of problems in terms of the resources required to decide or recognize them.

For example, if a problem can be formulated as a decision problem, i.e., a problem with a yes/no answer, and the language of all instances of the problem for which the answer is "yes" is a recursive language, then the problem is said to be decidable or solvable in polynomial time. On the other hand, if the language of all instances of the problem for which the answer is "yes" is only recursively enumerable, then the problem is said to be semi-decidable or partially solvable. Semi-decidable problems may still be computationally difficult, as they may require an exponential amount of time to recognize all instances of the problem.

Recursive languages and recursively enumerable languages are also important in the context of formal language theory, as they provide a way to characterize the complexity of formal languages. For example, it is known that the regular languages, which can be recognized by finite automata, are a subset of the recursive languages, which can be recognized by Turing machines. Similarly, it can be shown that every context-free language, which can be recognized by a pushdown automaton, is also a recursive language.

In summary, recursive languages and recursively enumerable languages are two important classes of formal languages that play a fundamental role in the study of automata and complexity theory. These classes provide a framework for classifying the complexity of problems and formal languages, and can help us understand the resources required to solve computational problems.

Exercise 6

1. Which of the following is true?

A) Every recursive language is recursively enumerable.
B) Every recursively enumerable language is recursive.
C) Every decidable language is recursively enumerable.
D) Every recursively enumerable language is decidable.

2. Which of the following is true?

A) Every recursive function is computable.
B) Every computable function is recursive.
C) Every primitive recursive function is recursive.
D) Every recursive function is primitive recursive.

3. Which of the following is false?

A) Every Turing machine can be simulated by a recursive function.
B) Every recursive function can be simulated by a Turing machine.
C) Every Turing machine can be simulated by a pushdown automaton.
D) Every pushdown automaton can be simulated by a Turing machine.

4. Which of the following is true?

A) Every recursive language is decidable.
B) Every recursively enumerable language is semi-decidable.
C) Every semi-decidable language is recursive.
D) Every decidable language is recursively enumerable.

5. Which of the following is true?

A) Every recursively enumerable language is context-free.
B) Every context-free language is recursively enumerable.
C) Every regular language is recursively enumerable.
D) Every recursively enumerable language is regular.

6. Which of the following is false?

A) Every recursive function is total.
B) Every total function is recursive.
C) Every partial function is not computable.
D) Every computable function is total or partial.

7. Which of the following is true?

A) Every recursively enumerable language is Turing-recognizable.
B) Every recursively enumerable language is Turing-decidable.
C) Every recursive language is Turing-recognizable.
D) Every recursive language is Turing-decidable.

8. Which of the following is false?

A) Every recursive function is primitive recursive.
B) Every primitive recursive function is recursive.
C) Every primitive recursive function is total.
D) Every total function is recursive.

9. Which of the following is true?

A) Every recursively enumerable language is not recursive.
B) Every recursive language is recursively enumerable.
C) Every decidable language is not recursively enumerable.
D) Every semi-decidable language is decidable.

10. Which of the following is true?

A) Every recursively enumerable language is not context-sensitive.
B) Every context-sensitive language is recursively enumerable.
C) Every context-free language is context-sensitive.
D) Every context-sensitive language is context-free.

Chapter 7

Computational complexity

In automata and complexity theory, computational complexity is the study of the resources required to solve computational problems, particularly in terms of time and space. It seeks to understand which problems can be solved efficiently and which problems require an inordinate amount of resources to solve.

One way to measure the complexity of a problem is to analyze the running time of an algorithm that solves the problem. The running time of an algorithm is often expressed in terms of its worst-case time complexity, which is the maximum number of steps the algorithm will take on any input of size n. The time complexity is typically measured in terms of a function of n, such as O(n), O(n^2), or O(2^n), which describes the growth rate of the running time as the input size increases.

Another way to measure the complexity of a problem is to analyze the space required by an algorithm that solves the problem. The space complexity of an algorithm is the maximum amount of memory required by the algorithm on any input of size n. The space complexity is also typically measured in terms of a function of n, such as O(1), O(n), or O(n^2), which describes the growth rate of the memory usage as the input size increases.

