```
#alpha beta pruning

# auto_tic_tac_toe_alphabeta.py
# Automatic Tic-Tac-Toe demonstrations using Minimax + Alpha-Beta pruning
# No user input required: the script runs games automatically.

import math
import random
import time

EMPTY = " "
X = "X"   # we'll treat X as +1 (maximizer)
O = "O"   # O is -1 (minimizer)

WIN_LINES = [
    (0,1,2), (3,4,5), (6,7,8),   # rows
    (0,3,6), (1,4,7), (2,5,8),   # cols
    (0,4,8), (2,4,6)             # diags
]

def new_board():
    return [EMPTY] * 9

def print_board(board):
    for r in range(3):
        row = board[3*r:3*r+3]
        print(" " + " | ".join(row))
        if r < 2:
            print("---+---+---")
    print()

def available_moves(board):
    return [i for i, v in enumerate(board) if v == EMPTY]

def is_winner(board, player):
    return any(all(board[i] == player for i in line) for line in WIN_LINES)

def is_full(board):
    return all(cell != EMPTY for cell in board)

def evaluate(board):
    """Terminal evaluation: +1 if X wins, -1 if O wins, 0 otherwise."""
    if is_winner(board, X):
        return 1
    if is_winner(board, O):
        return -1
    return 0

def minimax_alpha_beta(board, player, alpha, beta, depth=0):
    """
    General minimax with alpha-beta pruning for Tic-Tac-Toe.
    player: the player to move now (X or O).
    Returns (best_score, best_move_index).
    Depth is used only to prefer faster wins / slower losses.
    """
    score = evaluate(board)
    if score != 0 or is_full(board):
        # If terminal, prefer faster win / slower loss by factoring depth
        if score == 1:
            return 10 - depth, None   # X wins -> positive
        if score == -1:
            return depth - 10, None  # O wins -> negative
        return 0, None               # draw

    if player == X:
        max_eval = -math.inf
        best_move = None
        for move in available_moves(board):
            board[move] = X
            eval_score, _ = minimax_alpha_beta(board, O, alpha, beta, depth+1)
            board[move] = EMPTY

            if eval_score > max_eval:
                max_eval = eval_score
                best_move = move

            alpha = max(alpha, eval_score)
            if beta <= alpha:
                break  # beta cut-off
        return max_eval, best_move
```

```
        else:  # player == O (minimizer)
            min_eval = math.inf
            best_move = None
            for move in available_moves(board):
                board[move] = O
                eval_score, _ = minimax_alpha_beta(board, X, alpha, beta, depth+1)
                board[move] = EMPTY

                if eval_score < min_eval:
                    min_eval = eval_score
                    best_move = move

                beta = min(beta, eval_score)
                if beta <= alpha:
                    break  # alpha cut-off
            return min_eval, best_move

    def best_move_alphabeta(board, player):
        """Return best move for player using alpha-beta pruning."""
        _, move = minimax_alpha_beta(board, player, -math.inf, math.inf, depth=0)
        return move

    def best_move_random(board):
        """Random move for weak opponent."""
        moves = available_moves(board)
        return random.choice(moves) if moves else None

    def play_auto_game(starting_player, mode="AIvRandom", verbose=True, delay=0.3):
        """
        Play a single automatic game.
        starting_player: X or O
        mode: "AIvRandom" or "AIvAI"
        verbose: print board and moves if True
        delay: seconds to wait between moves (for readability)
        Returns final board and result string.
