Artificial Intelligence Notes

AI

Thinking Humanly	Thinking Rationally
"The exciting new effort to make computers think . . . *machines with minds*, in the full and literal sense." (Haugeland, 1985) "[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning . . ." (Bellman, 1978)	"The study of mental faculties through the use of computational models." (Charniak and McDermott, 1985) "The study of the computations that make it possible to perceive, reason, and act." (Winston, 1992)
Acting Humanly	**Acting Rationally**
"The art of creating machines that perform functions that require intelligence when performed by people." (Kurzweil, 1990) "The study of how to make computers do things at which, at the moment, people are better." (Rich and Knight, 1991)	"Computational Intelligence is the study of the design of intelligent agents." (Poole *et al.*, 1998) "AI . . . is concerned with intelligent behavior in artifacts." (Nilsson, 1998)

Figure 1.1 Some definitions of artificial intelligence, organized into four categories.

Chapter 2: Intelligent Agent

An **agent** is anything that can be viewed as perceiving its **environment** through **sensors** and acting upon that environment through **actuators**.

A human agent has eyes, ears, and other organs for sensors and hands, legs, vocal tract, and so on for actuators.

A robotic agent might have cameras and infrared range finders for sensors and various motors for actuators.

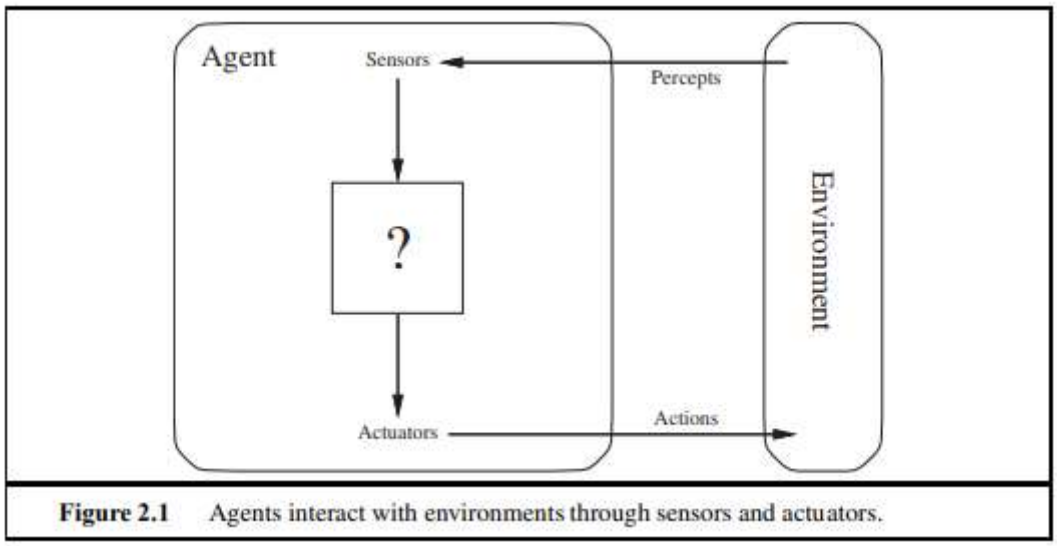


Figure 2.1 Agents interact with environments through sensors and actuators.

Percept to refer to the agent's perceptual inputs at any given instant.

An agent's **percept sequence** is the complete history of everything the agent has ever perceived.

An agent's choice of action at any given instant can depend on the entire percept sequence observed to date, but not on anything it hasn't perceived.

By specifying the agent's choice of action for every possible percept sequence, we have said more or less everything there is to say about the agent. Mathematically speaking, we say that an agent's behaviour is described by the **agent function** that maps any given percept sequence to an action.

We can imagine tabulating the agent function that describes any given agent; for most agents, this would be a very large table—infinite, in fact, unless we place a bound on the length of percept sequences we want to consider. Given an agent to experiment with, we can, in principle, construct this table by trying out all possible percept sequences and recording which actions the agent does in response.

Internally, the agent function for an artificial agent will be implemented by an **agent program**.

Example - the vacuum-cleaner world

This world is so simple that we can describe everything that happens; it's also a made-up world, so we can invent many variations. This particular world has just two locations: squares A and B. The vacuum agent perceives which square it is in and whether there is dirt in the square. It can choose to move left, move right,

suck up the dirt, or do nothing. One very simple agent function is the following: if the current square is dirty, then suck; otherwise, move to the other square

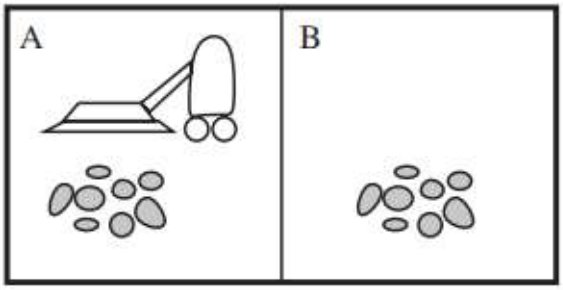


Figure 2.2 A vacuum-cleaner world with just two locations.

Percept sequence	Action
$[A, Clean]$	*Right*
$[A, Dirty]$	*Suck*
$[B, Clean]$	*Left*
$[B, Dirty]$	*Suck*
$[A, Clean], [A, Clean]$	*Right*
$[A, Clean], [A, Dirty]$	*Suck*
⋮	⋮
$[A, Clean], [A, Clean], [A, Clean]$	*Right*
$[A, Clean], [A, Clean], [A, Dirty]$	*Suck*
⋮	⋮

Figure 2.3 Partial tabulation of a simple agent function for the vacuum-cleaner world shown in Figure 2.2.

RATIONAL AGENT

A **rational agent** is one that does the right thing—conceptually speaking, every entry in the table for the agent function is filled out correctly.

When an agent is plunked down in an environment, it generates a sequence of actions according to the percepts it receives. This sequence of actions causes the environment to go through a sequence of states. If the sequence is desirable, then the agent has performed well. This notion of desirability is captured by a **performance measure** that evaluates any given sequence of environment states.

Rationality

What is rational at any given time depends on four things:

• The performance measure that defines the criterion of success.

• The agent's prior knowledge of the environment.

• The actions that the agent can perform.