The study of computational complexity has led to the identification of several important complexity classes, such as P, NP, and NP-complete. The class P consists of problems that can be solved in polynomial time, i.e., in time O(n^k) for some fixed constant k. The class NP consists of problems for which a proposed solution can be verified in polynomial time, but for which no known algorithm can solve them in polynomial time. The class NP-complete consists of the hardest problems in NP, and any problem in NP can be reduced to an NP-complete problem in polynomial time.

Understanding the computational complexity of a problem is important because it can help us determine whether a problem is solvable in practice, and if so, how efficiently it can be solved. In addition, the study of computational complexity has led to the development of many important algorithms and data structures, such as dynamic programming, divide and conquer, and greedy algorithms, which are used to solve problems efficiently in practice.

One of the most famous open problems in computational complexity is the P vs. NP problem, which asks whether every problem in NP can be solved in polynomial time. This problem has deep implications for cryptography, optimization, and artificial intelligence, among other fields, and it remains one of the most important and challenging problems in computer science.

Another important concept in computational complexity is the notion of hardness of approximation. Many optimization problems are NP-hard, meaning that they are at least as hard as the hardest problems in NP. However, some NP-hard optimization problems can be approximated to within some factor of the optimal solution, and the study of hardness of approximation seeks to understand the limits of such approximation algorithms.

Computational complexity is also closely related to the theory of computability, which concerns the fundamental limits of computation. The Church-Turing thesis, for example, states that every

effectively computable function can be computed by a Turing machine. This thesis has profound implications for the study of complexity, as it implies that any algorithm that can be implemented on a physical computer can be simulated by a Turing machine.

In summary, computational complexity is the study of the resources required to solve computational problems, particularly in terms of time and space. It seeks to understand which problems can be solved efficiently and which problems require an inordinate amount of resources to solve, and it has led to the identification of several important complexity classes, such as P, NP, and NP-complete. The study of computational complexity has deep implications for cryptography, optimization, and artificial intelligence, among other fields, and it remains an active area of research in computer science today.

Big-O notations

In automata and complexity theory, Big-O notation is a way to describe the growth rate of a function as its input size approaches infinity. It is used to analyze the time complexity and space complexity of algorithms, and to compare the efficiency of different algorithms for solving the same problem.

Formally, let f(n) and g(n) be two functions that take a positive integer n as input. We say that f(n) is O(g(n)), or that f(n) is "big-O" of g(n), if there exist positive constants c and n_0 such that for all n >= n_0, f(n) <= c * g(n). Intuitively, this means that f(n) grows no faster than g(n) up to a constant factor, as n becomes very large.

For example, if we have an algorithm that takes n^2 + n + 1 steps to solve a problem of size n, we can say that the running time of the algorithm is O(n^2). This is because as n becomes very large, the term n^2 dominates the other terms, and the running time grows no faster than n^2 up to a constant factor. Similarly, if we have an algorithm that requires n^2 bits of memory to store its data structures, we can say that the space complexity of the algorithm is O(n^2).

Big-O notation is useful because it allows us to compare the efficiency of different algorithms for solving the same problem, without worrying about the details of their implementation. For example, if we have two algorithms for sorting a list of n numbers, one with a running time of O(n^2) and another with a running time of O(n log n), we know that the second algorithm is more efficient for large values of n, even if the constants hidden by the big-O notation are different.

Big-O notation is also useful because it allows us to reason about the scalability of algorithms. For example, if we have an algorithm with a running time of O(n^2), we know that the running time will increase quadratically as the input size grows, which may not be feasible for very large inputs. On the other hand, if we have an algorithm with a running time of O(n log n), we know that the running time will increase much more slowly as the input size grows, which may be feasible for very large inputs.

In summary, Big-O notation is a way to describe the growth rate of a function as its input size approaches infinity. It is used to analyze the time complexity and space complexity of algorithms, and to compare the efficiency of different algorithms for solving the same problem.

