ALC

Assignment-II

**Short Answer Questions**

1.Distinguish between a PDA and a FA.

2Construct push down automata that accept the language L = {anbn / n>=1}.

3a)List the closure properties of Recursively Enumerable Languages

b) What is unrestricted grammar? Give an example.

4. Compare and contrast DPDA and NPDA

|  |
| --- |
| 5. Write short notes on Liner Bounded Automata  6.Write Short notes on Pumping Lemma |

**Long Answer Questions**

1. What is Turing machine? Specify its model and construct TM for the language

L = {anbncn : n>=1}

2. a)Construct PDA that recognizes the language : L1 = {an b2n / n>=1}

b) Compare and contrast DPDA and NPDA

3. a)Distinguish between regular languages and recursively enumerable languages

b) Explain Chomsky Hierarchy of languages.

4. Design Turing machine to accept all set of palindromes over {0, 1}\*, also write the instantaneous description on the string 1001001.

5. Construct a PDA that recognizes strings (over alphabet 0 and 1) that contain equal number of 0s and 1s.

Distinguish between a PDA and a FA

A PDA (Pushdown Automaton) and a FA (Finite Automaton) are both types of automata that are used to recognize languages. However, they have some key differences:

* A PDA has an additional memory component called a pushdown stack, which allows it to remember and access multiple pieces of information at a time. This makes PDAs more powerful than FAs, and able to recognize a larger class of languages called context-free languages.
* A FA only has a finite amount of memory, which is represented by the current state. This means that it can only remember a limited amount of information at a time, and it can only recognize a smaller class of languages called regular languages.
* PDAs can have multiple possible transitions for a given input symbol and current state, whereas FAs have exactly one transition.
* PDAs are more powerful and expressive than FAs, but they are also more complex and harder to design and analyze.

Construct push down automata that accept the language L = {anbn / n>=1}

A pushdown automaton (PDA) that accepts the language L = {anbn / n>=1} can be constructed as follows:

* The set of states is {q0, q1, q2, q3, q4}.
* The initial state is q0.
* The set of input symbols is {a, b}.
* The set of stack symbols is {Z, A}. where Z is the initial stack symbol
* The transition function is defined as follows:
  + (q0, a, Z) -> (q1, AZ)
  + (q1, a, A) -> (q1, AA)
  + (q1, b, A) -> (q2, Z)
  + (q2, b, Z) -> (q2, Z)
  + (q2, epsilon, Z) -> (q3, epsilon)
* The set of final states is {q3}

The PDA starts in the initial state q0 with an empty stack. When it reads 'a', it pushes an 'A' onto the stack. For each 'a' it reads it pushes an 'A' into the stack. When it reads 'b' it pops an 'A' from the stack. As long as it reads 'b' and there is an 'A' in the stack it continues to pop the stack. When the stack becomes empty and there is no more input, it moves to the final state q3, indicating that the string is accepted.

This PDA can accept strings of the form anbn where n>=1, such as "ab", "aabb", "aaabbb", and so on.

List the closure properties of Recursively Enumerable Languages

The closure properties of recursively enumerable languages (also known as Turing-recognizable languages) include:

1. Union: The union of two recursively enumerable languages is also recursively enumerable.
2. Concatenation: The concatenation of two recursively enumerable languages is also recursively enumerable.
3. Kleene star: The Kleene star of a recursively enumerable language is also recursively enumerable.
4. Homomorphism: The image of a recursively enumerable language under a homomorphism is recursively enumerable.
5. Inverse homomorphism: The inverse image of a recursively enumerable language under a homomorphism is recursively enumerable.
6. Intersection with regular languages: The intersection of a recursively enumerable language with a regular language is recursively enumerable.

It's worth to mention that not all recursively enumerable languages have the same power, some recursively enumerable languages are more powerful than others. The class of languages that are recursively enumerable but not recursive is called the context-sensitive languages.