• The agent's percept sequence to date.

This leads to a **definition of a rational agent**.

Omniscience

An omniscient agent knows the actual outcome of its actions and can act accordingly; but omniscience is impossible in reality.

Doing actions in order to modify future percepts—sometimes called **information gathering**—is an important part of rationality.

A second example of information gathering is provided by the **exploration** that must be undertaken by a vacuum-cleaning agent in an initially unknown environment.

AUTONOMY

To the extent that an agent relies on the prior knowledge of its designer rather than on its own percepts, we say that the agent lacks **autonomy**.

Specifying the task environment

In our discussion of the rationality of the simple vacuum-cleaner agent, we had to specify the performance measure, the environment, and the agent's actuators and sensors. We group all these under the heading of the task environment. For the acronymically minded, we call **PEAS** this the **PEAS** (**Performance**, **Environment**, **Actuators**, **Sensors**) description.

Agent Type	Performance Measure	Environment	Actuators	Sensors
Taxi driver	Safe, fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers	Steering, accelerator, brake, signal, horn, display	Cameras, sonar, speedometer, GPS, odometer, accelerometer, engine sensors, keyboard

Figure 2.4 PEAS description of the task environment for an automated taxi.

Properties of task environments

Fully observable vs. partially observable

If an agent's sensors give it access to the complete state of the environment at each point in time, then we say that the task environment is **fully observable**.

A task environment is effectively **fully observable** if the sensors detect all aspects that are relevant to the choice of action; relevance, in turn, depends on the performance measure.

An environment might be **partially observable** because of noisy and inaccurate sensors or because parts of the state are simply missing from the sensor data.

INTRODUCTION TO GAME PLAYING

Game Playing is an important domain of artificial intelligence. Games don't require much knowledge; the only knowledge we need to provide is the rules, legal moves and the conditions of winning or losing the game.

Both players try to win the game. So, both of them try to make the best move possible at each turn. Searching techniques like BFS(Breadth First Search) are not accurate for this as the branching factor is very high, so searching will take a lot of time. So, we need another search procedures that improve –

- **Generate procedure** so that only good moves are generated.
- **Test procedure** so that the best move can be explored first.

The most common search technique in game playing is Minimax search procedure. It is depth-first depth-limited search procedure. It is used for games like chess and tic-tac-toe.

Minimax algorithm uses two functions –

MOVEGEN : It generates all the possible moves that can be generated from the current position.

STATICEVALUATION : It returns a value depending upon the goodness from the viewpoint of two-player

This algorithm is a two players game, so we call the first player as PLAYER1 and second player as PLAYER2. The value of each node is backed-up from its children. For PLAYER1 the backed-up value is the maximum value of its children and for

PLAYER2 the backed-up value is the minimum value of its children. It provides most promising move to PLAYER1, assuming that the PLAYER2 has make the best move. It is a recursive algorithm, as same procedure occurs at each level.