Here are a few examples of Big-O notation:

1. Suppose we have an algorithm that takes 5n^3 + 2n^2 + 10n + 100 steps to solve a problem of size n. We can say that the running time of the algorithm is O(n^3), because as n becomes very large, the term 5n^3 dominates the other terms, and the running time grows no faster than n^3 up to a constant factor.

2. Suppose we have an algorithm that requires 3n^2 + 5n + 2 bits of memory to store its data structures. We can say that the space complexity of the algorithm is O(n^2), because as n becomes very large, the term 3n^2 dominates the other terms, and the memory usage grows no faster than n^2 up to a constant factor.

3. Suppose we have two algorithms for sorting a list of n numbers. Algorithm A has a running time of 2n^2 + 10n, while algorithm B has a running time of 3n log n + 5n. We can say that the running time of algorithm A is O(n^2), and the running time of algorithm B is O(n log n). This means that algorithm B is more efficient than algorithm A for large values of n, even if the constants hidden by the big-O notation are different.

4. Suppose we have an algorithm that takes 2^n steps to solve a problem of size n. We can say that the running time of the algorithm is O(2^n), because the running time grows exponentially with the input size. This means that the algorithm is not feasible for large values of n, as the running time becomes intractable very quickly.

In general, the purpose of Big-O notation is to provide a simple and standardized way to compare the efficiency of algorithms and to reason about their scalability. By using Big-O notation, we can focus on the most significant terms of the running time or memory usage, and ignore the details of the implementation that may vary depending on the programming language or hardware platform.

Class P vs class NP

The classes P and NP are two of the most important complexity classes in automata and complexity theory. The class P consists of decision problems that can be solved by a deterministic Turing machine in polynomial time. The class NP consists of decision problems for which a proposed solution can be verified in polynomial time by a deterministic Turing machine.

The key difference between P and NP is that problems in P can be solved efficiently in polynomial time, while problems in NP may be difficult to solve, but easy to verify. This means that if we have a proposed solution to an NP problem, we can check whether the solution is correct in polynomial time, but we may not be able to find the solution itself in polynomial time.

One famous open problem in computer science is whether P = NP. This problem asks whether every problem in NP can be solved in polynomial time. If P = NP, then it would mean that every problem for which a solution can be verified in polynomial time can also be solved in polynomial time. This would have profound implications for many fields, including cryptography, optimization, and artificial intelligence.

Despite decades of research, the question of whether P = NP remains unresolved. Many researchers believe that P ≠ NP, and that there are problems in NP that are fundamentally difficult to solve.

However, no one has been able to prove this rigorously, and the problem remains one of the most important and challenging open problems in computer science.

To summarize, the classes P and NP are two of the most important complexity classes in automata and complexity theory. Problems in P can be solved efficiently in polynomial time, while problems in NP may be difficult to solve, but easy to verify. The question of whether P = NP is one of the most important open problems in computer science, and its resolution would have profound implications for many fields.

Here's an example of a problem in P and a problem in NP:

1. Problem in P: Given a list of n numbers, find the largest number in the list. This problem can be solved by scanning the list once and keeping track of the largest number seen so far. The running time of this algorithm is O(n), which is polynomial in the size of the input.

2. Problem in NP: Given a list of n numbers and a target sum k, find a subset of the numbers that adds up to k. This problem is known as the subset sum problem, and it is an example of an NP-complete problem. It is easy to verify a proposed solution to the subset sum problem, by checking whether the sum of the selected numbers is equal to k. However, it is not known whether there exists an algorithm that can solve the subset sum problem in polynomial time, i.e., whether the subset sum problem is in P or not.

In general, problems in P are those that can be solved efficiently in polynomial time, while problems in NP are those for which a proposed solution can be verified in polynomial time, but for which no known algorithm can solve them in polynomial time. The question of whether P = NP is an open problem in computer science, and its resolution would have profound implications for many fields, including cryptography, optimization, and artificial intelligence.

