ARTIFICIAL INTELLIGENCE

INTERIM SEMESTER 2021-22 BPL

CSE3007-LT-AB306

FACULTY: SIMI V.R.

Adversarial search

- **Competitive** environments
- which the agents’ goals are in conflict
- giving rise to **adversarial search** problems—often known as **games**.
- Games, are interesting *because* they are too hard to solve.
- Optimal move and an algorithm for finding it
- **Pruning** allows us to ignore portions of the search tree that make no difference to the final choice
- **Heuristic evaluation functions** allow us to approximate the true utility of a state without doing a complete search.

Games

A game can be formally defined as a kind of search problem with the following elements:

- S_0: The initial state, which specifies how the game is set up at the start.
- PLAYER(*s*): Defines which player has the move in a state.
- ACTIONS(*s*): Returns the set of legal moves in a state.
- RESULT(*s*, *a*): The transition model, which defines the result of a move.
- TERMINAL-TEST(*s*): A terminal test, which is true when the game is over and false otherwise.
- States where the game has ended are called terminal states.
- UTILITY(*s*, *p*): A utility function (also called an objective function or payoff function),

  defines the final numeric value for a game that ends in terminal state s for a player p.

In chess, the outcome is a win, loss, or draw, with values +1, 0, or . Some games have a wider variety of possible outcomes; the payoffs in backgammon range from 0 to +192.

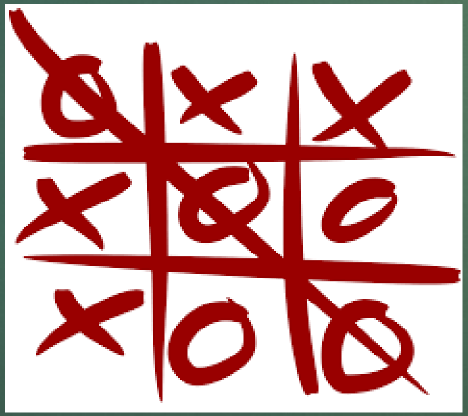
Game tree

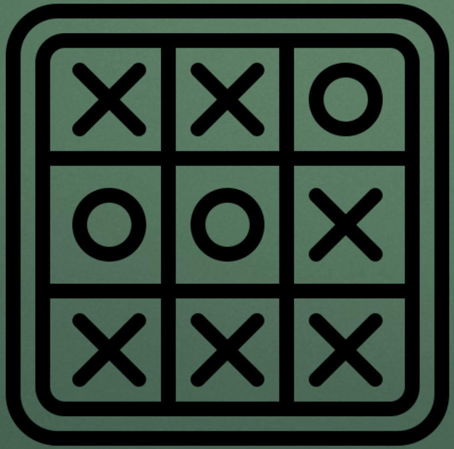
- A **zero-sum game** is defined as one where the total payoff to all players is the same for every instance of the game.
- Chess is zero-sum because every game has payoff of either 0 + 1, 1 + 0 or + .

Game Tree

- The initial state, **ACTIONS** function, and **RESULT** function define the **game tree** for the game.
- The nodes are game states and the edges are moves.

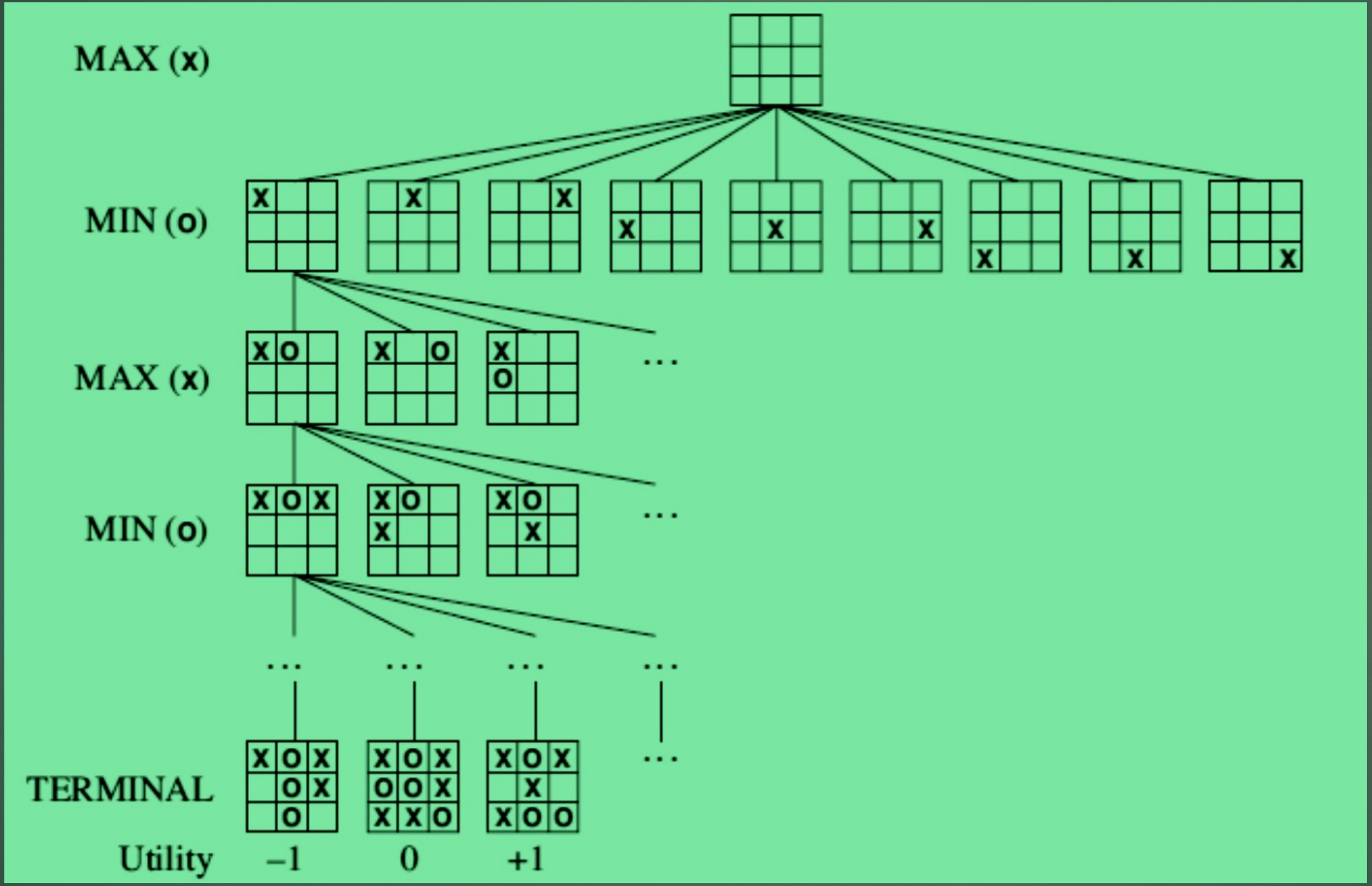
Game tree for tic-tac-toe (noughts and crosses)





Game tree for tic-tac-toe (Cont’d)

- From the initial state, MAX has nine possible moves.
- Play alternates between MAX’s placing an X and MIN’s placing an O until we reach leaf nodes corresponding to terminal states such that one player has three in a row or all the squares are filled.
- The number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are assumed to be good for MAX and bad for MIN (which is how the players get their names).
- For tic-tac-toe the game tree is relatively small—fewer than 9! = 362, 880 terminal nodes.



A (partial) game tree for the game of tic-tac-toe.