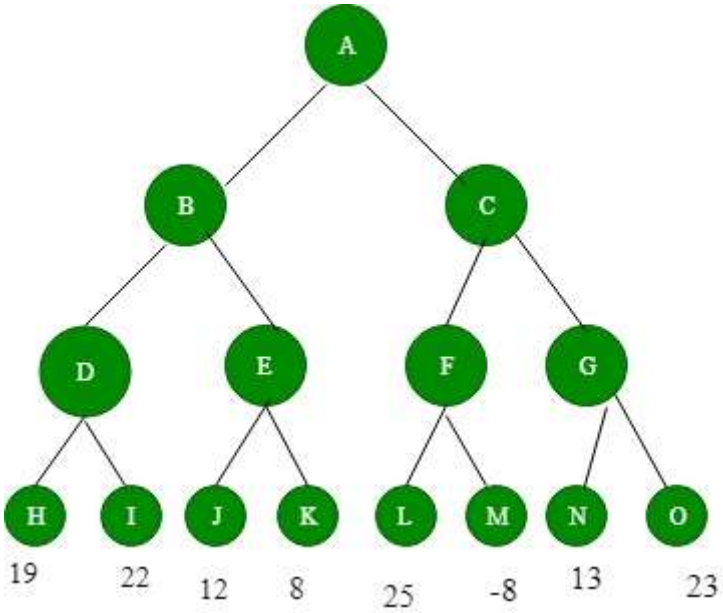


Figure 1: Before backing-up of values

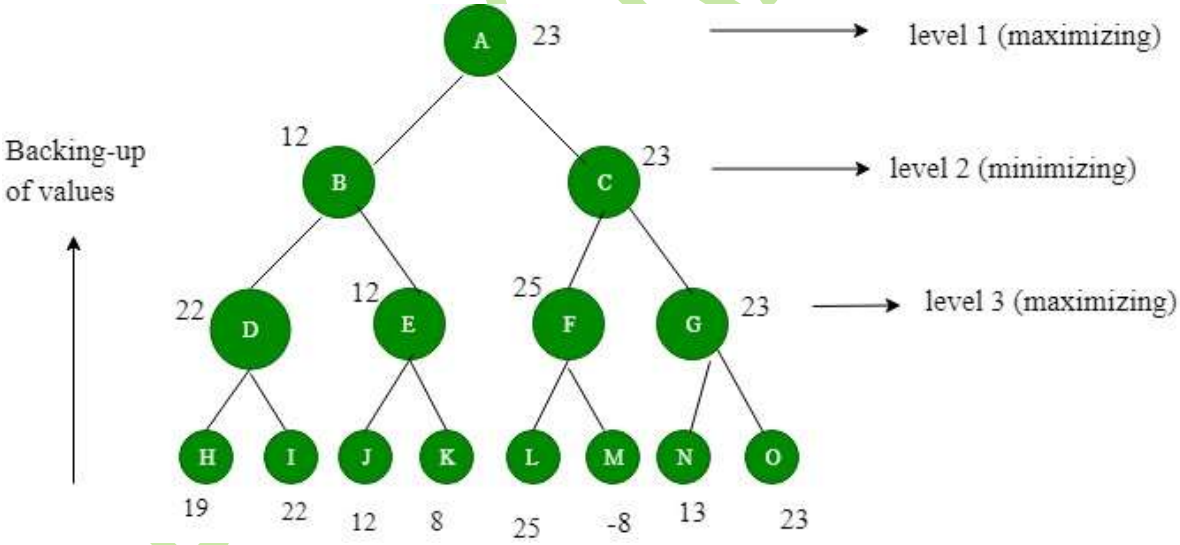


Figure 2: After backing-up of values

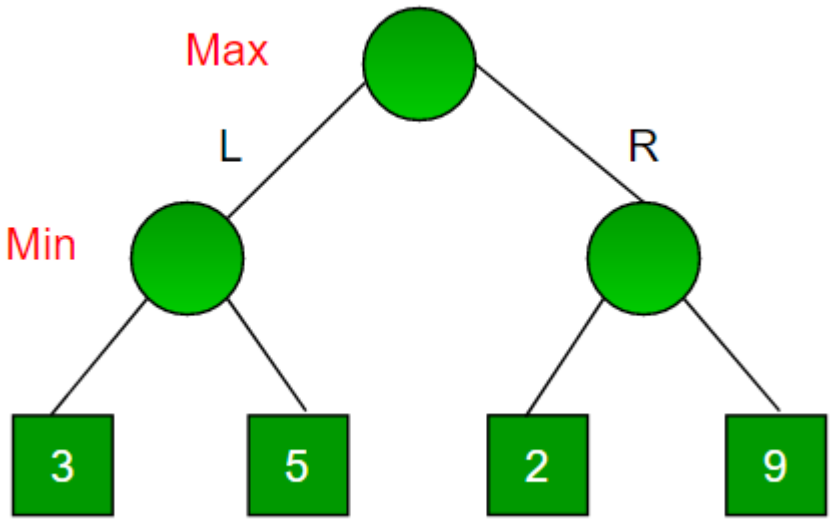
We assume that PLAYER1 will start the game. 4 levels are generated. The value to nodes H, I, J, K, L, M, N, O is provided by STATICEVALUATION function. Level 3 is maximizing level, so all nodes of level 3 will take maximum values of their children. Level 2 is minimizing level, so all its nodes will take minimum values of

their children. This process continues. The value of A is 23. That means A should choose C move to win.

MINIMAX

Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. It is widely used in two player turn-based games such as Tic-Tac-Toe, Backgammon, Mancala, Chess, etc.
In Minimax the two players are called maximizer and minimizer. The **maximizer** tries to get the highest score possible while the **minimizer** tries to do the opposite and get the lowest score possible.
Every board state has a value associated with it. In a given state if the maximizer has upper hand then, the score of the board will tend to be some positive value. If the minimizer has the upper hand in that board state then it will tend to be some negative value. The values of the board are calculated by some heuristics which are unique for every type of game.

Example:
Consider a game which has 4 final states and paths to reach final state are from root to 4 leaves of a perfect binary tree as shown below. Assume you are the maximizing player and you get the first chance to move, i.e., you are at the root and your opponent at next level. **Which move you would make as a maximizing player considering that your opponent also plays optimally?**

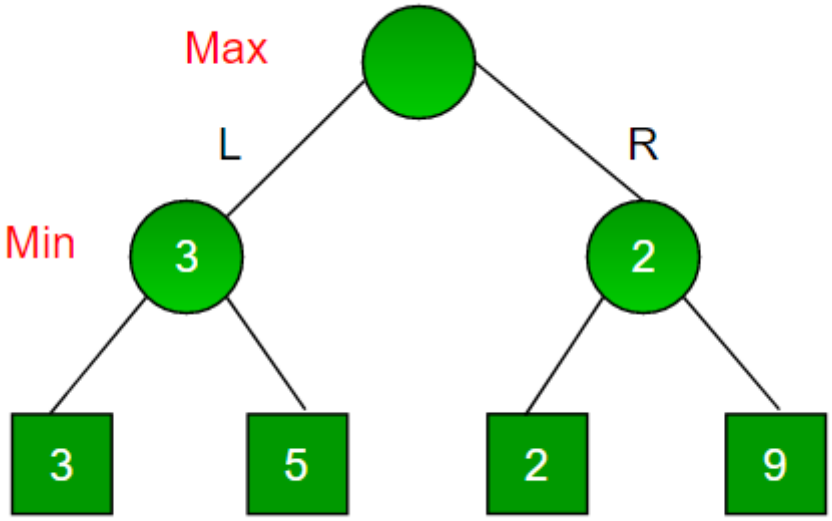


Since this is a backtracking based algorithm, it tries all possible moves, then backtracks and makes a decision.

- Maximizer goes LEFT: It is now the minimizers turn. The minimizer now has a choice between 3 and 5. Being the minimizer it will definitely choose the least among both, that is 3
- Maximizer goes RIGHT: It is now the minimizers turn. The minimizer now has a choice between 2 and 9. He will choose 2 as it is the least among the two values.

Being the maximizer you would choose the larger value that is 3. Hence the optimal move for the maximizer is to go LEFT and the optimal value is 3.

Now the game tree looks like below :



The idea of this article is to introduce Minimax with a simple example.

- In the above example, there are only two choices for a player. In general, there can be more choices. In that case, we need to recur for all possible moves and find the maximum/minimum. For example, in Tic-Tac-Toe, the first player can make 9 possible moves.
- In the above example, the scores (leaves of Game Tree) are given to us. For a typical game, we need to derive these values.