Polynomial time reduction and NP-complete problems

Polynomial time reduction is a key concept in automata and complexity theory, and it is used to show that one problem is at least as hard as another problem. A polynomial time reduction from problem A to problem B is an algorithm that transforms instances of problem A into instances of problem B in polynomial time, such that the answer to the transformed instance of problem B is the same as the answer to the original instance of problem A. In other words, if we can solve problem B efficiently, then we can solve problem A efficiently as well.

One important use of polynomial time reduction is in the study of NP-complete problems. A problem is NP-complete if it is in NP and every other problem in NP can be reduced to it in polynomial time. The first NP-complete problem was discovered by Stephen Cook in 1971, and it is called the Boolean satisfiability problem (SAT). The SAT problem asks whether there exists a truth assignment that satisfies a given Boolean formula, and it is known to be NP-complete.

The importance of NP-complete problems lies in the fact that if we can solve one NP-complete problem in polynomial time, then we can solve all NP-complete problems in polynomial time, and hence P = NP. This is because any problem in NP can be reduced to an NP-complete problem in

polynomial time, and hence if we can solve an NP-complete problem efficiently, we can use polynomial time reduction to solve any other problem in NP efficiently as well.

Despite decades of research, no one has been able to find a polynomial time algorithm for any NP-complete problem. This has led many researchers to believe that NP-complete problems are inherently difficult to solve, and that P ≠ NP. However, this remains an open question, and the resolution of the question of whether P = NP or not remains one of the most important open problems in computer science.

In summary, polynomial time reduction is a key concept in automata and complexity theory, and it is used to show that one problem is at least as hard as another problem. NP-complete problems are problems that are in NP and every other problem in NP can be reduced to them in polynomial time. The resolution of the question of whether P = NP or not remains one of the most important open problems in computer science, and it has profound implications for many fields, including cryptography, optimization, and artificial intelligence.

Here's an example of polynomial time reduction and an NP-complete problem:

The problem of the Hamiltonian cycle is an NP-complete problem. Given an undirected graph G, a Hamiltonian cycle is a cycle that visits every vertex exactly once. The problem asks whether there exists a Hamiltonian cycle in the graph G.

We can reduce the problem of the Hamiltonian cycle to the problem of the travelling salesman problem (TSP), which is another NP-complete problem. Given a set of cities and the distances between them, the TSP asks for the shortest possible route that visits each city exactly once and returns to the starting city. The reduction works as follows:

1. Given a graph G, we construct a complete graph G' by adding edges between all pairs of vertices that are not already connected in G. The weight of each edge in G' is set to 1.

2. We then apply an algorithm that solves the TSP to the graph G'. If the shortest route found by the algorithm has length n, then we know that there exists a Hamiltonian cycle in G if and only if the length of the route is n.

The reduction from the Hamiltonian cycle problem to the TSP problem shows that the Hamiltonian cycle problem is at least as hard as the TSP problem. Since the TSP problem is known to be NP-complete, this implies that the Hamiltonian cycle problem is also NP-complete.

In general, the use of polynomial time reduction allows us to show that one problem is at least as hard as another problem, and hence to understand the relative difficulty of different problems. The existence of NP-complete problems implies that there are problems in NP that are inherently difficult to solve, and that may not have efficient algorithms. The resolution of the question of whether P = NP or not remains one of the most important open problems in computer science.

Cook's Theorem

Cook's theorem, also known as the Cook-Levin theorem, is a fundamental result in complexity theory that shows that the Boolean satisfiability problem (SAT) is NP-complete. The theorem was proved independently by Stephen Cook and Leonid Levin in 1971, and it is widely regarded as one of the most important results in the theory of computation.

The SAT problem asks whether there exists a truth assignment that satisfies a given Boolean formula. It is one of the most studied problems in theoretical computer science and has numerous applications in areas such as formal verification, cryptography, and artificial intelligence.

Cook's theorem states that every problem in NP can be reduced to SAT in polynomial time. This means that if we can efficiently solve the SAT problem, we can efficiently solve any other problem in NP. The reduction is achieved by encoding the input to the problem as a Boolean formula, such that the formula is satisfiable if and only if the input instance is a "yes" instance of the problem.