        """
        board = new_board()
        current = starting_player

        if verbose:
            print("Starting automatic game. Mode:", mode)
            print("Starting player:", current)
            print_board(board)
            time.sleep(delay)

        while True:
            if mode == "AIvRandom":
                if current == X:
                    move = best_move_alphabeta(board, X)
                else:  # O is random
                    move = best_move_random(board)
            else:  # "AIvAI"
                move = best_move_alphabeta(board, current)

            if move is None:
                # No moves left (should be handled by terminal checks)
                break

            board[move] = current
            if verbose:
                print(f"{current} -> {move+1}")
                print_board(board)
                time.sleep(delay)

            if is_winner(board, current):
                if current == X:
                    result = "X wins"
                else:
                    result = "O wins"
                if verbose:
                    print("Result:", result)
                return board, result

            if is_full(board):
                if verbose:
                    print("Result: Draw")
                return board, "Draw"

            current = O if current == X else X

    def demo():
        random.seed(1)  # deterministic randomness for repeatability
```

```
    print("\n--- Demo 1: AI (X, alpha-beta) vs Random (O) ---\n")
    board, result = play_auto_game(starting_player=X, mode="AIvRandom", verbose=True, delay=0.15)

    print("\n--- Demo 2: AI (X) vs AI (O) (both alpha-beta) ---\n")
    board2, result2 = play_auto_game(starting_player=X, mode="AIvAI", verbose=True, delay=0.15)

    print("\nSummary:")
    print("Demo 1 result:", result)
    print("Demo 2 result:", result2)

if __name__ == "__main__":
    demo()
```

```
   | O |
---+---+---
   |   |

X -> 2
 X | X |
---+---+---
   | O |
---+---+---
   |   |

O -> 3
 X | X | O
---+---+---
   | O |
---+---+---
   |   |

X -> 7
 X | X | O
---+---+---
   | O |
---+---+---
 X |   |

O -> 4
 X | X | O
---+---+---
 O | O |
---+---+---
 X |   |

X -> 6
 X | X | O
---+---+---
 O | O | X
---+---+---
 X |   |

O -> 8
 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O |

X -> 9
 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O | X

Result: Draw

Summary:
Demo 1 result: X wins
Demo 2 result: Draw
```

```
#VACCUm CLEaner

# Vacuum Cleaner Problem (2 Rooms)
# Basic Reflex Agent

def vacuum_world(state, location):
    """
    state: dictionary with room cleanliness status
           {"A": "Dirty" or "Clean", "B": "Dirty" or "Clean"}
    location: "A" or "B"
    """

    print(f"\nVacuum starts in Room {location}")
    print(f"Initial State: {state}")
```

```
    # Action loop until all rooms are clean
    steps = 0
    while state["A"] == "Dirty" or state["B"] == "Dirty":
        steps += 1
        print(f"\nStep {steps}: Vacuum is at Room {location}")

        if state[location] == "Dirty":
            print(f"Room {location} is Dirty → SUCK")
            state[location] = "Clean"
        else:
            print(f"Room {location} is Clean → MOVE to other room")
            location = "A" if location == "B" else "B"

        print(f"Current State: {state}")

    print("\nAll rooms are clean!")
    print(f"Total steps taken: {steps}")


# Example test run
initial_state = {"A": "Dirty", "B": "Dirty"}
start_location = "A"

vacuum_world(initial_state, start_location)
```

```
Vacuum starts in Room A
Initial State: {'A': 'Dirty', 'B': 'Dirty'}

Step 1: Vacuum is at Room A
Room A is Dirty → SUCK
Current State: {'A': 'Clean', 'B': 'Dirty'}

Step 2: Vacuum is at Room A
Room A is Clean → MOVE to other room
Current State: {'A': 'Clean', 'B': 'Dirty'}

Step 3: Vacuum is at Room B
Room B is Dirty → SUCK
Current State: {'A': 'Clean', 'B': 'Clean'}

All rooms are clean!
Total steps taken: 3
```

```
#UNIFICATION

# fol_unify.py
# A simple First-Order Logic unifier with parser and occurs-check.

import re
from typing import List, Dict, Optional, Union, Tuple

# ----------------------------
# Term classes
# ----------------------------

class Term:
    def apply_subst(self, subst: Dict["Variable", "Term"]) -> "Term":
        raise NotImplementedError()

    def vars(self) -> set:
        """Return set of Variable objects occurring in this term."""
        raise NotImplementedError()

    def __repr__(self):
        return self.__str__()

class Variable(Term):
    def __init__(self, name: str):
        self.name = name

    def apply_subst(self, subst: Dict["Variable", Term]) -> Term:
        # If variable is in substitution map, apply substitution (and then apply recursively)
        for v in subst:
            # use name equality for lookup convenience
            if v.name == self.name:
                return subst[v].apply_subst(subst)
        return self

    def vars(self) -> set:
        return {self}
```

```
        def __eq__(self, other):
            return isinstance(other, Variable) and self.name == other.name

        def __hash__(self):
            return hash(("Var", self.name))

        def __str__(self):
            return self.name

    class Constant(Term):
        def __init__(self, name: str):
            self.name = name

        def apply_subst(self, subst: Dict[Variable, Term]) -> Term:
            return self  # constants unaffected

        def vars(self) -> set:
            return set()

        def __eq__(self, other):
            return isinstance(other, Constant) and self.name == other.name

        def __hash__(self):
            return hash(("Const", self.name))

        def __str__(self):
            return self.name

    class Function(Term):
        def __init__(self, name: str, args: List[Term]):
            self.name = name
            self.args = args

        def apply_subst(self, subst: Dict[Variable, Term]) -> Term:
            return Function(self.name, [arg.apply_subst(subst) for arg in self.args])

        def vars(self) -> set:
            s = set()
            for a in self.args:
                s |= a.vars()
            return s

        def __eq__(self, other):
            return isinstance(other, Function) and self.name == other.name and self.args == other.args

        def __hash__(self):
            return hash(("Func", self.name, tuple(self.args)))

        def __str__(self):
            if len(self.args) == 0:
                return self.name
            return f"{self.name}({', '.join(map(str, self.args))})"


    # ----------------------------
    # Substitution utilities
    # ----------------------------

    Subst = Dict[Variable, Term]

    def compose_subst(s1: Subst, s2: Subst) -> Subst:
        """
        Return composition s = s1 ∘ s2 meaning apply s2 then s1.