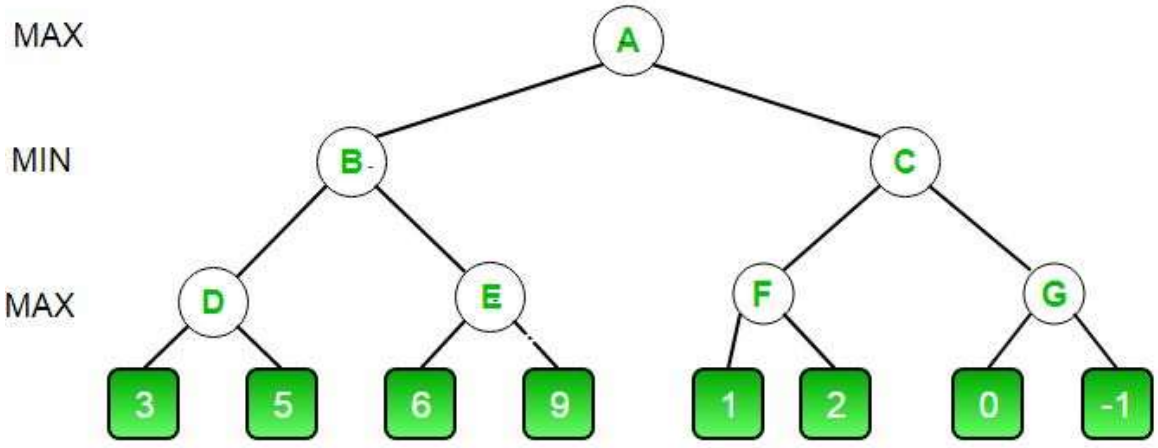
ALPHA-BETA PRUNING

Alpha-Beta pruning is not actually a new algorithm, but rather an optimization technique for the minimax algorithm. It reduces the computation time by a huge factor. This allows us to search much faster and even go into deeper levels in the game tree. It cuts off branches in the game tree which need not be searched because there already exists a better move available. It is called Alpha-Beta pruning because it passes 2 extra parameters in the minimax function, namely alpha and beta.

Let's define the parameters alpha and beta.

Alpha is the best value that the **maximizer** currently can guarantee at that level or above.
Beta is the best value that the **minimizer** currently can guarantee at that level or below.

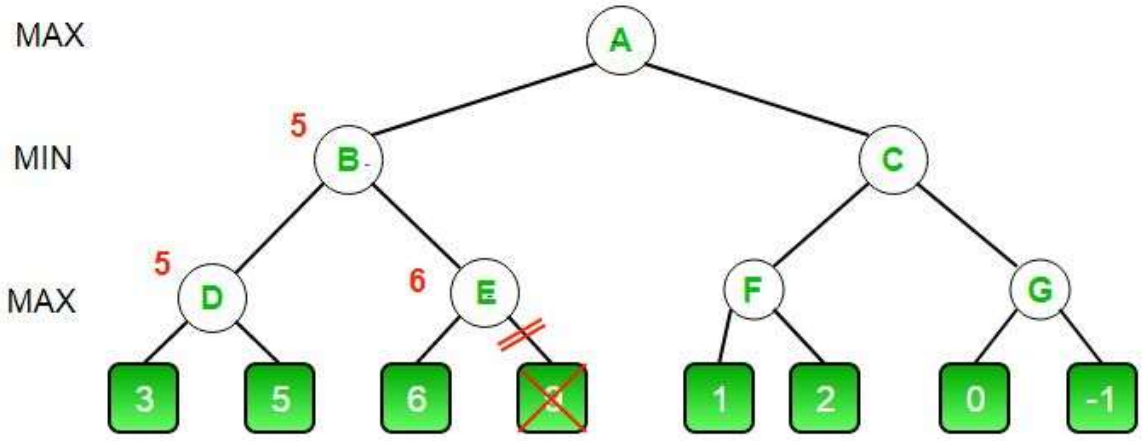
Let's make the above algorithm clear with an example.



- The initial call starts from **A**. The value of alpha here is **-INFINITY** and the value of beta is **+INFINITY**. These values are passed down to subsequent nodes in the tree. At **A** the maximizer must choose max of **B** and **C**, so **A** calls **B** first
- At **B** it the minimizer must choose min of **D** and **E** and hence calls **D** first.
- At **D**, it looks at its left child which is a leaf node. This node returns a value of 3. Now the value of alpha at **D** is max(-INF, 3) which is 3.

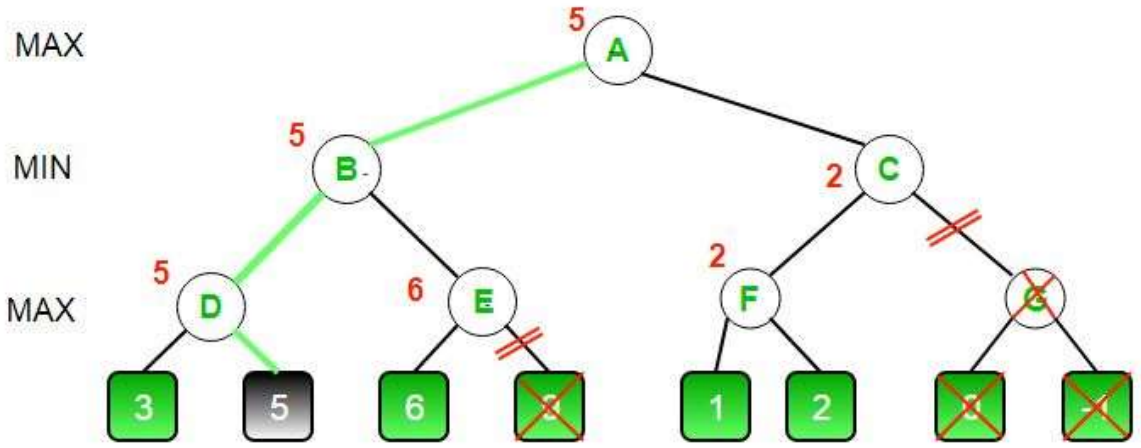
- To decide whether its worth looking at its right node or not, it checks the condition beta<=alpha. This is false since beta = +INF and alpha = 3. So it continues the search.
- **D** now looks at its right child which returns a value of 5.At **D**, alpha = max(3, 5) which is 5. Now the value of node **D** is 5
- **D** returns a value of 5 to **B**. At **B**, beta = min(+INF, 5) which is 5. The minimizer is now guaranteed a value of 5 or lesser. **B** now calls **E** to see if he can get a lower value than 5.
- At **E** the values of alpha and beta is not -INF and +INF but instead -INF and 5 respectively, because the value of beta was changed at **B** and that is what **B** passed down to **E**
- Now **E** looks at its left child which is 6. At **E**, alpha = max(-INF, 6) which is 6. Here the condition becomes true. beta is 5 and alpha is 6. So beta<=alpha is true. Hence it breaks and **E** returns 6 to **B**
- Note how it did not matter what the value of **E**'s right child is. It could have been +INF or -INF, it still wouldn't matter, We never even had to look at it because the minimizer was guaranteed a value of 5 or lesser. So as soon as the maximizer saw the 6 he knew the minimizer would never come this way because he can get a 5 on the left side of **B**. This way we didn't have to look at that 9 and hence saved computation time.
- **E** returns a value of 6 to **B**. At **B**, beta = min(5, 6) which is 5.The value of node **B** is also 5

So far this is how our game tree looks. The 9 is crossed out because it was never computed.