- The top node is the initial state, and MAX moves first, placing an X in an empty square.
- We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

Optimal decisions in games

Two-ply game tree

- An optimal strategy leads to outcomes at least as good as any other strategy when one is playing an infallible opponent.
- The possible moves for MAX at the root node are labelled *a1*, *a2*, and *a3*. The possible replies to *a1* for MIN are *b1*, *b2*, *b3*, and so on.
- This particular game ends after one move each by MAX and MIN.
- The utilities of PLY the terminal states in this game range from 2 to 14.

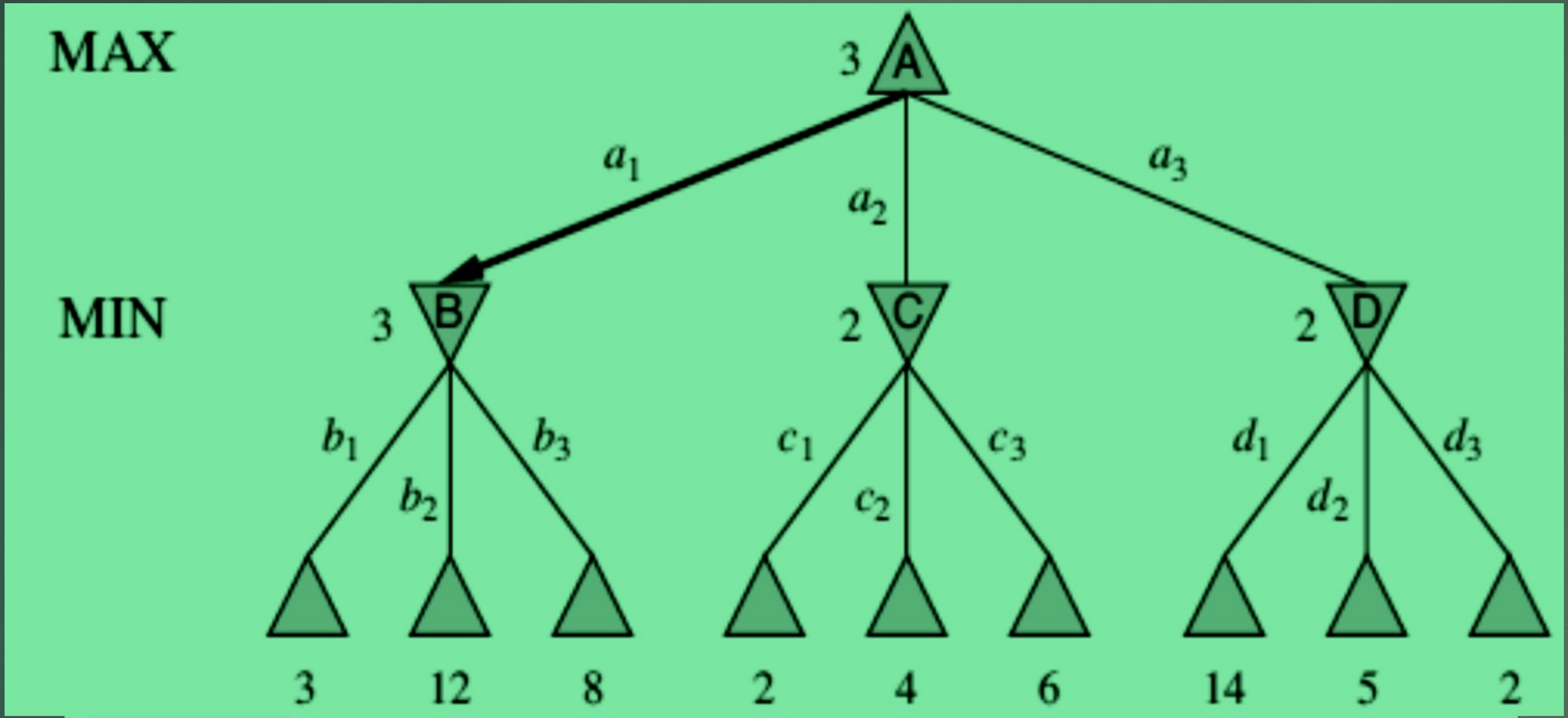
PLY MINIMAX VALUE

- Given a game tree, the optimal strategy can be determined from the **minimax value** of each node, which we write as MINIMAX(*n*).
- The minimax value of a node is the utility (for MAX) of being in the corresponding state, *assuming that both players play optimally* from there to the end of the game.
- The minimax value of a terminal state is just its utility.
- MAX prefers to move to a state of maximum value, whereas MIN prefers a state of minimum value.

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

- MINIMAX algorithm – Back tracking algorithm
- Best move strategy

A two-ply game tree



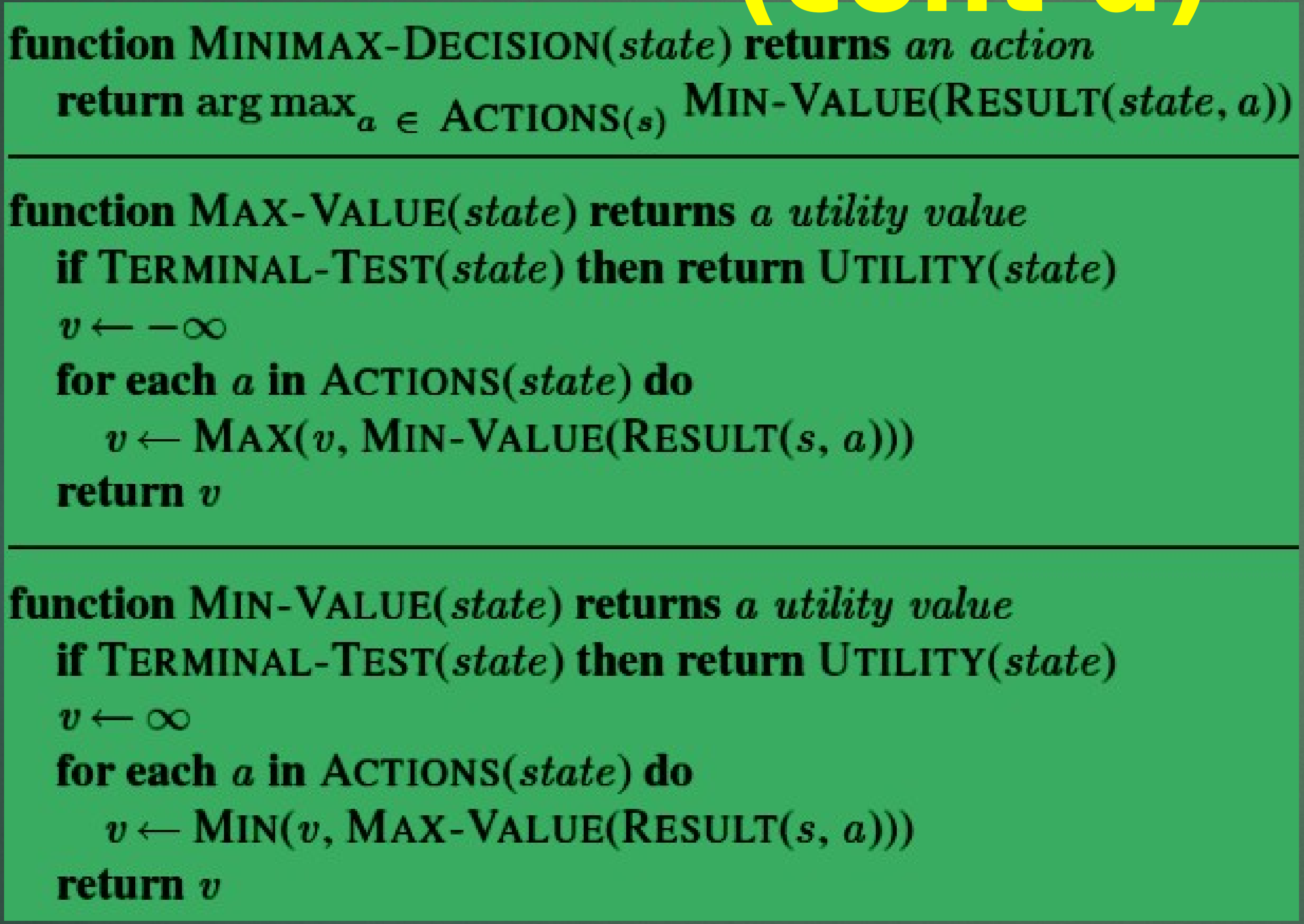
A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

- The terminal nodes on the bottom level get their utility values from the game's UTILITY function.
- The first MIN node, labelled B, has three successor states with values 3, 12, and 8, so its minimax value is 3.
- Similarly, the other two MIN nodes have minimax value 2.
- The root node is a MAX node; its successor states have minimax values 3, 2, and 2; so it has a minimax value of 3.
- MINIMAX DECISION at the root: action a_1 is the optimal choice for MAX because it leads to the state with the highest minimax value.