        Implemented so that each term in s1 has s2 applied, and we keep s2 entries that aren't overridden.
        """
        new = {}
        # Apply s2 to all terms in s1
        for v, t in s1.items():
            new_v = v
            new_t = t.apply_subst(s2)
            new[new_v] = new_t
        # Add entries from s2 that are not in s1
        for v, t in s2.items():
            if v not in new:
                new[v] = t
        return new

    def apply_subst_to_term(term: Term, subst: Subst) -> Term:
        return term.apply_subst(subst)


    # ----------------------------
```

```
# Occurs-check
# ----------------------------

def occurs_check(var: Variable, term: Term, subst: Subst) -> bool:
    """
    Check whether variable `var` occurs in `term` after applying current substitution `subst`.
    Returns True if occurs -> then cannot bind var to term (would create a cyclic substitution).
    """
    term_applied = term.apply_subst(subst)
    return var in term_applied.vars()


# ----------------------------
# Unification algorithm (Robinson's algorithm with occurs-check)
# ----------------------------

def unify(t1: Term, t2: Term, subst: Optional[Subst] = None) -> Optional[Subst]:
    """
    Attempt to unify terms t1 and t2 given initial substitution subst.
    Returns the most general unifier (a substitution dict Variable -> Term) or None on failure.
    """
    if subst is None:
        subst = {}

    # Worklist of pairs
    pairs: List[Tuple[Term, Term]] = [(t1, t2)]
    current_subst: Subst = dict(subst)  # copy

    while pairs:
        s, t = pairs.pop(0)
        # apply current substitution
        s = s.apply_subst(current_subst)
        t = t.apply_subst(current_subst)

        # print("Debug pair:", s, t)  # uncomment for step debugging

        if s == t:
            continue

        if isinstance(s, Variable):
            if occurs_check(s, t, current_subst):
                return None  # failure due to occurs-check
            # add substitution s -> t
            current_subst = compose_subst({s: t}, current_subst)
            continue

        if isinstance(t, Variable):
            if occurs_check(t, s, current_subst):
                return None
            current_subst = compose_subst({t: s}, current_subst)
            continue

        # Both are functions or constants
        if isinstance(s, Constant) and isinstance(t, Constant):
            # different constants can't be unified
            if s.name != t.name:
                return None
            else:
                continue

        if isinstance(s, Function) and isinstance(t, Function):
            if s.name != t.name or len(s.args) != len(t.args):
                return None
            # push pairwise arguments
            pairs = [(sa, ta) for sa, ta in zip(s.args, t.args)] + pairs
            continue

        # other cases not unifyable
        return None

    return current_subst


# ----------------------------
# Simple parser for terms and predicates
# ----------------------------

TOKEN_REGEX = r"\s*([A-Za-z_][A-Za-z_0-9]*|\(|\)|,)\s*"

def tokenize(s: str) -> List[str]:
    tokens = re.findall(TOKEN_REGEX, s)
    return [t for t in tokens if t.strip() != ""]
```

```
def parse_term_from_tokens(tokens: List[str], pos: int = 0) -> Tuple[Term, int]:
    """
    Parse a term starting at tokens[pos].