What is unrestricted grammar? Give an example

An unrestricted grammar is a type of formal grammar that does not impose any restrictions on the form of the productions (or rules) that generate the strings of the language. In other words, an unrestricted grammar can generate any string of terminals, regardless of the context or structure of the string.

An example of an unrestricted grammar is a type-3 grammar in the Chomsky hierarchy also called as context-free grammar.

Here is an example of an unrestricted grammar that generates the language {a^n b^n | n >= 1}

* S -> aSb | ab
* a,b are the terminals
* S is the start symbol

This grammar generates strings of the form "ab", "aab", "aaab", and so on. The unrestricted grammar allows the production S -> aSb to generate any number of a's before generating b's, there is no restriction on the context of the string.

Compare and contrast DPDA and NPDA

DPDA (Deterministic Pushdown Automata) and NPDA (Nondeterministic Pushdown Automata) are both types of pushdown automata (PDAs) that are used to recognize languages. However, they have some key differences:

* Deterministic: A DPDA has exactly one transition for a given input symbol and current state. In contrast, an NPDA may have multiple possible transitions for a given input symbol and current state.
* Accepting: A DPDA accepts a string if there is a sequence of transitions starting from the initial state that leads to a final state and consumes all the input symbols, and the stack is empty. An NPDA accepts a string if there is at least one sequence of transitions starting from the initial state that leads to a final state and consumes all the input symbols, and the stack is empty.
* Language recognition: DPDA's recognize only regular languages, which are a subset of context-free languages, while NPDA's can recognize context-free languages.
* Simplicity: DPDA's are simpler to design, analyze and understand than NPDA's because of the deterministic nature. NPDA's are more powerful than DPDA's but also more complex.
* Equivalence: Every NPDA can be converted into a DPDA, but not vice-versa.

In summary, DPDA's are deterministic, simpler, and recognize only regular languages while NPDA's are non-deterministic, more powerful, and can recognize context-free languages.

Write short notes on Liner Bounded Automata

Linear Bounded Automata (LBA) is a type of automaton that operates on an input tape and a tape called the "bounded tape". It's a variation of Turing Machine.

* The input tape is infinite, and the bounded tape is finite. The automaton can move left or right on the input tape, but it can only move left on the bounded tape.
* LBA's have a read-write head that reads the input tape and the bounded tape, and can write symbols to the bounded tape.
* LBA's are more powerful than a finite automaton, and can recognize a larger class of languages called context-sensitive languages.
* LBA's are used in the study of computational complexity and in the design of algorithms for parsing and translation.
* The bounded tape of LBA is used as a memory that can store a limited amount of information and helps to keep track of the context of the input string.
* LBA's are less powerful than Turing machines because the bounded tape is finite and therefore the LBA's cannot store an unbounded amount of information.
* An LBA can be thought of as a Turing Machine with a "bounded memory" that can only store a limited amount of information.

Write Short notes on Pumping Lemma

The pumping lemma is a tool used to prove that a language is not regular. It is a way of showing that a given language cannot be generated by any finite automaton or regular grammar.

* The pumping lemma states that for any regular language L and any number p (the pumping length), any string w in L such that |w| >= p, can be written as w = xyz, where:
  + |xy| <= p
  + |y| > 0
  + for all i >= 0, xy^iz is in L
* The pumping lemma can be used to show that a language is not regular by assuming that the language is regular and then finding a string that cannot be broken down into the xyz form specified by the pumping lemma.
* The pumping lemma applies to all regular languages, it is not restricted to context-free languages or context-sensitive languages.
* The pumping lemma is used to show that a language is not regular, it cannot be used to prove that a language is regular.
* The pumping lemma is a useful tool for understanding the properties of regular languages, it is widely used in automata theory and formal language theory.

In summary, the pumping lemma is a technique used to show that a language is not regular by showing that there exists a string in the language that cannot be broken down into a specific form, it's widely used in automata theory and formal language theory, but it can't prove that a language is regular.