The proof of Cook's theorem is based on the concept of a non-deterministic Turing machine, which is a theoretical model of computation that allows for multiple possible computations at each step. Cook showed that every problem in NP can be decided by a non-deterministic Turing machine in polynomial time, and that this computation can be simulated by a Boolean circuit of polynomial size. The Boolean circuit can then be encoded as a Boolean formula in conjunctive normal form (CNF), which is a standard representation of Boolean formulas used in SAT solvers.

Cook's theorem has important implications for the theory of computation and for practical applications in computer science. It provides a theoretical justification for the study of NP-complete problems, which are believed to be inherently difficult to solve. It also provides a framework for developing approximation algorithms and heuristics for solving NP-hard problems, which are problems that are at least as hard as NP-complete problems. Finally, it has led to the development of efficient SAT solvers, which have numerous applications in formal verification, model checking, and software engineering.

The importance of Cook's theorem goes beyond its theoretical implications. One of the most significant practical applications of the theorem is in the field of automated reasoning and formal verification. Formal verification is the process of verifying whether a system or a program satisfies a given specification or a set of requirements. Formal verification is important in safety-critical systems, such as avionics, medical devices, and nuclear power plants, where errors can have catastrophic consequences.

SAT solvers, which are based on the principles of Cook's theorem, are widely used in formal verification to check the correctness of complex systems and software. The idea is to encode the correctness conditions of the system as a SAT problem and then use a SAT solver to check whether the conditions are satisfied. This approach has been successfully applied in many real-world applications, including the verification of hardware designs, software systems, and cryptographic protocols.

In summary, Cook's theorem is a fundamental result in complexity theory that shows that the Boolean satisfiability problem is NP-complete. The theorem has important theoretical implications

for the study of NP-complete problems and their properties. It also has important practical applications in formal verification, automated reasoning, and software engineering. The development of efficient SAT solvers, which are based on the principles of Cook's theorem, has enabled the verification of complex systems and software, and has contributed to improving the safety and reliability of critical systems.

Here's an example of how Cook's theorem can be used to show that a problem is NP-complete:

Consider the problem of 3-SAT, which is a variant of the SAT problem. In 3-SAT, we are given a Boolean formula in conjunctive normal form (CNF) in which each clause contains exactly three literals. The problem is to determine whether there exists a truth assignment that satisfies the formula.

To show that 3-SAT is NP-complete, we can use Cook's theorem to reduce the SAT problem to 3-SAT. The reduction works as follows:

1. Given an instance of the SAT problem, we convert the Boolean formula into an equivalent formula in CNF form.
2. We then transform each clause in the CNF formula into a set of clauses that contain exactly three literals. This can be done by introducing new variables and using the distributive property of Boolean logic.
3. Finally, we combine all the transformed clauses into a single CNF formula in which each clause contains exactly three literals.

It can be shown that the reduction from SAT to 3-SAT can be done in polynomial time, and that the resulting 3-SAT formula is satisfiable if and only if the original SAT formula is satisfiable. Therefore, if we can solve the 3-SAT problem efficiently, we can solve the SAT problem efficiently as well.

Since SAT is an NP-complete problem, and we have shown that SAT can be reduced to 3-SAT in polynomial time, it follows that 3-SAT is also NP-complete.

In summary, Cook's theorem provides a powerful tool for showing that a problem is NP-complete. By using polynomial time reductions, we can show that a problem is at least as hard as an NP-complete problem, and hence is itself NP-complete. The reduction from SAT to 3-SAT is just one example of how Cook's theorem can be used to establish the complexity of a problem.