    Returns (term, new_pos)
    Grammar (simple):
      term ::= ID | ID '(' term (',' term)* ')'
    We decide variable vs constant/function name by identifier's first character:
      - If starts with lowercase letter -> Variable
      - Else -> Constant or Function (if followed by '(')
    """
    if pos >= len(tokens):
        raise ValueError("Unexpected end of tokens")

    token = tokens[pos]
    if re.match(r"[A-Za-z_][A-Za-z_0-9]*", token) is None:
        raise ValueError(f"Expected identifier at pos {pos}, found {token}")

    name = token
    pos += 1
    # function / predicate with args?
    if pos < len(tokens) and tokens[pos] == "(":
        pos += 1  # skip '('
        args = []
        # handle empty-arg functions (rare)
        if pos < len(tokens) and tokens[pos] == ")":
            pos += 1
            term = Function(name, [])
            return term, pos

        while True:
            arg, pos = parse_term_from_tokens(tokens, pos)
            args.append(arg)
            if pos >= len(tokens):
                raise ValueError("Missing closing ')' in function")
            if tokens[pos] == ",":
                pos += 1
                continue
            elif tokens[pos] == ")":
                pos += 1
                break
            else:
                raise ValueError(f"Unexpected token {tokens[pos]} in args")
        term = Function(name, args)
        return term, pos
    else:
        # no args -> variable or constant
        if name[0].islower():
            return Variable(name), pos
        else:
            return Constant(name), pos

def parse_term(s: str) -> Term:
    tokens = tokenize(s)
    term, pos = parse_term_from_tokens(tokens, 0)
    if pos != len(tokens):
        raise ValueError("Extra tokens after parsing: " + " ".join(tokens[pos:]))
    return term

# ---------------------------
# Utility to pretty-print substitution
# ---------------------------

def subst_to_str(subst: Optional[Subst]) -> str:
    if subst is None:
        return "Fail"
    if not subst:
        return "{}"
    items = []
    for v, t in subst.items():
        items.append(f"{v} -> {t}")
    return "{ " + ", ".join(items) + " }"


# ---------------------------
# Demonstration / Tests
# ---------------------------

if __name__ == "__main__":
    examples = [
        ("Eats(x, Apple)", "Eats(Riya, y)"),
        ("p(f(a), g(Y))", "p(X, X)"),           # should fail (example in prompt)
```

```
        ("Knows(John, x)", "Knows(x, Elisabeth)"),  # should fail
        ("P(x, h(y))", "P(a, f(z))"),           # should fail because h != f
        ("f(x, x)", "f(a, b)"),                 # fail: x must be both a and b
        ("parent(John, Mary)", "parent(John, Mary)"), # trivial unify
        ("q(X, g(Y))", "q(f(a), g(b))"),        # Y->b, X->f(a)
        ("r(X)", "r(f(X))"),                    # occurs-check failure
    ]

    for a, b in examples:
        t1 = parse_term(a)
        t2 = parse_term(b)
        print("-----------------------------------------------")
        print("Unify:", a, "  WITH  ", b)
        result = unify(t1, t2)
        print("Result:", subst_to_str(result))
```

```
-----------------------------------------------
Unify: Eats(x, Apple)   WITH   Eats(Riya, y)
Result: { y -> Apple, x -> Riya }
-----------------------------------------------
Unify: p(f(a), g(Y))   WITH   p(X, X)
Result: Fail
-----------------------------------------------
Unify: Knows(John, x)   WITH   Knows(x, Elisabeth)
Result: Fail
-----------------------------------------------
Unify: P(x, h(y))   WITH   P(a, f(z))
Result: Fail
-----------------------------------------------
Unify: f(x, x)   WITH   f(a, b)
Result: { a -> b, x -> a }
-----------------------------------------------
Unify: parent(John, Mary)   WITH   parent(John, Mary)
Result: {}
-----------------------------------------------
Unify: q(X, g(Y))   WITH   q(f(a), g(b))
Result: Fail
-----------------------------------------------
Unify: r(X)   WITH   r(f(X))
Result: Fail
```

```
# fol_forward_chaining.py
# Forward chaining for FOL definite clauses with unification and standardizing variables

import re
import itertools
import math
from typing import List, Dict, Tuple, Optional, Set

# ------------------------
# Term classes + parser
# ------------------------

class Term:
    def apply(self, subst: Dict[str, "Term"]) -> "Term":
        raise NotImplementedError()
    def vars(self) -> Set[str]:
        raise NotImplementedError()

class Var(Term):
    def __init__(self, name: str):
        self.name = name
    def apply(self, subst):
        if self.name in subst:
            return subst[self.name].apply(subst)
        return self
    def vars(self):
        return {self.name}
    def __eq__(self, other):
        return isinstance(other, Var) and self.name == other.name
    def __hash__(self):
        return hash(("Var", self.name))
    def __str__(self):
        return self.name

class Const(Term):
    def __init__(self, name: str):
        self.name = name
    def apply(self, subst):
        return self
    def vars(self):
        return set()
    def __eq__(self, other):
```

```
        return isinstance(other, Const) and self.name == other.name
    def __hash__(self):
        return hash(("Const", self.name))
    def __str__(self):
        return self.name

class Func(Term):
    def __init__(self, name: str, args: List[Term]):
        self.name = name
        self.args = args
    def apply(self, subst):
        return Func(self.name, [a.apply(subst) for a in self.args])
    def vars(self):
        s = set()
        for a in self.args:
            s |= a.vars()