Construct PDA that recognizes the language : L1 = {an b2n / n>=1}

A PDA (Pushdown Automata) can be constructed to recognize the language L1 = {an b2n / n>=1} as follows:

* The input alphabet is {a, b}
* The stack alphabet is {A, B, Z} where Z is the initial stack symbol
* The set of states is {q0, q1, qAccept, qReject}
* The start state is q0
* The accept state is qAccept
* The reject state is qReject
* The transition function is as follows:

q0, a, Z -> q0, AZ, R q0, b, Z -> q1, BZ, R q1, b, Z -> q1, BZ, R q1, b, B -> q0, BB, R q0, a, A -> q0, AA, R q0, e, A -> qAccept, Z, L

* The PDA starts in state q0 and the stack contains the initial symbol Z.
* When it reads a 'a' it pushes an 'A' on top of the stack and moves to the next input symbol.
* When it reads a 'b' it pushes a 'B' on top of the stack and moves to the next input symbol.
* When it reads a 'b' again, it pops a 'B' from the stack and pushes another 'B' on top of the stack and moves to the next input symbol.
* When it reads a 'a' again, it pops an 'A' from the stack and pushes another 'A' on top of the stack and moves to the next input symbol.
* When it reads the end of the input and the stack only contains 'Z' , it goes to the accept state qAccept, otherwise it goes to the reject state qReject.

This PDA will accept the strings of the form an b2n, where n>=1 and will reject all other strings.

Compare and contrast DPDA and NPDA

DPDA (Deterministic Pushdown Automata) and NPDA (Non-deterministic Pushdown Automata) are both types of pushdown automata (PDA) that are used to recognize context-free languages. The main difference between the two is that a DPDA is deterministic, meaning that for a given input and current state, there is only one next state and one operation that can be performed, while an NPDA is non-deterministic, meaning that for a given input and current state, there can be multiple next states and/or operations that can be performed.

Here are some key points of comparison between DPDA and NPDA:

* DPDA can only have one transition for a given input symbol and current state, while NPDA can have multiple transitions for the same input symbol and current state.
* DPDA is strictly less powerful than NPDA, i.e. any context-free language that can be recognized by an NPDA can also be recognized by a DPDA, but the converse is not true. NPDA can recognize languages that are not context-free (such as context-sensitive languages)
* DPDA are easier to construct and understand since there is only one transition for a given input symbol and current state, while NPDA can be more complex since there can be multiple transitions for the same input symbol and current state.
* DPDA can be simulated by an NPDA by adding a new state for each possible transition, resulting in a single path through the automata, hence deterministic.
* DPDA and NPDA both use a stack to store information, but DPDA can only pop one symbol from the stack while reading one input symbol while NPDA can pop multiple symbols or push multiple symbols in one transition.

Overall, both DPDA and NPDA are powerful tools for recognizing context-free languages, but NPDA is more powerful than DPDA and can be used to recognize a wider range of languages.

Distinguish between regular languages and recursively enumerable languages

Regular languages and recursively enumerable (RE) languages are both types of formal languages, but they have some distinct characteristics that set them apart.

A regular language is a language that can be recognized by a finite automaton, which is a simple machine that reads input symbols one at a time and changes its internal state based on a set of transition rules. Regular languages are also known as "type 3 languages" and they are closed under regular operations such as union, concatenation and Kleene star.

On the other hand, recursively enumerable languages are languages that can be recognized by a Turing machine, which is a more powerful type of machine that can read and write symbols on an infinite tape. Recursively enumerable languages are also known as "type 0 languages" and they are closed under computable operations such as union and concatenation but not under the Kleene star.

Here are some key points of distinction between regular languages and recursively enumerable languages:

* Regular languages are recognized by a finite automaton, while recursively enumerable languages are recognized by a Turing machine.
* Regular languages are closed under regular operations such as union, concatenation and Kleene star, while recursively enumerable languages are closed under computable operations such as union and concatenation but not under the Kleene star
* Regular languages are a strict subset of recursively enumerable languages, i.e. all regular languages are recursively enumerable but not all recursively enumerable languages are regular.
* Regular languages are simpler to understand and work with, while recursively enumerable languages are more powerful and can recognize a wider range of languages.
* Every context-free language is recursively enumerable but not all recursively enumerable languages are context-free.