- **B** returns 5 to **A**. At **A**, alpha = max(-INF, 5) which is 5. Now the maximizer is guaranteed a value of 5 or greater. **A** now calls **C** to see if it can get a higher value than 5.
- At **C**, alpha = 5 and beta = +INF. **C** calls **F**
- At **F**, alpha = 5 and beta = +INF. **F** looks at its left child which is a 1. alpha = max(5, 1) which is still 5.
- **F** looks at its right child which is a 2. Hence the best value of this node is 2. Alpha still remains 5
- **F** returns a value of 2 to **C**. At **C**, beta = min(+INF, 2). The condition beta <= alpha becomes true as beta = 2 and alpha = 5. So it breaks and it does not even have to compute the entire sub-tree of **G**.
- The intuition behind this break-off is that, at **C** the minimizer was guaranteed a value of 2 or lesser. But the maximizer was already guaranteed a value of 5 if he choose **B**. So why would the maximizer ever choose **C** and get a value less than 2 ? Again you can see that it did not matter what those last 2 values were. We also saved a lot of computation by skipping a whole sub-tree.
- **C** now returns a value of 2 to **A**. Therefore the best value at **A** is max(5, 2) which is a 5.
- Hence the optimal value that the maximizer can get is 5

This is how our final game tree looks like. As you can see **G** has been crossed out as it was never computed.



FUZZY LOGIC

The term **fuzzy** refers to things that are not clear or are vague. In the real world many times we encounter a situation when we can't determine whether the state is true or false, their fuzzy logic provides very valuable flexibility for reasoning. In this way, we can consider the inaccuracies and uncertainties of any situation.

Fuzzy Logic is a form of many-valued logic in which the truth values of variables may be any real number between 0 and 1, instead of just the traditional values of true or false. It is used to deal with imprecise or uncertain information and is a mathematical method for representing vagueness and uncertainty in decision-making.

Fuzzy Logic is based on the idea that in many cases, the concept of true or false is too restrictive, and that there are many shades of gray in between. It allows for partial truths, where a statement can be partially true or false, rather than fully true or false.

Fuzzy Logic is used in a wide range of applications, such as control systems, image processing, natural language processing, medical diagnosis, and artificial intelligence.

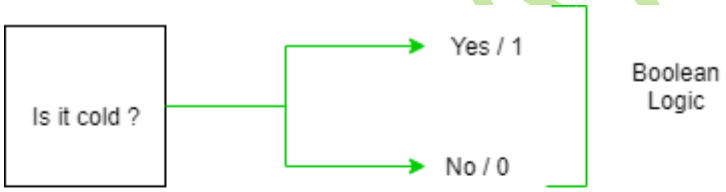
The fundamental concept of Fuzzy Logic is the membership function, which defines the degree of membership of an input value to a certain set or category. The membership function is a mapping from an input value to a membership

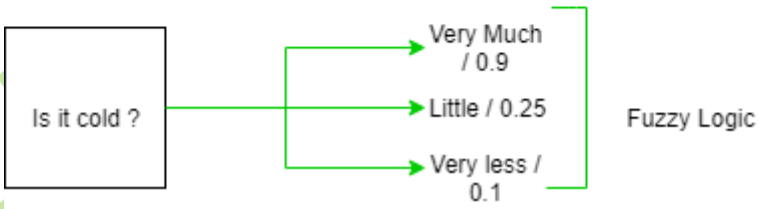
degree between 0 and 1, where 0 represents non-membership and 1 represents full membership.

Fuzzy Logic is implemented using Fuzzy Rules, which are if-then statements that express the relationship between input variables and output variables in a fuzzy way. The output of a Fuzzy Logic system is a fuzzy set, which is a set of membership degrees for each possible output value.

In summary, Fuzzy Logic is a mathematical method for representing vagueness and uncertainty in decision-making, it allows for partial truths, and it is used in a wide range of applications. It is based on the concept of membership function and the implementation is done using Fuzzy rules.

In the boolean system truth value, 1.0 represents the absolute truth value and 0.0 represents the absolute false value. But in the fuzzy system, there is no logic for the absolute truth and absolute false value. But in fuzzy logic, there is an intermediate value too present which is partially true and partially false.





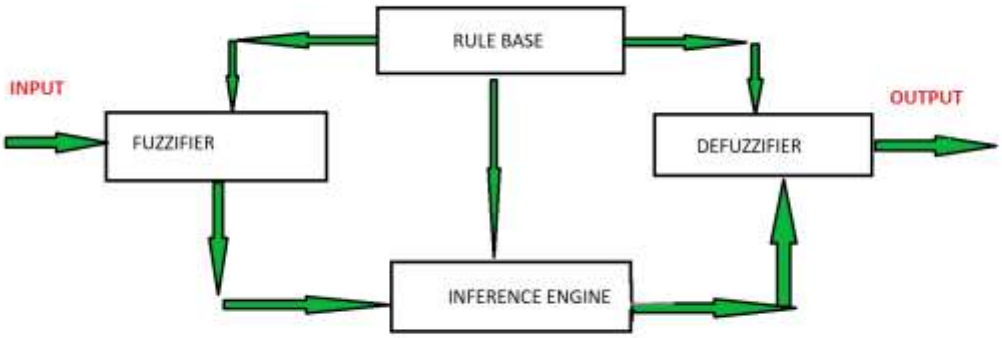
ARCHITECTURE

Its Architecture contains four parts :

- RULE BASE: It contains the set of rules and the IF-THEN conditions provided by the experts to govern the decision-making system, on the basis of linguistic information. Recent developments in fuzzy theory offer several effective methods for the design and tuning of fuzzy controllers. Most of these developments reduce the number of fuzzy rules.
- FUZZIFICATION: It is used to convert inputs i.e. crisp numbers into fuzzy sets. Crisp inputs are basically the exact inputs measured by sensors and

passed into the control system for processing, such as temperature, pressure, rpm's, etc.