The minimax algorithm

- The **minimax algorithm** computes the minimax decision from the current state.
- It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations.
- The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are **backed up** through the tree as the recursion unwinds.
- For example, in Figure given above, the algorithm first recurses down to the three bottom left nodes and uses the UTILITY function on them to discover that their values are 3, 12, and 8, respectively.
- Then it takes the minimum of these values, 3, and returns it as the backed up value of node B.
- A similar process gives the backed-up values of 2 for C and 2 for D.
- Finally, we take the maximum of 3, 2, and 2 to get the backed-up value of 3 for the root node.
- The minimax algorithm performs a complete depth-first exploration of the game tree.

The minimax algorithm (cont'd)



```
function MINIMAX-DECISION(state) returns an action
    return $\arg\max_{a \in \text{ACTIONS}(s)}$ MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    $v \leftarrow -\infty$
    for each a in ACTIONS(state) do
        $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a)))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    $v \leftarrow \infty$
    for each a in ACTIONS(state) do
        $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a)))
    return v
```

An algorithm for calculating minimax decisions.

- It returns the action corresponding
- to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility.
- The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state.

$\text{argmax}_{a \in S} f(a)$

The notation computes the element a of set S that has the maximum value of *f(a)*.

Pruning

- The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree.
- Unfortunately, we can’t eliminate the exponent, but it turns out we can effectively cut it in half.
- **Pruning** eliminate large parts of the tree from consideration.

ALPHA–BETA

- When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

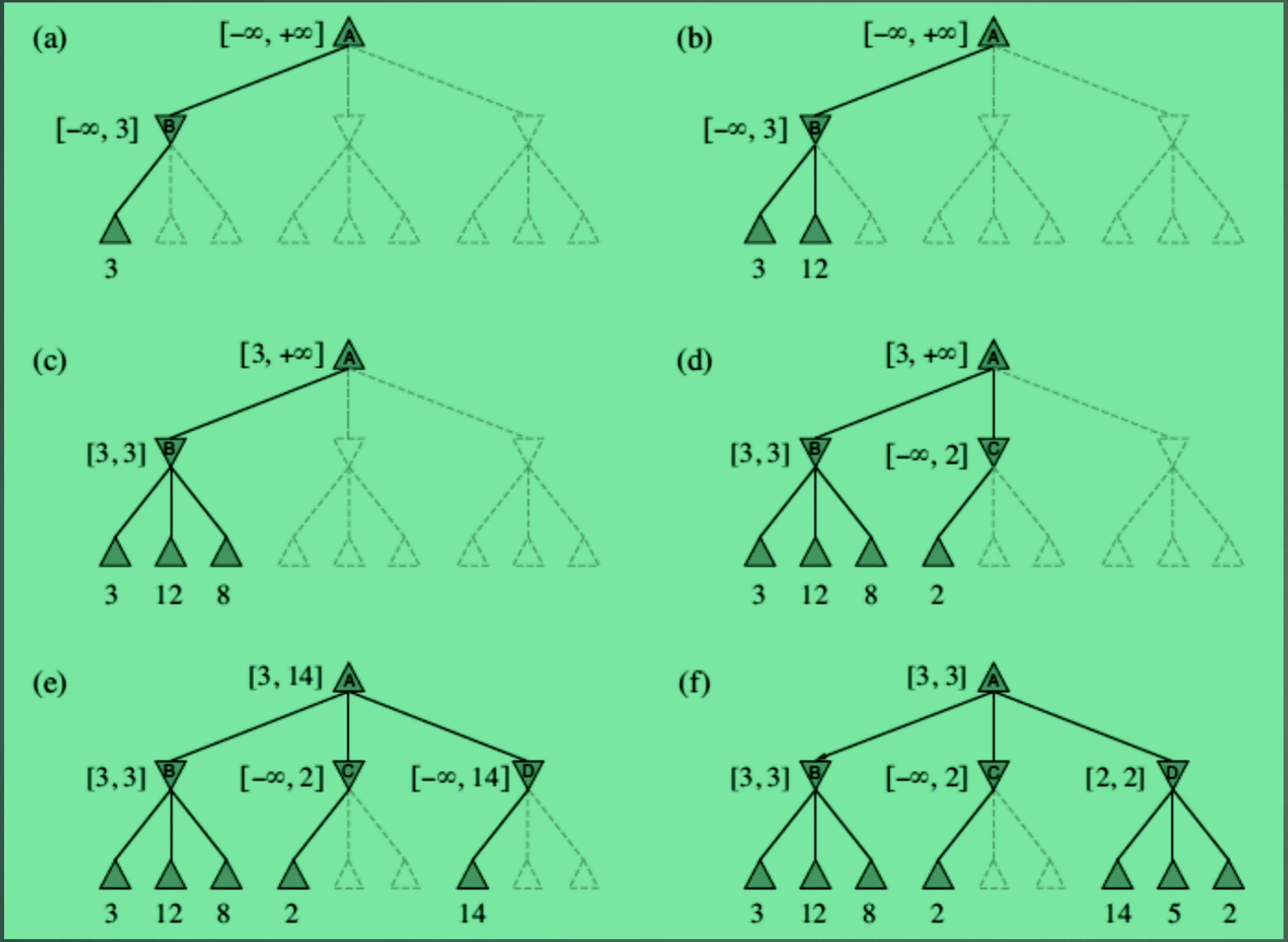


Figure 1: Stages in the calculation of the optimal decision for the game tree

At each point, we show the range of possible values for each node.

(a)The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of *at most* 3.

(b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3.

(c) The third leaf below B has a value of 8; we have seen all B's successor states, so the value of B is exactly 3.

Now, we can infer that the

(d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of *at most* 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successor states of C. This is an example of alpha–beta pruning.

(e) The first leaf below D has the value 14, so D is worth *at most* 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring D's successor states. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14.