Exercise 7

1. Which of the following measures the worst-case running time of an algorithm?

   a. Big-O notation
   b. Theta notation
   c. Omega notation
   d. None of the above

2. Which of the following is not a standard complexity class?

a. P
b. NP
c. NP-hard
d. NPH

3. Which of the following is an example of an NP-complete problem?

a. Finding the shortest path in a graph
b. Sorting a list of numbers
c. Finding the maximum element in an array
d. The traveling salesman problem

4. Which of the following is a characteristic of problems in P?

a. They can be solved in exponential time
b. They can be solved in polynomial time
c. They are always solvable
d. They are always unsolvable

5. Which of the following is true about problems in NP?

a. They can be solved in exponential time
b. They can be solved in polynomial time
c. They are always solvable
d. They are always unsolvable

6. Which of the following is an example of a polynomial time reduction?

a. Reducing the 3-SAT problem to the traveling salesman problem
b. Reducing the Boolean satisfiability problem to the Hamiltonian cycle problem
c. Reducing the subset sum problem to the knapsack problem
d. None of the above

7. Which of the following is a consequence of Cook's theorem?

a. Every problem in P is also in NP
b. Every problem in NP can be reduced to the Boolean satisfiability problem in polynomial time
c. Every problem in NP can be solved in polynomial time
d. None of the above

8. Which of the following is a property of NP-complete problems?

a. They are always solvable in polynomial time
b. They are always unsolvable
c. They are at least as hard as any other problem in NP
d. None of the above

9. Which of the following is an example of an NP-hard problem?

a. The knapsack problem
b. The traveling salesman problem
c. The subset sum problem
d. All of the above

10. What is the significance of Cook's theorem?

a. It shows that the Boolean satisfiability problem is in P
b. It shows that the traveling salesman problem is NP-complete
c. It shows that every problem in NP can be reduced to the Boolean satisfiability problem in polynomial time
d. None of the above

Answer key

Exercise 1's Answers

1. b) It is a branch of mathematics that studies the behavior of computing devices
2. b) A set of symbols used to form strings
3. d) The set of all strings that start with 1 and end with 0
4. b) A mathematical model of a computing device that recognizes languages generated by regular grammars
5. c) DFAs have a deterministic transition function, while NFAs have a non-deterministic transition function
6. b) They can only generate regular languages
7. a) $S \rightarrow aSb \mid \varepsilon$
8. b) A mathematical model of a computing device that recognizes languages generated by context-free grammars
9. d) They can recognize recursively enumerable languages
10. b) They are sets of strings generated by a set of rules

Exercise 2's Answers

1. D. {0, 1, 00, 01, 10, 11, 000, 001, ...} is a regular language that can be generated by a regular grammar or recognized by a finite automaton.
2. D. Concatenation is a basic operator used in regular expressions to combine two or more regular expressions to form a new regular expression.
3. A. Every regular language can be generated by a regular expression. This is known as the Kleene's theorem.
4. D. $S \rightarrow AB$ is a production rule in a regular grammar that maps the starting symbol S to the concatenation of the nonterminal symbols A and B.
5. C. The pumping lemma for regular languages is used to prove that a language is not regular. It states that every regular language has a pumping length such that any string longer than that length can be divided into three parts that can be pumped. If a language does not satisfy this condition, it cannot be regular.
6. D. {a^n b^n c^n d^n | $n \geq 0$} is not a regular language because it requires counting the number of occurrences of two different symbols, which cannot be done by a finite automaton.

7. B. Context-free grammars generate context-free languages, which are a more general class of languages than regular languages.
8. C. Closure under Kleene star is a basic property of regular languages. If L is a regular language, then L* (the Kleene star of L) is also a regular language.
9. B. {a^n b^n c^n | n ≥ 0} is not a regular language because it requires comparing the number of occurrences of three different symbols, which cannot be done by a finite automaton.
10. D. Every regular language has an infinite number of strings, and the pumping lemma for regular languages provides a way to identify a subset of those strings that can be pumped.