In summary, regular languages are a subset of recursively enumerable languages, and they are recognized by a simpler machine (finite automata) while recursively enumerable languages are recognized by a more powerful machine (Turing machine) and they can recognize more complex languages.

Explain Chomsky Hierarchy of languages

The Chomsky hierarchy is a classification system for formal languages that was developed by Noam Chomsky. It organizes formal languages into four classes, each class having a greater expressive power than the one before it:

1. Type-0 languages (Recursively Enumerable languages or RE languages): These are the most general class of languages and include all languages that can be recognized by a Turing machine. These languages are also known as recursively enumerable languages.
2. Type-1 languages (Context-sensitive languages): These languages are a subset of type-0 languages and can be recognized by a Linear bounded automata. These languages are also known as context-sensitive languages.
3. Type-2 languages (Context-free languages): These languages are a subset of type-1 languages and can be recognized by a pushdown automata (PDA). These languages are also known as context-free languages.
4. Type-3 languages (Regular languages): These languages are a subset of type-2 languages and can be recognized by a finite automaton. These languages are also known as regular languages.

The Chomsky hierarchy is based on the idea that the more powerful a machine is, the more complex the languages it can recognize. For example, a finite automaton can only recognize regular languages, which are the simplest class of languages, while a Turing machine can recognize all recursively enumerable languages, which are the most general class of languages.

It's worth noting that the languages of each class are closed under the respective operations of their class, for example regular languages are closed under regular operations such as union, concatenation and Kleene star, while recursively enumerable languages are closed under computable operations such as union and concatenation but not under the Kleene star.

In summary, the Chomsky hierarchy is a classification system for formal languages that separates them into four classes based on the machine that can recognize them and their expressive power.

Design Turing machine to accept all set of palindromes over {0, 1}\*

A Turing machine to accept all set of palindromes over {0, 1}\* would have the following properties:

* The machine starts in a designated "start" state.
* The machine reads the input string from left to right and compares each character to its corresponding character on the right side of the string.
* If the characters match, the machine moves to the next pair of characters.
* If the characters do not match, the machine moves to a designated "reject" state.
* If the machine reaches the middle of the input string (i.e. all characters have been successfully matched), it moves to a designated "accept" state.

Here is one possible design of the Turing machine's transition function:

current state | current symbol | next state | next symbol | direction

----------------------------------------------------------------------

start | 0 or 1 | compare | \_ | left

compare | 0 or 1 | compare | \_ | left

compare | \_ | accept | \_ | \_

start | \_ | reject | \_ | \_

Copy code

current state | current symbol | next state | next symbol | direction ---------------------------------------------------------------------- start | 0 or 1 | compare | \_ | left compare | 0 or 1 | compare | \_ | left compare | \_ | accept | \_ | \_ start | \_ | reject | \_ | \_

Here, the machine starts in the "start" state and reads the input symbol from left to right. Then it moves to the "compare" state and reads the input from the right side of the string. If the current symbol matches with the corresponding symbol from the left side of the input string, it stays in the "compare" state and move towards left. If it finds any mismatches it moves to the "reject" state. Once it reaches the middle of the input string it moves to "accept" state.

This is one possible design and it can be improved depending on the requirement.