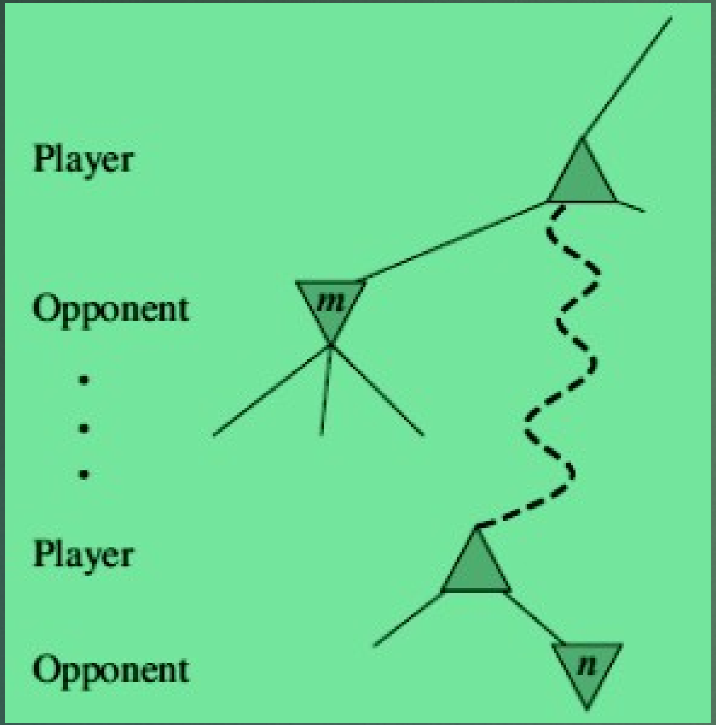
(f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX's decision at the root is to move to B, giving a value of 3.

The value of the root node is given by

$$\begin{aligned}\textsc{Minimax}(root) &= \max(\min(3,12,8), \min(2,x,y), \min(14,5,2)) \\ &= \max(3, \min(2,x,y), 2) \\ &= \max(3, z, 2) \qquad \text{where } z = \min(2,x,y) \le 2 \\ &= 3.\end{aligned}$$

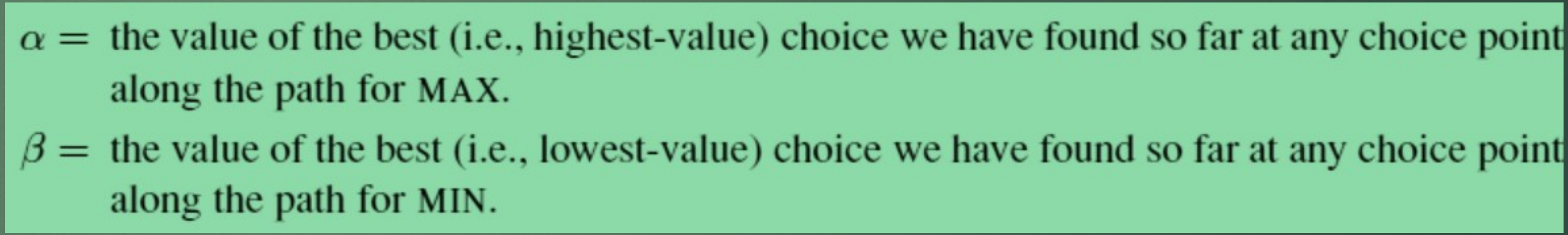
The value of the root and hence the minimax decision are *independent* of the values of the pruned leaves x and y.

General Case for alpha-beta pruning



If m is better than n for Player, we will never get to n in play.

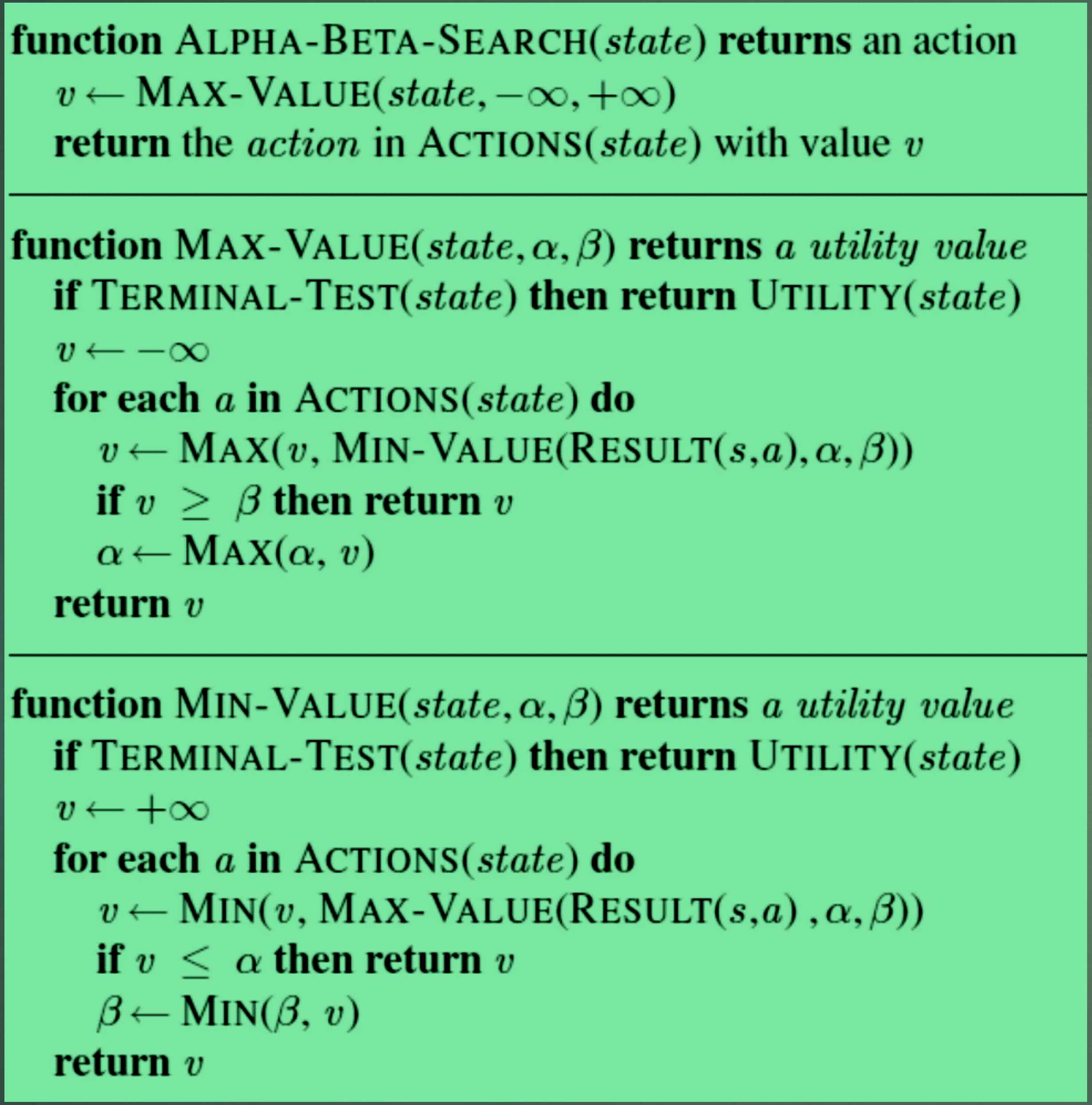
Alpha-beta pruning gets its name from the following two parameters that describe bounds on the backed-up values that appear anywhere along the path:



$\alpha =$ the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.

$\beta =$ the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.

Alpha-beta search updates the values of α and β as it goes along and prunes the remaining branches at a node (i.e., terminates the recursive call) as soon as the value of the current node is known to be worse than the current α or β value for MAX or MIN, respectively



```
function ALPHA-BETA-SEARCH(state) returns an action
   $v \leftarrow$ MAX-VALUE(state, $-\infty, +\infty$)
   return the action in ACTIONS(state) with value $v$

function MAX-VALUE(state, $\alpha, \beta$) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow -\infty$
   for each $a$ in ACTIONS(state) do
      $v \leftarrow$ MAX($v$, MIN-VALUE(RESULT($s,a$), $\alpha, \beta$))
      if $v \geq \beta$ then return $v$
      $\alpha \leftarrow$ MAX($\alpha$, $v$)
   return $v$

function MIN-VALUE(state, $\alpha, \beta$) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow +\infty$
   for each $a$ in ACTIONS(state) do
      $v \leftarrow$ MIN($v$, MAX-VALUE(RESULT($s,a$) , $\alpha, \beta$))
      if $v \leq \alpha$ then return $v$
      $\beta \leftarrow$ MIN($\beta$, $v$)
   return $v$
```

The alpha–beta search algorithm.