        return s
    def __eq__(self, other):
        return isinstance(other, Func) and self.name == other.name and self.args == other.args
    def __hash__(self):
        return hash(("Func", self.name, tuple(self.args)))
    def __str__(self):
        if not self.args:
            return self.name
        return f"{self.name}({', '.join(map(str, self.args))})"

TOKEN_RE = r"\s*([A-Za-z_][A-Za-z_0-9]*|\(|\)|,|\->)\s*"

def tokenize(s: str):
    toks = re.findall(TOKEN_RE, s)
    return [t for t in toks if t.strip()]

def parse_term_from_tokens(tokens, pos=0) -> Tuple[Term,int]:
    if pos >= len(tokens):
        raise ValueError("unexpected end")
    tok = tokens[pos]
    if not re.match(r"[A-Za-z_][A-Za-z_0-9]*", tok):
        raise ValueError("expected identifier")
    name = tok
    pos += 1
    if pos < len(tokens) and tokens[pos] == "(":
        pos += 1
        args = []
        if pos < len(tokens) and tokens[pos] == ")":
            pos += 1
            return Func(name, []), pos
        while True:
            arg, pos = parse_term_from_tokens(tokens, pos)
            args.append(arg)
            if pos >= len(tokens):
                raise ValueError("missing ')'")
            if tokens[pos] == ",":
                pos += 1
                continue
            if tokens[pos] == ")":
                pos += 1
                break
            raise ValueError("unexpected token " + tokens[pos])
        return Func(name, args), pos
    else:
        # variable if starts with lowercase, constant/function symbol if uppercase
        if name[0].islower():
            return Var(name), pos
        else:
            return Const(name), pos

def parse_atom(s: str) -> Func:
    tokens = tokenize(s)
    term, pos = parse_term_from_tokens(tokens, 0)
    if pos != len(tokens):
        raise ValueError("extra tokens")
    if not isinstance(term, Func):
        # treat zero-arg predicate as Func(name, [])
        return Func(str(term), [])
    return term

# -----------------------
# Unification (with occurs-check)
# -----------------------

def occurs_check(varname: str, term: Term, subst: Dict[str, Term]) -> bool:
    t = term.apply(subst)
```

```
        return varname in t.vars()

    def unify_terms(t1: Term, t2: Term, subst: Dict[str, Term]) -> Optional[Dict[str, Term]]:
        # returns updated subst or None if failure
        # apply current substitution first
        t1 = t1.apply(subst)
        t2 = t2.apply(subst)
        if isinstance(t1, Var):
            if t1.name == t2.__dict__.get("name", None) and isinstance(t2, Var):
                return subst
            if occurs_check(t1.name, t2, subst):
                return None
            new = dict(subst)
            new[t1.name] = t2
            return new
        if isinstance(t2, Var):
            return unify_terms(t2, t1, subst)
        if isinstance(t1, Const) and isinstance(t2, Const):
            if t1.name == t2.name:
                return subst
            return None
        if isinstance(t1, Func) and isinstance(t2, Func):
            if t1.name != t2.name or len(t1.args) != len(t2.args):
                return None
            s = dict(subst)
            for a,b in zip(t1.args, t2.args):
                s = unify_terms(a, b, s)
                if s is None:
                    return None
            return s
        return None

    def unify_atoms(a: Func, b: Func, subst: Dict[str, Term]) -> Optional[Dict[str, Term]]:
        # predicate names and arity must match
        if a.name != b.name or len(a.args) != len(b.args):
            return None
        s = dict(subst)
        for x,y in zip(a.args, b.args):
            s = unify_terms(x, y, s)
            if s is None:
                return None
        return s

    # ------------------------
    # Clauses / KB
    # ------------------------

    class Rule:
        def __init__(self, antecedents: List[Func], consequent: Func):
            self.antecedents = antecedents
            self.consequent = consequent
        def __str__(self):
            if self.antecedents:
                return f"{' & '.join(map(str,self.antecedents))} -> {self.consequent}"
            else:
                return str(self.consequent)

    # Standardize variables apart by renaming variables in rule to fresh names
    _counter = 0
    def fresh_var_name(base: str) -> str:
        global _counter
        _counter += 1
        return f"{base}_{_counter}"

    def standardize_apart(rule: Rule) -> Rule:
        # collect variables in the rule
        varset = set()
        for a in rule.antecedents + [rule.consequent]:
            varset |= a.vars()
        mapping = {}
        for v in varset:
            mapping[v] = Var(fresh_var_name(v))
        # apply mapping to terms
        def remap_term(t: Term):
            if isinstance(t, Var):
                return mapping[t.name]
            if isinstance(t, Const):
                return t
            return Func(t.name, [remap_term(arg) for arg in t.args])
        antecedents = [remap_term(a) for a in rule.antecedents]
        consequent = remap_term(rule.consequent)
        return Rule(antecedents, consequent)
```

```
# ------------------------
# Forward chaining algorithm (FOL-FC-ASK)
# ------------------------

def fol_fc_ask(kb_facts: List[Func], kb_rules: List[Rule], query: Func) -> Optional[Dict[str, Term]]:
    """
    kb_facts: list of ground atoms (Func) -- constants only expected
    kb_rules: list of Rule objects (with Vars)
    query: atom to prove (may contain constants/vars)
    Returns substitution (dict var->Term) that proves query or None.