An example of the instantaneous description of the Turing machine on the input string "1001" would be:

| **Current State** | **Current Symbol** | **Next State** | **Next Symbol** | **Direction** |
| --- | --- | --- | --- | --- |
| start | 1 | compare | 1 | left |
| compare | 0 | compare | 0 | left |
| compare | 1 | compare | 1 | left |
| compare | \_ | accept | \_ | \_ |

Here, the machine starts in the "start" state and reads the input symbol from left to right, then it moves to the "compare" state, reads the input from the right side of the string and found that it's matching the corresponding symbol from the left side of the input string, so it stays in the "compare" state, moves towards left and reaches the middle of the input string, so it moves to "accept" state.

also write the instantaneous description on the string 1001001

Here is one possible instantaneous description of the Turing machine running on the input string "1001001":

1. The machine starts in the "start" state and reads the first symbol, which is "1".
2. The machine moves to the next symbol, which is also "1". Since these symbols match, the machine stays in the "start" state and moves to the next pair of symbols.
3. The machine reads the next symbol, which is "0". Since this does not match the corresponding symbol on the right side of the input string, the machine moves to the "reject" state.

So, the machine would be rejected when it reads the symbol '0' on the 4th step.

Here is the instantaneous description of the machine:

| **Current State** | **Current Symbol** | **Next State** | **Next Symbol** | **Direction** |
| --- | --- | --- | --- | --- |
| start | 1 | start | 1 | right |
| start | 1 | start | 1 | right |
| start | 0 | reject | \_ | \_ |

Note: \_ denotes the empty symbol

As the machine reaches the "reject" state, it will stop and the input string "1001001" will not be accepted as it's not a palindrome.

Construct a PDA that recognizes strings (over alphabet 0 and 1) that contain equal number of 0s and 1s.

A PDA (Pushdown Automaton) that recognizes strings (over alphabet 0 and 1) that contain equal number of 0s and 1s can be designed as follows:

* The machine starts in an initial state "q0".
* The machine reads the input string from left to right.
* The machine maintains a stack to keep track of the number of 0s and 1s that have been read so far.
* The machine has two stack operations: "push" and "pop".
* The machine has two types of transitions: "0-transitions" and "1-transitions".
* The machine has two final states: "q1" and "q2", where "q1" is the accept state and "q2" is the reject state.

Here is one possible design of the PDA's transition function:

current state | current symbol | stack operation | next state | next symbol

----------------------------------------------------------------------

q0 | 0 | push 0 | q0 | 0

q0 | 1 | push 1 | q0 | 1

q0 | \_ | pop | q1 | \_

q1 | \_ | \_ | q2 | \_

Copy code

current state | current symbol | stack operation | next state | next symbol ---------------------------------------------------------------------- q0 | 0 | push 0 | q0 | 0 q0 | 1 | push 1 | q0 | 1 q0 | \_ | pop | q1 | \_ q1 | \_ | \_ | q2 | \_

Here, when the machine is in state q0 and it reads a symbol "0", it performs a "push 0" operation on the stack and remains in the same state q0. When it reads a symbol "1", it performs a "push 1" operation on the stack and remains in the same state q0. Once it reaches the end of the input string, it performs a "pop" operation on the stack and if the stack is empty it moves to state q1, which is the accept state. Otherwise it moves to state q2, which is the reject state.

An example of the instantaneous description of the PDA on the input string "0101" would be:

Step | Current State | Current Symbol | Stack | Next State | Next Symbol

----------------------------------------------------------------------

1 | q0 | 0 | [0] | q0 | 0

2 | q0 | 1 | [0,1] | q0 | 1

3 | q0 | 0 | [0] | q0 | 0

4 | q0 | 1 | [] | q1 | \_

Copy code

Step | Current State | Current Symbol | Stack | Next State | Next Symbol ---------------------------------------------------------------------- 1 | q0 | 0 | [0] | q0 | 0 2 | q0 | 1 | [0,1] | q0 | 1 3 | q0 | 0 | [0] | q0 | 0 4 | q0 | 1 | [] | q1 | \_

Here, the machine starts in the "q0" state, reads the input symbol "0" and performs a "push 0" operation on the stack. Then it reads the next symbol "1", performs a "push 1" operation on the stack. After that it reads the next symbol "0" and performs a "push 0" operation on the stack. Finally, it reads the last symbol "1" and performs a "pop" operation on the stack and finds that the stack is empty, so it moves to state "q1" which is the accept state.

It's worth noting that this is one possible design and it can be improved depending on the requirement.