Notice that these routines are the same as the MINIMAX functions except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

Knowledge Representation

KNOWLEDGE BASE

- The central component of a knowledge-based agent is its **knowledge base**, or KB.
- A knowledge base is a set of **sentences**. (Here "sentence" is used as a technical term. It is related but not identical to the sentences of English and other natural languages.)

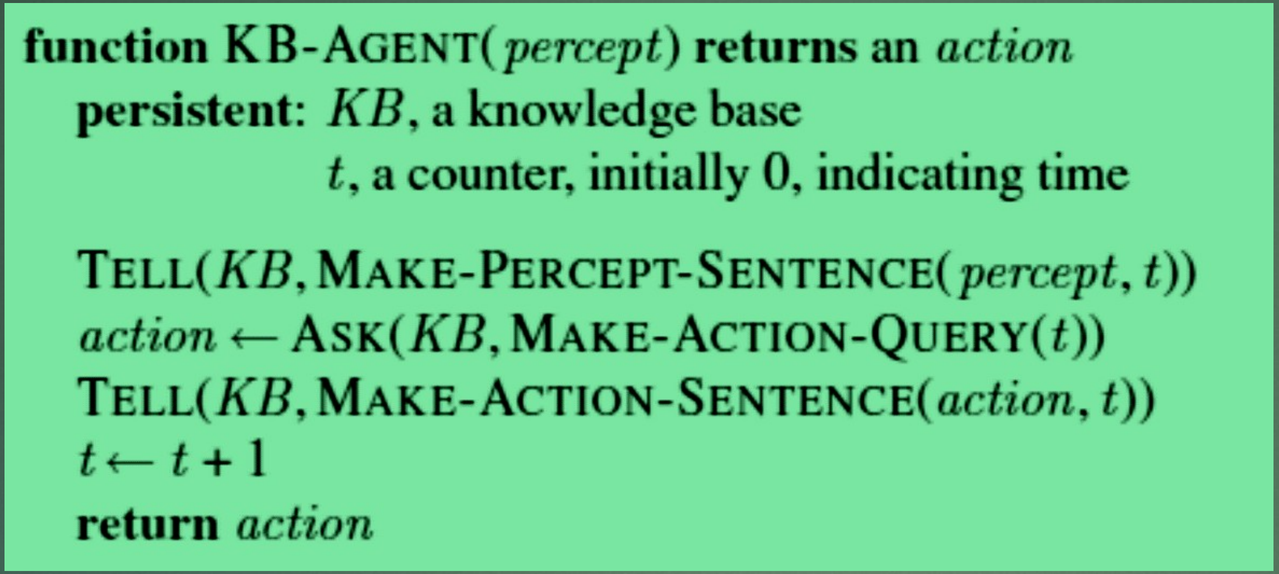
KNOWLEDGE REPRESENTATION LANGUAGE

- Each sentence is expressed in a language called a **knowledge representation language** and represents some assertion about the world.
- Sometimes we dignify a sentence with the name **axiom**, when the sentence is taken as given without being derived from other sentences.

INFERENCE

- There must be a way to add new sentences to the knowledge base and a way to query what is known.
- The standard names for these operations are TELL and ASK, respectively.
- Both operations may involve **inference**—that is, deriving new sentences from old.
- Inference must obey the requirement that when one ASKs a question of the knowledge base, the answer should follow from what has been told to the knowledge base previously.

A generic knowledge-based agent



```
function KB-AGENT(percept) returns an action
    persistent: KB, a knowledge base
                t, a counter, initially 0, indicating time

    TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
    action ← ASK(KB, MAKE-ACTION-QUERY(t))
    TELL(KB, MAKE-ACTION-SENTENCE(action, t))
    t ← t + 1
    return action
```

Given a percept, the agent adds the percept to its knowledge base, asks the knowledge base for the best action, and tells the knowledge base that it has in fact taken that action.

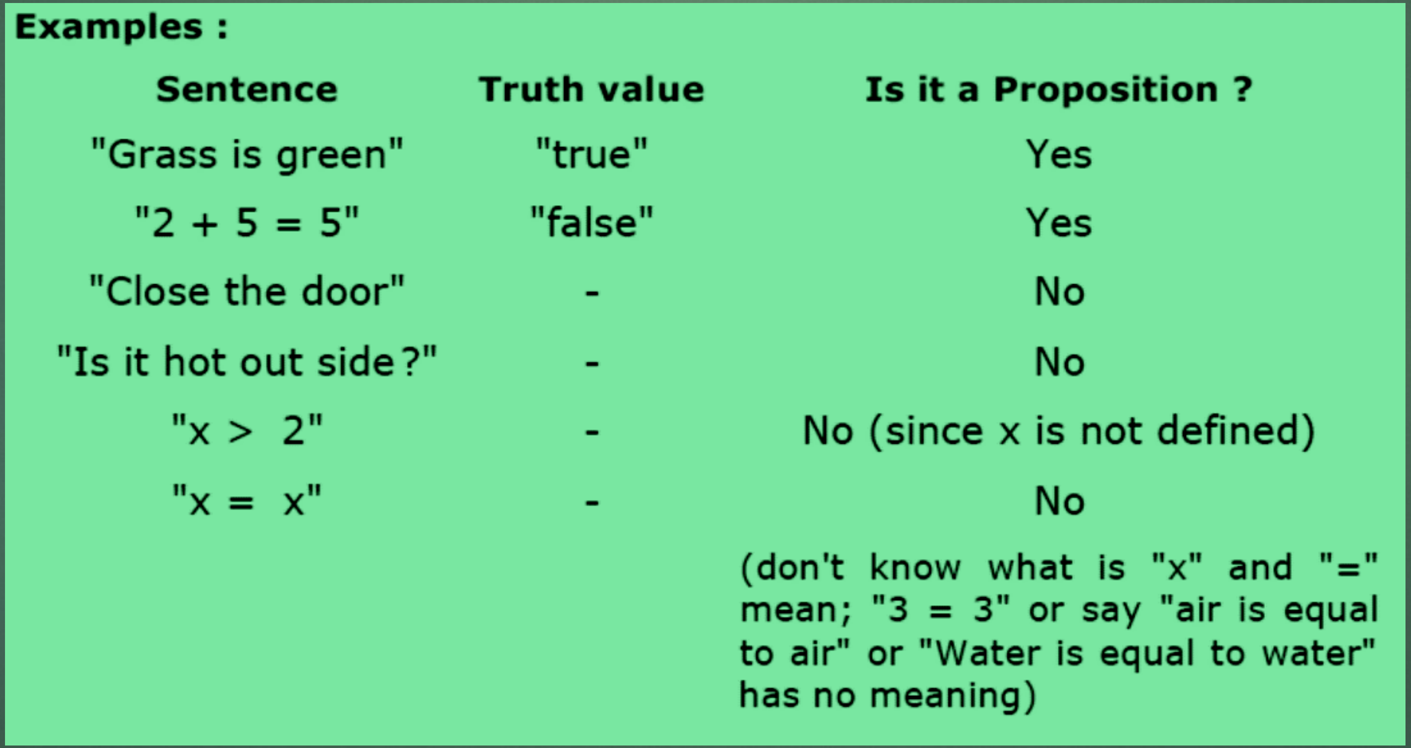
The details of the representation language are hidden inside **three functions** that implement the interface between the sensors and actuators on one side and the core representation and reasoning system on the other.