    """
    # KB as set for fast membership (use string repr for simplicity)
    known = list(kb_facts)  # allow duplicates semantics via list
    known_set = set(str(f) for f in known)
    print("Initial facts:")
    for f in known:
        print("  ", f)
    iteration = 0
    while True:
        iteration += 1
        new_facts = []
        # For each rule, standardized apart
        for rule in kb_rules:
            std_rule = standardize_apart(rule)
            n = len(std_rule.antecedents)
            # if no antecedents (a fact rule), just try consequent
            if n == 0:
                g = std_rule.consequent
                ground_g = g.apply({})  # no subst
                if str(ground_g) not in known_set:
                    new_facts.append(ground_g)
                continue
            # try to find substitutions that make each antecedent unify with some known fact
            # we will try all combinations of known facts of length n (with repetition allowed)
            for facts_combo in itertools.product(known, repeat=n):
                s = {}
                failed = False
                for pat, fact in zip(std_rule.antecedents, facts_combo):
                    s = unify_atoms(pat, fact, s)
                    if s is None:
                        failed = True
                        break
                if failed:
                    continue
                # s is a substitution making all antecedents match these facts
                # produce the instantiated consequent
                instantiated_consequent = std_rule.consequent.apply(s)
                if str(instantiated_consequent) not in known_set:
                    new_facts.append(instantiated_consequent)
        # deduplicate new_facts
        added_any = False
        for nf in new_facts:
            if str(nf) not in known_set:
                known.append(nf)
                known_set.add(str(nf))
                added_any = True
                print(f"[Iter {iteration}] Inferred: {nf}")
                # check whether query is satisfied by this new fact (allow query with variables or constants)
                s = {}
                unify_result = unify_atoms(query, nf, s)
                if unify_result is not None:
                    print("Query unified with inferred fact:", nf)
                    return unify_result
        # Also check existing facts for query (in case present at start)
        for f in known:
            res = unify_atoms(query, f, {})
            if res is not None:
                print("Query matches known fact:", f)
                return res
        if not added_any:
            print("No new facts inferred; stopping.")
            return None

# ------------------------
# Build the KB for the Robert example
# ------------------------

def build_robert_kb():
    facts = []
    rules = []
    # Given facts:
```

```
    # Existential instantiation: we assume T1 is a fresh constant for the missile that A owns
    # Owns(A, T1)
    # Missile(T1)
    facts.append(parse_atom("Owns(A, T1)"))
    facts.append(parse_atom("Missile(T1)"))
    # American(Robert)
    facts.append(parse_atom("American(Robert)"))
    # Enemy(A, America)
    facts.append(parse_atom("Enemy(A, America)"))

    # Rules:
    # Missile(x) => Weapon(x)
    r1 = Rule([parse_atom("Missile(x)")], parse_atom("Weapon(x)"))
    rules.append(r1)

    # All missiles were sold to country A by Robert:
    # For all x: Missile(x) ∧ Owns(A, x) ⇒ Sells(Robert, x, A)
    r2 = Rule([parse_atom("Missile(x)"), parse_atom("Owns(A, x)")], parse_atom("Sells(Robert, x, A)"))
    rules.append(r2)

    # Enemy(x, America) => Hostile(x)
    r3 = Rule([parse_atom("Enemy(x, America)")], parse_atom("Hostile(x)"))
    rules.append(r3)

    # American(p) ∧ Weapon(q) ∧ Sells(p, q, r) ∧ Hostile(r) ⇒ Criminal(p)
    r4 = Rule([
        parse_atom("American(p)"),
        parse_atom("Weapon(q)"),
        parse_atom("Sells(p, q, r)"),
        parse_atom("Hostile(r)")
    ], parse_atom("Criminal(p)"))
    rules.append(r4)

    return facts, rules