- INFERENCE ENGINE: It determines the matching degree of the current fuzzy input with respect to each rule and decides which rules are to be fired according to the input field. Next, the fired rules are combined to form the control actions.
- DEFUZZIFICATION: It is used to convert the fuzzy sets obtained by the inference engine into a crisp value. There are several defuzzification methods available and the best-suited one is used with a specific expert system to reduce the error.



FUZZY LOGIC ARCHITECTURE

Noam Chomsky Hierarchy

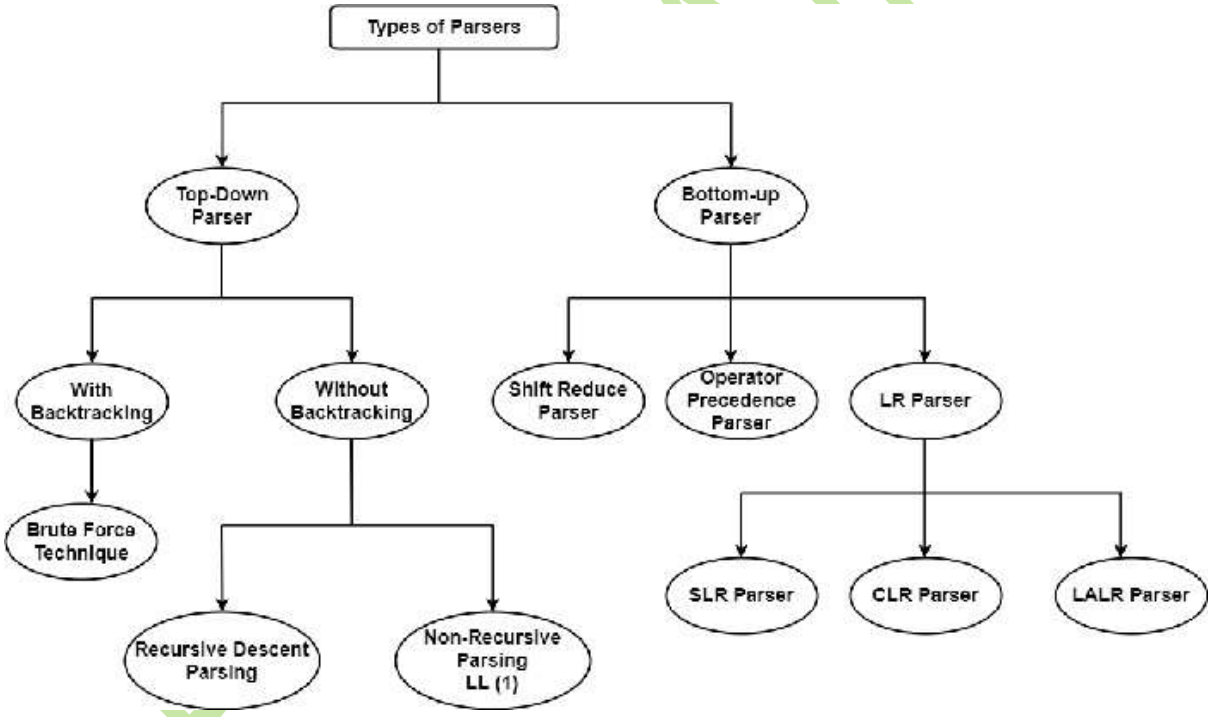
According to Noam Chomsky, there are four types of grammars − Type 0, Type 1, Type 2, and Type 3.

Noam Chomsky have mathematical model of grammar which is efficient for writing computer knowledge.

Grammar Type	Grammar Accepted	Language Accepted	Automaton
Type 0	Unrestricted grammar	Recursively enumerable language	Turing Machine

Type 1	Context-sensitive grammar	Context-sensitive language	Linear-bounded automaton
Type 2	Context-free grammar	Context-free language	Pushdown automaton
Type 3	Regular grammar	Regular language	Finite state automaton

Parsing is known as Syntax Analysis. It contains arranging the tokens as source code into grammatical phases that are used by the compiler to synthesis output generally grammatical phases of the source code are defined by parse tree. There are various types of parsing techniques which are as follows –



- **Top-Down Parser**

It generates the Parse Tree from root to leaves. In top-down parsing, the parsing begins from the start symbol and changes it into the input symbol.

An example of a Top-Down Parser is Predictive Parsers, Recursive Descent Parser.

Predictive Parser − Predictive Parser is also known as Non-Recursive Predictive Parsing. A predictive parser is an effective approach of implementing recursive descent parsing by manipulating the stack of activation records explicitly. The predictive parser has an input, a stack, a parsing table, and an output. The input includes the string to be parsed, followed by $, the right-end marker.

Recursive Descent Parser − A top-down parser that executes a set of recursive procedures to process the input without backtracking is known as recursive-descent parser, and parsing is known as recursive-descent parsing.

- **Bottom-Up Parser**

It generates the Parse Tree from leaves to root for a given input string. In Grammar, the input string will be reduced to the starting symbol.

Example of Bottom-Up Parser is Shift Reduce Parser, Operator Precedence Parser, and LR Parsers.

Shift Reduce Parser − Shift reduce parser is a type of bottom-up parser. It uses a stack to influence the grammar symbols. A parser goes on changing the input symbols onto the stack until a handle comes on the top of the stack. When a handle occurs on the top of the stack, it implements reduction.

Operator Precedence Parser − The shift-reduce parsers can be generated by hand for a small class of grammars. These grammars have the property that no production on the right side is ϵ or has two adjacent non-terminals. Grammar with the latter property is known as operator grammar.