- **MAKE-PERCEPT-SENTENCE** constructs a sentence asserting that the agent perceived the given percept at the given time.
- **MAKE-ACTION-QUERY** constructs a sentence that asks what action should be done at the current time.
- **MAKE-ACTION-SENTENCE** constructs a sentence asserting that the chosen action was executed.

The details of the inference mechanism are hidden inside TELL and ASK. Later sectio

Logic

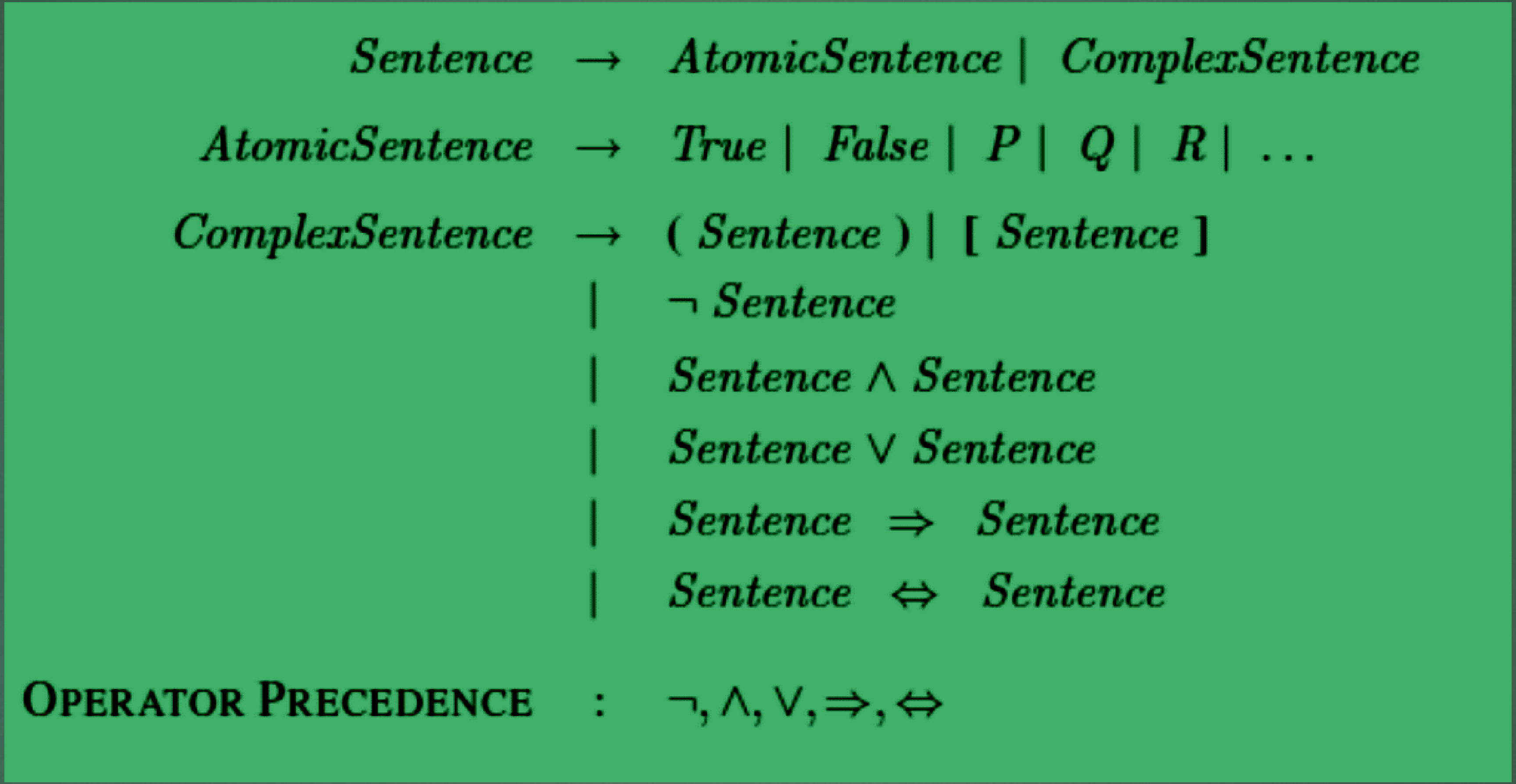
- Logic is used to represent simple facts.
- Logic defines the ways of putting symbols together to form sentences that represent facts.
- Sentences are either true or false but not both are called propositions.



Examples :

Sentence	Truth value	Is it a Proposition ?
"Grass is green"	"true"	Yes
"2 + 5 = 5"	"false"	Yes
"Close the door"	-	No
"Is it hot out side ?"	-	No
"x > 2"	-	No (since x is not defined)
"x = x"	-	No (don't know what is "x" and "=" mean; "3 = 3" or say "air is equal to air" or "Water is equal to water" has no meaning)

Propositional Logic (PL)



$$
\begin{array}{rcl}
\mathit{Sentence} & \rightarrow & \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} & \rightarrow & \mathit{True} \mid \mathit{False} \mid P \mid Q \mid R \mid \ldots \\
\mathit{ComplexSentence} & \rightarrow & (\ \mathit{Sentence}\ ) \mid [\ \mathit{Sentence}\ ] \\
 & \mid & \neg\ \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \wedge \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \vee \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \ \Rightarrow\ \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \ \Leftrightarrow\ \mathit{Sentence} \\
 & & \\
\textsc{Operator Precedence} & : & \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{array}
$$

A BNF (Backus–Naur Form) grammar of sentences in propositional logic, along with operator precedences, from highest to lowest.

Propositional Logic (PL) (Cont’d)

- **A proposition is a statement - which in English is a declarative sentence and Logic defines the ways of putting symbols together to form sentences that represent facts.**
- **Every proposition is either true or false.**
- **Propositional logic is also called Boolean algebra.**

Examples: (a) The sky is blue., (b) Snow is cold. , (c) 12 * 12=144

Propositional logic : It is fundamental to all logic.

- Propositions are “Sentences”; either true or false but not both.
- A sentence is smallest unit in propositional logic
- f proposition is true, then truth value is "true"; else “false”

Example: Sentence "Grass is green";