# ------------------------
# Demo run
# ------------------------

if __name__ == "__main__":
    kb_facts, kb_rules = build_robert_kb()
    query = parse_atom("Criminal(Robert)")
    print("\nProving query:", query)
    subst = fol_fc_ask(kb_facts, kb_rules, query)
    print("\nFinal result:")
    if subst is None:
        print("Could NOT prove", query)
    else:
        print("Proved", query, "with substitution:", subst)
```

```
Proving query: Criminal(Robert)
Initial facts:
   Owns(A, T1)
   Missile(T1)
   American(Robert)
   Enemy(A, America)
[Iter 1] Inferred: Weapon(T1)
[Iter 1] Inferred: Sells(Robert, T1, A)
[Iter 1] Inferred: Hostile(A)
[Iter 2] Inferred: Criminal(Robert)
Query unified with inferred fact: Criminal(Robert)

Final result:
Proved Criminal(Robert) with substitution: {}
```

```
CAP_A = 4
CAP_B = 3
GOAL_A = 2

def is_goal(state):
    a, b = state
    return a == GOAL_A

def successors(state):
    a, b = state
    moves = []
    moves.append(((CAP_A, b), "FillA"))
    moves.append(((a, CAP_B), "FillB"))
    moves.append(((0, b), "EmptyA"))
    moves.append(((a, 0), "EmptyB"))
    t = min(a, CAP_B - b)
```

```
        moves.append(((a - t, b + t), "PourA->B"))
        t = min(b, CAP_A - a)
        moves.append(((a + t, b - t), "PourB->A"))
        seen = set()
        out = []
        for st, act in moves:
            if st != state and st not in seen:
                seen.add(st)
                out.append((st, act))
        return out

    def dfs_collect_all(node, limit, path_states, path_actions, seen_global, solutions):
        seen_global.add(node)
        if is_goal(node):
            solutions.append((path_states + [node], path_actions.copy()))
        if limit == 0:
            return
        for child, action in successors(node):
            if child not in path_states:
                dfs_collect_all(child, limit - 1,
                                path_states + [node],
                                path_actions + [action],
                                seen_global,
                                solutions)

    def find_all_solutions(start=(0,0), max_depth=12):
        solutions = []
        seen = set()
        dfs_collect_all(start, max_depth, [], [], seen, solutions)
        return solutions, seen

    def pretty_print_solutions(solutions, seen_states):
        print(f"Distinct states seen ({len(seen_states)}):")
        for s in sorted(seen_states):
            print(" ", s)
        print("\nTotal solutions found:", len(solutions))
        solutions_sorted = sorted(solutions, key=lambda x: (len(x[1]), x[0]))
        for i, (states, actions) in enumerate(solutions_sorted, start=1):
            print(f"\nSolution {i} - actions: {len(actions)}")
            for j in range(len(actions)):
                print(f"  {states[j]}  --{actions[j]}-->  {states[j+1]}")
            print("  Final state:", states[-1])

    if __name__ == "__main__":
        MAX_DEPTH = 12
        START = (0, 0)
        solutions, seen_states = find_all_solutions(START, max_depth=MAX_DEPTH)
        pretty_print_solutions(solutions, seen_states)
        if not solutions:
            print("\nNo solutions found within depth", MAX_DEPTH)
```

```
  (1, 0)  --PourA->B-->  (0, 1)
  (0, 1)  --FillA-->  (4, 1)
  (4, 1)  --PourA->B-->  (2, 3)
  (2, 3)  --EmptyA-->  (0, 3)
  (0, 3)  --PourB->A-->  (3, 0)
  (3, 0)  --FillB-->  (3, 3)
  (3, 3)  --PourB->A-->  (4, 2)
  (4, 2)  --EmptyA-->  (0, 2)
  (0, 2)  --PourB->A-->  (2, 0)
  Final state: (2, 0)

Solution 25 - actions: 12
  (0, 0)  --FillA-->  (4, 0)
  (4, 0)  --PourA->B-->  (1, 3)
  (1, 3)  --EmptyB-->  (1, 0)
  (1, 0)  --PourA->B-->  (0, 1)
  (0, 1)  --FillA-->  (4, 1)
  (4, 1)  --FillB-->  (4, 3)
  (4, 3)  --EmptyA-->  (0, 3)
  (0, 3)  --PourB->A-->  (3, 0)
  (3, 0)  --FillB-->  (3, 3)
  (3, 3)  --PourB->A-->  (4, 2)
  (4, 2)  --EmptyA-->  (0, 2)
  (0, 2)  --PourB->A-->  (2, 0)
  Final state: (2, 0)
```

```
#HILL MISPLACES TILES

GOAL = [1,2,3,4,5,6,7,8,0]

def h_misplaced(state):
    return sum(1 for i in range(9) if state[i] != 0 and state[i] != GOAL[i])