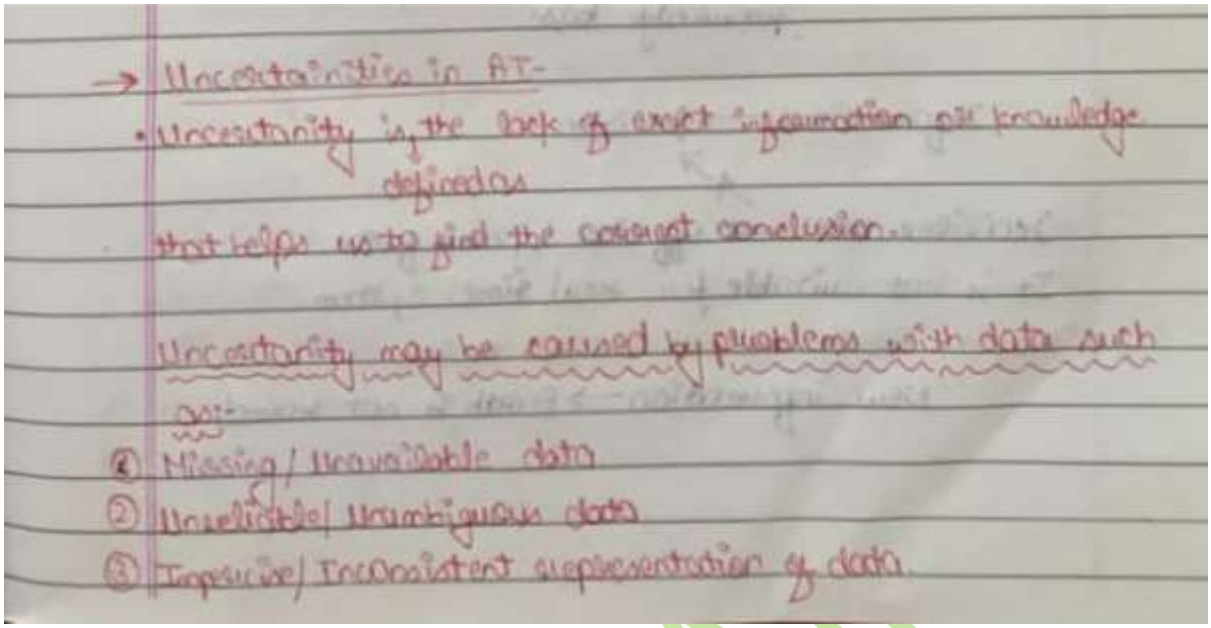
LR Parsers − The LR Parser is a shift-reduce parser that creates use of deterministic finite automata, identifying the set of all viable prefixes by reading the stack from bottom to top. It decides what handle, if any, is available.

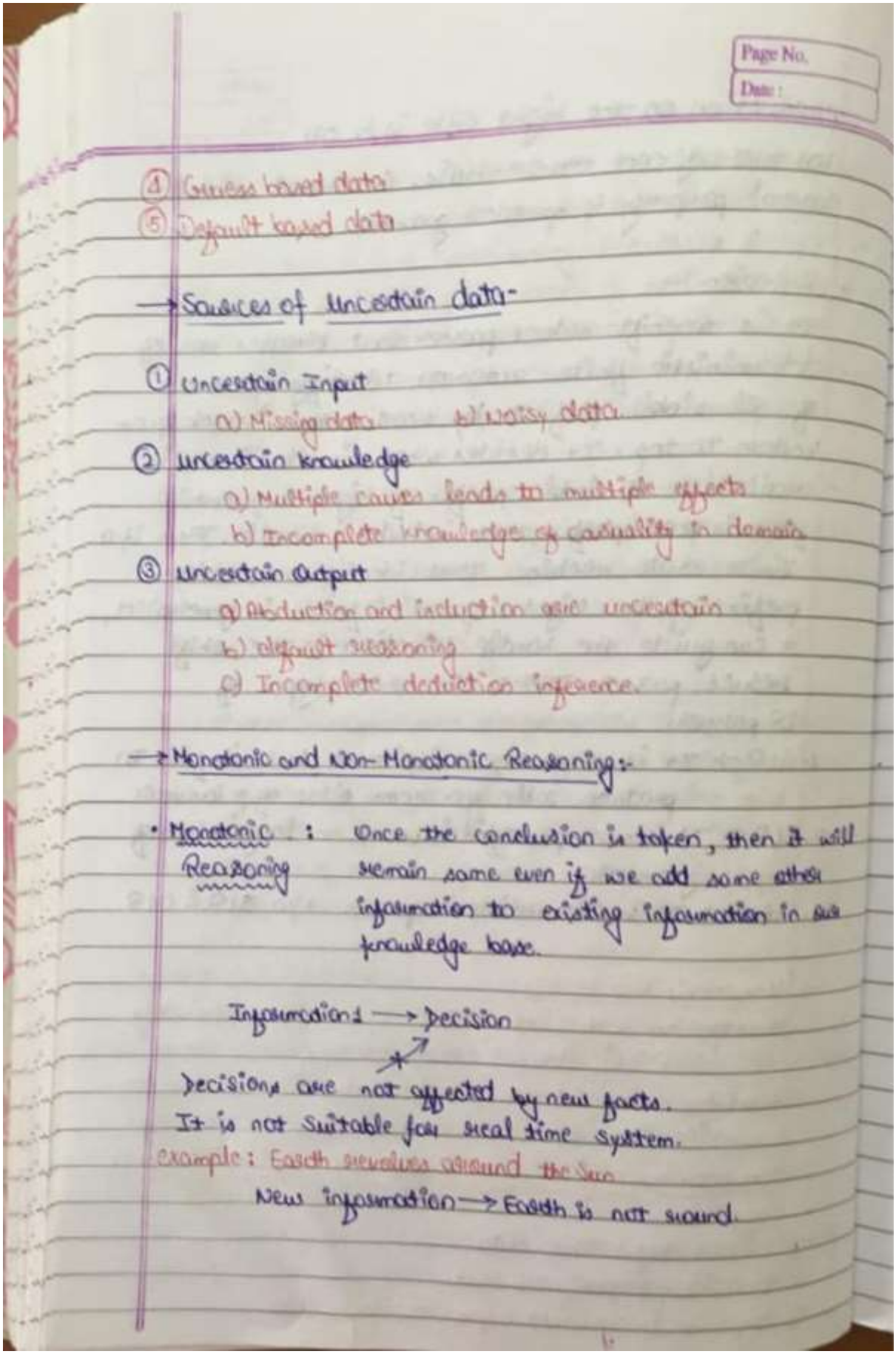
A viable prefix of a right sequential form is that prefix that includes a handle, but no symbol to the right of the handle. Thus, if a finite state machine that identifies viable prefixes of the right sentential form is generated, it can guide the handle selection in the shift-reduce parser.