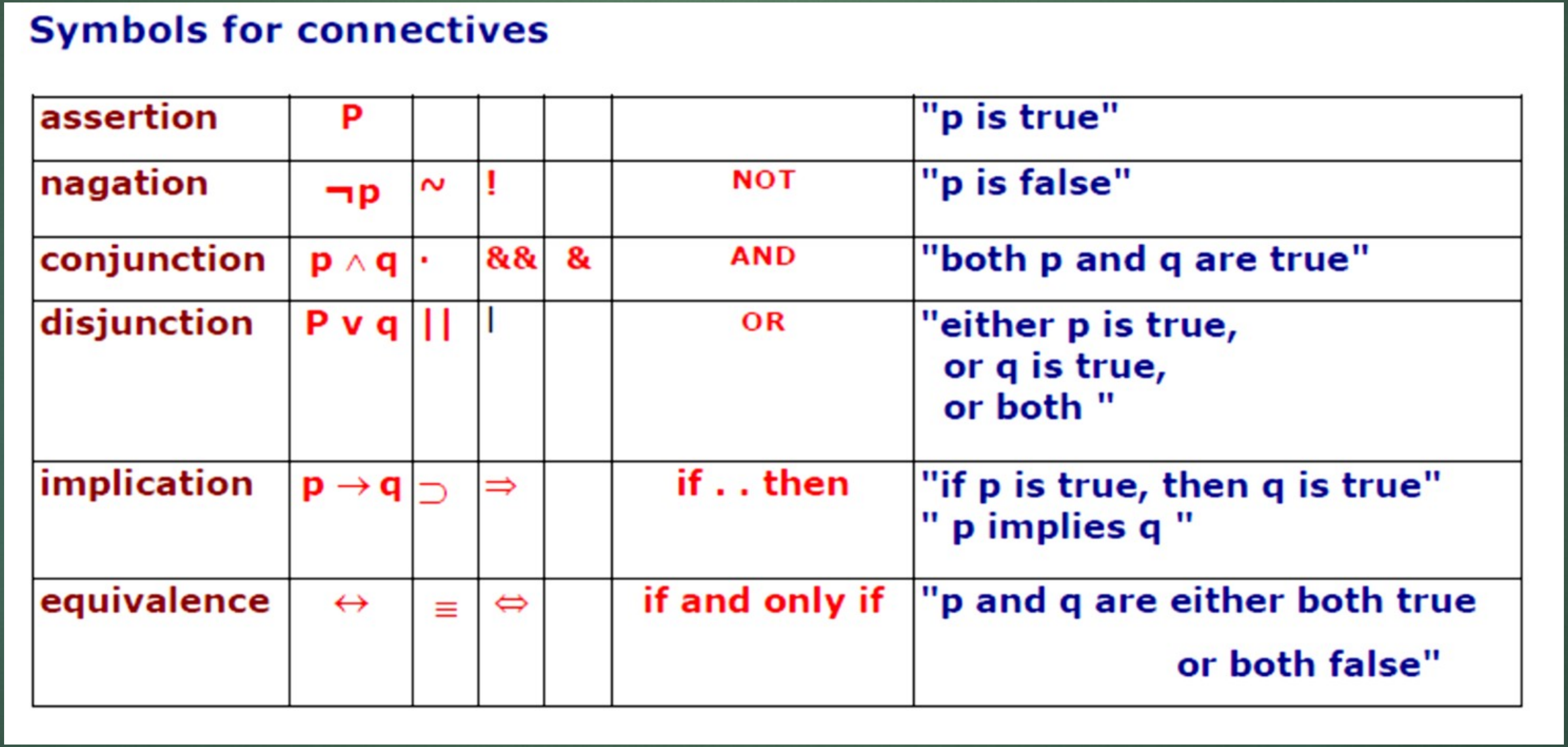
Truth value “ true”;

Proposition “yes”

Statement, Variables and Symbols

Statement : A simple statement is one that does not contain any other statement as a part. A compound statement is one that has two or more simple statements as parts called components.

Operator or connective : Joins simple statements into compounds, and joins compounds into larger compounds.



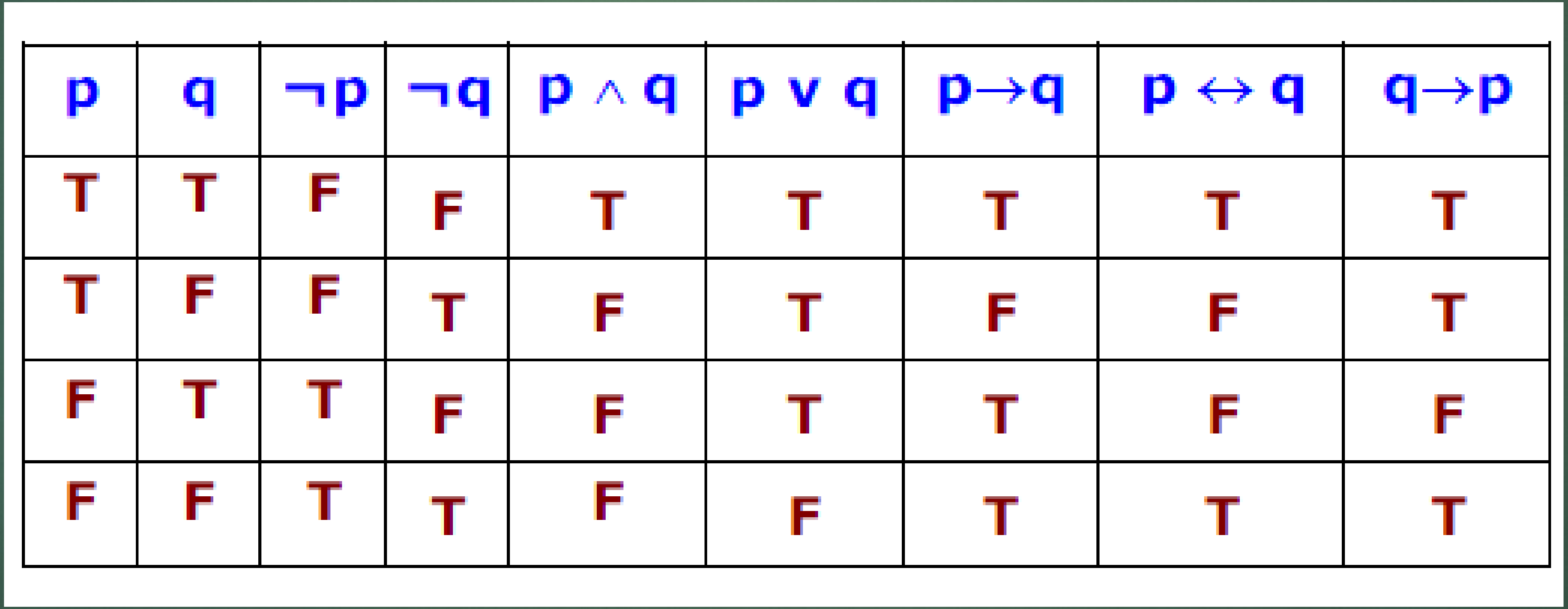
Symbols for connectives

assertion	P					"p is true"
nagation	¬p	~	!		NOT	"p is false"
conjunction	p ∧ q	·	&&	&	AND	"both p and q are true"
disjunction	P v q	||	|		OR	"either p is true, or q is true, or both "
implication	p → q	⊃	⇒		if . . then	"if p is true, then q is true" " p implies q "
equivalence	↔	≡	⇔		if and only if	"p and q are either both true or both false"

Truth table

- Is a convenient way of showing relationship between several propositions.

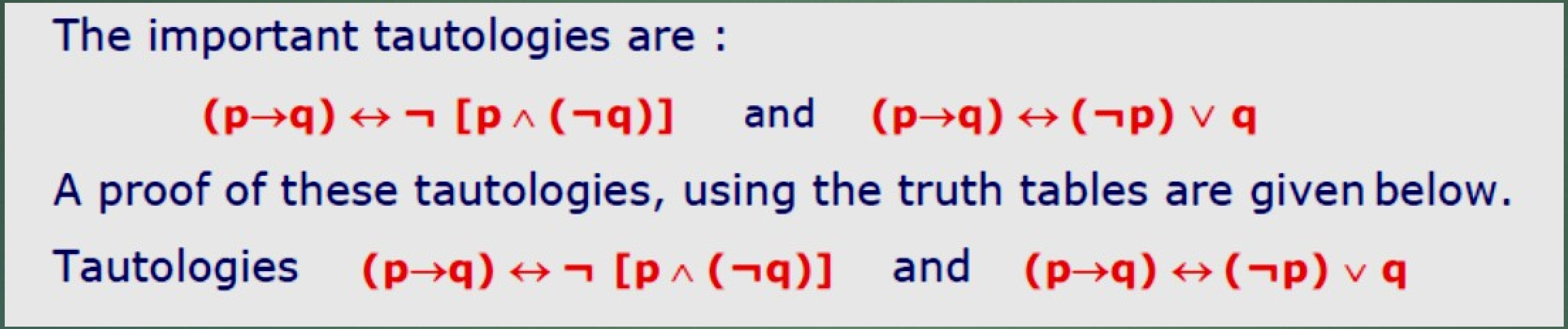
The truth table for negation, conjunction, disjunction, implication and equivalence are shown below.



p	q	¬p	¬q	p ∧ q	p ∨ q	p→q	p ↔ q	q→p
T	T	F	F	T	T	T	T	T
T	F	F	T	F	T	F	F	T
F	T	T	F	F	T	T	F	F
F	F	T	T	F	F	T	T	T

Tautology

A Tautology is proposition formed by combining other propositions (p, q, r, . . .) which is true regardless of truth or falsehood of p, q, r, ...