Exercise 3's Answers

1. d) {a^n b^m c^k | n + m = k}
2. d) Ambiguity occurs when a grammar generates more than one parse tree for a given string.
3. a) S, b) aSb, c) aaSbb, and d) aab are all sentential forms of the grammar.
4. b) is a parse tree for the string "aabb" generated by the grammar.
5. c) S → AaB | AB | a, A → B | a, B → Ab is a simplified grammar equivalent to the given grammar.
6. d) Introducing new nonterminal symbols is not a method for simplifying a context-free grammar, but rather a common technique used in other transformations such as eliminating left recursion.
7. c) Type-2 (context-free grammars) can generate all context-free languages.
8. c) E → T + E | T, T → F T' | F, T' → * F T' | ε, F → (E) | id is a context-free grammar for the language of arithmetic expressions with addition and multiplication.
9. d) A pushdown automaton can recognize non-context-free languages, but it cannot recognize all context-sensitive languages.
10. b) LR parser is a parsing algorithm that uses a bottom-up strategy.
11. a) {a^n b^n c^n | n ≥ 0} is not context-free, but the others are.
12. d) A parse tree may not be uniquely constructed for a given string generated by a grammar if the grammar is ambiguous.
13. c) Converting the grammar to regular form is not a step in simplifying a context-free grammar, but rather a transformation that changes the type of grammar.

14. a) S → 0S0 | 1S1 | ε is a context-free grammar for the language of all palindromes over the alphabet {0, 1}.
15. d) S → AB | BA, A → aB | ε, B → bA | ε is an ambiguous grammar, as it generates the same string with two different parse trees.
16. a) LL parser is a parsing algorithm that uses a top-down strategy.
17. d) S → aA | B, A → aA | ε, B → bB | ε is a context-free grammar for the language of all strings over the alphabet {a, b} that contain at least one 'a'.
18. c) Every context-free language is not necessarily a regular language, as there are context-free languages that cannot be generated by a regular grammar.
19. b) {0^n 1^m 0^n | n,m ≥ 0} can be generated by a context-sensitive grammar but not by a context-free grammar.
20. a) S → aS | ε is a simplification of the given grammar.

Exercise 4's Answers

1. c. Pushdown automaton
2. b. Non-deterministic pushdown automaton
3. c. Context-free language
4. a. Deterministic pushdown automaton
5. d. A proper subset of the context-free languages.
6. d. {a^n b^n c^n | n ≥ 0}
7. b. {0^n1^n | n ≥ 0}
8. b. Deterministic pushdown automaton
9. b. Deterministic context-free languages can be recognized by deterministic pushdown automata, which are easier to implement and analyze than non-deterministic pushdown automata.
10. a. Subset construction algorithm
11. c. They can be generated by a context-free grammar.
12. d. They have at most one possible transition from a given state and input symbol combination.

Exercise 5's Answers

1. c) Turing Machine
2. a) A Turing Machine that has only one tape

3. b) Define the input and output alphabets, define the tape alphabet, and define the behavior of the machine
4. a) A Turing Decidable problem always halts and gives a correct answer, while a Turing Acceptable language may not halt on some inputs outside the language.
5. a) A problem for which there is no Turing Machine that can solve the problem for all possible inputs
6. d) Determining whether a given Turing Machine halts on a given input
7. a) There exists a Turing Machine that can solve the problem for all possible inputs
8. c) A Turing Machine can move the head in both directions on the tape
9. d) The problem of determining whether a given Turing Machine accepts a given input
10. c) Diagonalization

Exercise 6's Answers

1. B) Every recursively enumerable language is recursive.
2. A) Every recursive function is computable.
3. C) Every Turing machine can be simulated by a pushdown automaton.
4. B) Every recursively enumerable language is semi-decidable.
5. B) Every context-free language is recursively enumerable.
6. A) Every recursive function is total.
7. A) Every recursively enumerable language is Turing-recognizable.
8. A) Every recursive function is primitive recursive.
9. B) Every recursive language is recursively enumerable.
10. B) Every context-sensitive language is recursively enumerable.

Exercise 7's Answers

1. a. Big-O notation
2. d. NPH
3. d. The traveling salesman problem
4. b. They can be solved in polynomial time
5. a. They can be solved in exponential time
6. b. Reducing the Boolean satisfiability problem to the Hamiltonian cycle problem
7. b. Every problem in NP can be reduced to the Boolean satisfiability problem in polynomial time

8. c. They are at least as hard as any other problem in NP
9. d. All of the above
10. c. It shows that every problem in NP can be reduced to the Boolean satisfiability problem in polynomial time