There are three types of LR Parsers which are as follows −

- **Simple LR Parser (SLR)** − It is very easy to implement but it fails to produce a table for some classes of grammars.
- **Canonical LR Parser (CLR)** − It is the most powerful and works on large classes of grammars.

- **Look Ahead LR Parser (LALR)** – It is intermediate in power between SLR and CLR.





④ Guess based data
⑤ Default based data

→ Sources of Uncertain data-

① Uncertain Input
   a) Missing data b) Noisy data
② Uncertain knowledge
   a) Multiple causes leads to multiple effects
   b) Incomplete knowledge of causality in domain
③ Uncertain Output
   a) Abduction and induction are uncertain
   b) default reasoning
   c) Incomplete deduction inference.

→ Monotonic and Non-Monotonic Reasoning:-

- Monotonic Reasoning : Once the conclusion is taken, then it will remain same even if we add some other information to existing information in our knowledge base.

Information → Decision

Decisions are not affected by new facts.
It is not suitable for real time system.
Example: Earth revolves around the Sun
New information → Earth is not round.

Page No.
Date :

Advantage: All odd proofs are valid.

Disadvantage: ① can not real world scenarios
② New knowledge from the world cannot be added.

- Non Monotonic Reasoning : In this some conclusions may be invalidated if we add some more information to our knowledge base. decision can be changed by new facts.

Information 1 ⟶ decision 1

Information 2 ⟶ decision 2 (×)

Example→ Information decision

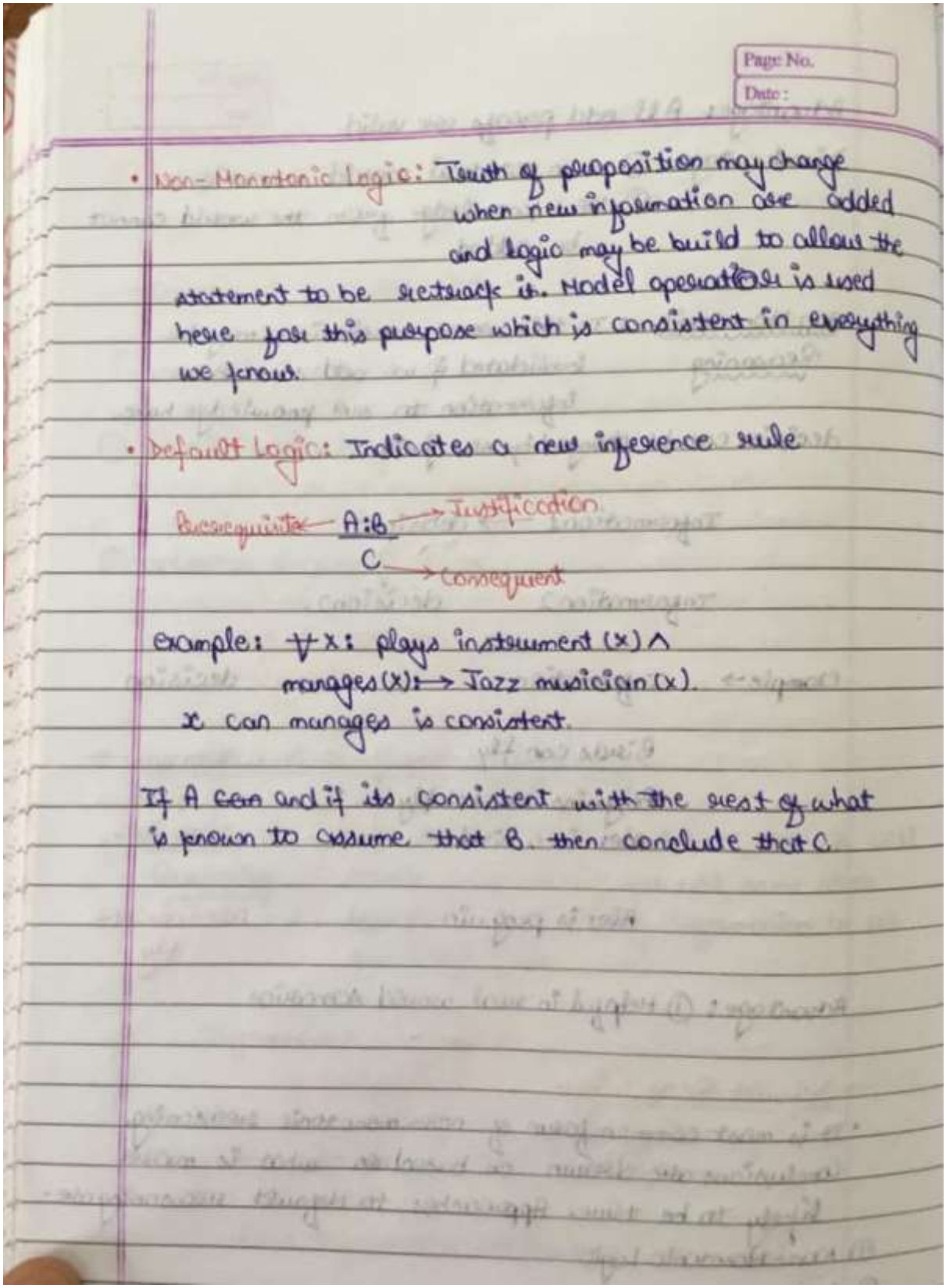
Birds can fly
Penguins can not fly
Alex is a bird. ⟶ Alex can fly

Alex is penguin ⟶ Alex can not fly.

Advantage: ① Helpful in real world scenarios

→ Default Reasoning:-

- It is most common form of non-monotonic reasoning. Conclusions are drawn on based on what is more likely to be true. Approaches to default reasoning are-
① Non-Monotonic Logic
② Default Logic.



- Non-Monotonic logic: Truth of proposition may change when new information are added and logic may be build to allow the statement to be retracted. Model operation is used here for this purpose which is consistent in everything we know.

- Default Logic: Indicates a new inference rule

$$\frac{A : B}{C}$$

Precequisite → A:B → Justification

C → Consequent

Example: $\forall x$: plays instrument (x) $\wedge$ manages (x) $\rightarrow$ Jazz musician (x).

x can manages is consistent.

If A can and if its consistent with the rest of what is known to assume that B. then conclude that C.