The important tautologies are :

$(p \to q) \leftrightarrow \neg\, [p \wedge (\neg q)]$ and $(p \to q) \leftrightarrow (\neg p) \vee q$

A proof of these tautologies, using the truth tables are given below.

Tautologies $(p \to q) \leftrightarrow \neg\, [p \wedge (\neg q)]$ and $(p \to q) \leftrightarrow (\neg p) \vee q$

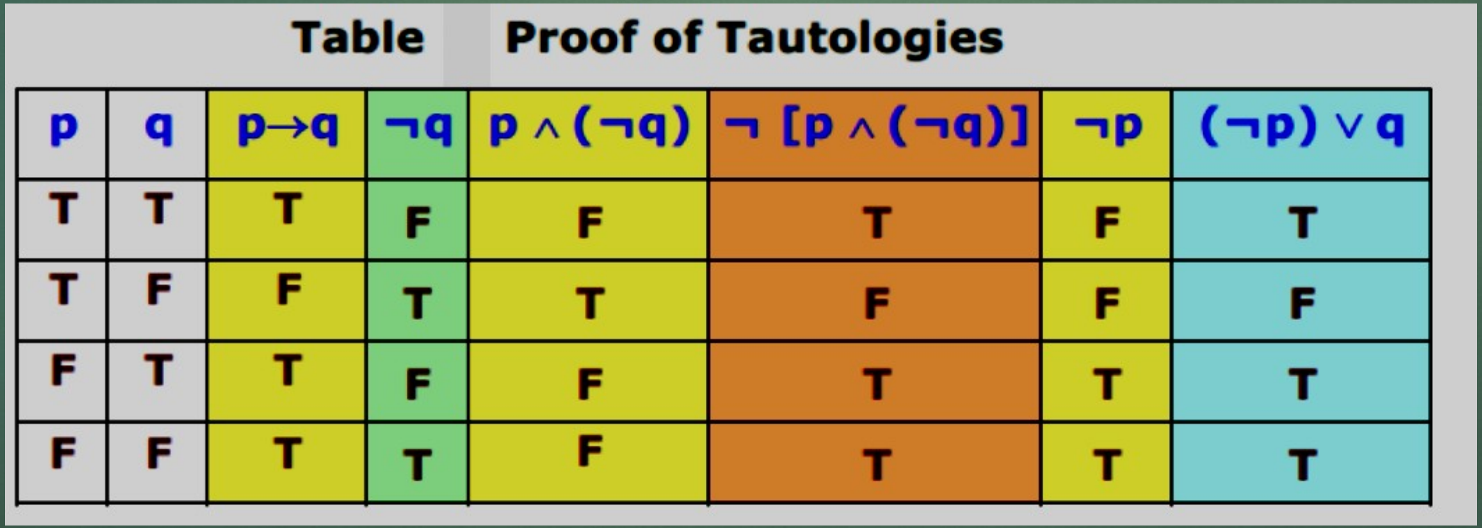
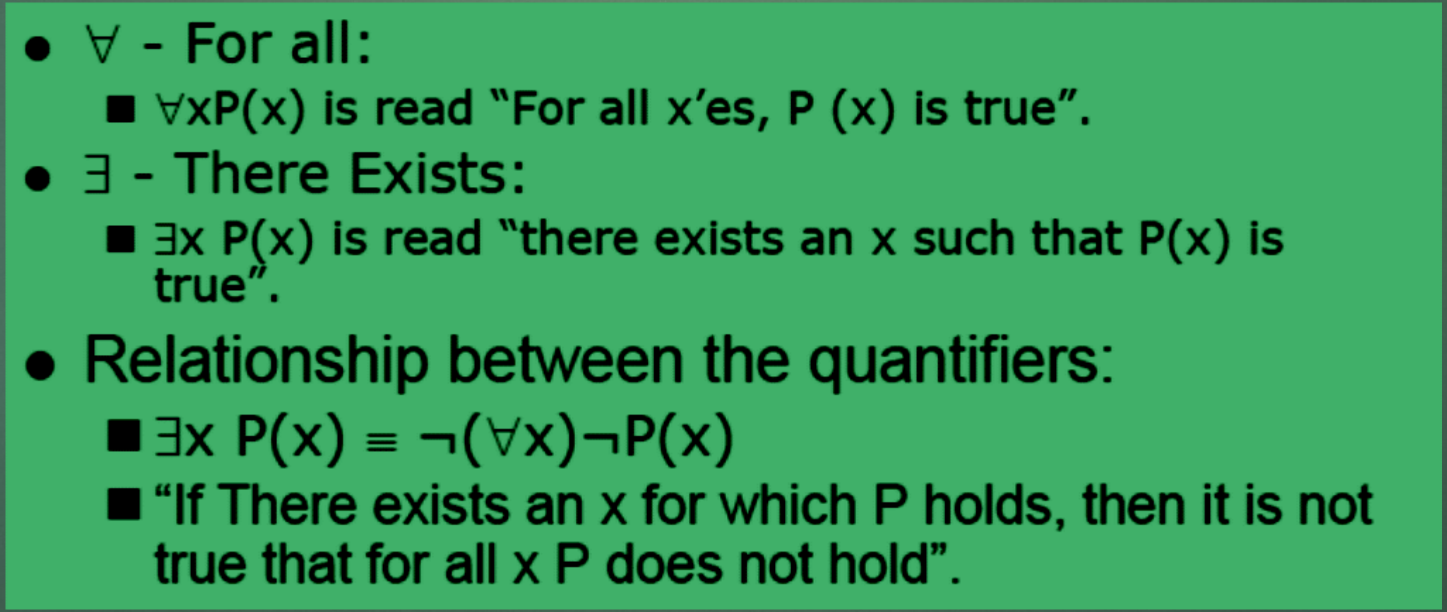


Table Proof of Tautologies

p	q	$p \to q$	$\neg q$	$p \wedge (\neg q)$	$\neg\, [p \wedge (\neg q)]$	$\neg p$	$(\neg p) \vee q$
T	T	T	F	F	T	F	T
T	F	F	T	T	F	F	F
F	T	T	F	F	T	T	T
F	F	T	T	F	T	T	T

Predicate Logic

- **Predicate Logic/ Calculus extends the syntax of propositional calculus with predicates and quantifiers:**
- **P(X) – P is a predicate.**
- **First Order Predicate Logic allows predicates to apply to objects or terms, but not functions or predicates.**



- $\forall$ - For all:
  - $\forall x P(x)$ is read "For all x'es, P (x) is true".
- $\exists$ - There Exists:
  - $\exists x\ P(x)$ is read "there exists an x such that P(x) is true".
- Relationship between the quantifiers:
  - $\exists x\ P(x) \equiv \neg(\forall x)\neg P(x)$
  - "If There exists an x for which P holds, then it is not true that for all x P does not hold".

References:

Stuart Russell and Peter Norvig, "Artificial intelligence-a modern approach 3rd ed." (2016).

S. Rajasekaran and G.A. Vijayalaksmi Pai ,Neural Network, Fuzzy Logic, and Genetic Algorithms - Synthesis and Applications, (2005), Prentice Hall.