def neighbors(state):
    idx = state.index(0)
    r, c = divmod(idx, 3)
    out = []
    for dr, dc in [(-1,0),(1,0),(0,-1),(0,1)]:
        nr, nc = r+dr, c+dc
        if 0 <= nr < 3 and 0 <= nc < 3:
            ni = nr*3 + nc
            new = state.copy()
            new[idx], new[ni] = new[ni], new[idx]
            out.append(new)
    return out

def hill_climb_misplaced(start, max_steps=1000):
    current = start.copy()
    path = [current.copy()]
    for _ in range(max_steps):
        if h_misplaced(current) == 0:
            return True, path
        neigh = neighbors(current)
        best = min(neigh, key=h_misplaced)
        if h_misplaced(best) < h_misplaced(current):
            current = best
            path.append(current.copy())
        else:
            return False, path
    return False, path

def print_path(path):
    for i, s in enumerate(path):
        print(f"Step {i}, h={h_misplaced(s)}")
        for r in range(3):
            print(s[3*r:3*r+3])
        print()

if __name__ == "__main__":
    start = [1,2,3,4,5,6,0,7,8]
    solved, path = hill_climb_misplaced(start)
    print("Misplaced heuristic: Solved?", solved)
    print("Path length:", len(path)-1)
    print_path(path)
```

```
Misplaced heuristic: Solved? True
Path length: 2
Step 0, h=2
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]

Step 1, h=1
[1, 2, 3]
[4, 5, 6]
```

```
[7, 0, 8]

Step 2, h=0
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
```

```
#HILL MANHATTAN DISTANCE

GOAL = [1,2,3,4,5,6,7,8,0]

def h_manhattan(state):
    total = 0
    for i, tile in enumerate(state):
        if tile == 0:
            continue
        r1, c1 = divmod(i, 3)
        goal_i = GOAL.index(tile)
        r2, c2 = divmod(goal_i, 3)
        total += abs(r1 - r2) + abs(c1 - c2)
    return total

def neighbors(state):
    idx = state.index(0)
    r, c = divmod(idx, 3)
    out = []
    for dr, dc in [(-1,0),(1,0),(0,-1),(0,1)]:
        nr, nc = r+dr, c+dc
        if 0 <= nr < 3 and 0 <= nc < 3:
            ni = nr*3 + nc
            new = state.copy()
            new[idx], new[ni] = new[ni], new[idx]
            out.append(new)
    return out

def hill_climb_manhattan(start, max_steps=1000):
    current = start.copy()
    path = [current.copy()]
    for _ in range(max_steps):
        if h_manhattan(current) == 0:
            return True, path
        neigh = neighbors(current)
        best = min(neigh, key=h_manhattan)
        if h_manhattan(best) < h_manhattan(current):
            current = best
            path.append(current.copy())
        else:
            return False, path
    return False, path

def print_path(path):
    for i, s in enumerate(path):
        print(f"Step {i}, h={h_manhattan(s)}")
        for r in range(3):
            print(s[3*r:3*r+3])
        print()

if __name__ == "__main__":
    start = [1,2,3,4,5,6,0,7,8]
    solved, path = hill_climb_manhattan(start)
    print("Manhattan heuristic: Solved?", solved)
    print("Path length:", len(path)-1)
    print_path(path)
```

```
#A* for *PUZZLE
# a_star_8p_show_f.py
import heapq

def reconstruct_path(came_from, current):
    path = [current]
    actions = []
    while current in came_from:
        current, action = came_from[current]
        path.append(current)
        actions.append(action)
    path.reverse()
    actions.reverse()
    return path, actions

def a_star(start, goal_test, neighbors_fn, h_fn):
```

```
    open_heap = []
    entry_count = 0
    g_score = {start: 0}
    f_start = g_score[start] + h_fn(start)
    heapq.heappush(open_heap, (f_start, entry_count, start))
    entry_count += 1
    came_from = {}
    closed = set()

    while open_heap:
        f, _, current = heapq.heappop(open_heap)

        # show f, g, h for the popped node
        g = g_score.get(current, float('inf'))
        h = h_fn(current)
        print(f"POP: state={current}  g={g}  h={h}  f={g+h}")

        if goal_test(current):
            return reconstruct_path(came_from, current)

        if current in closed:
            continue
        closed.add(current)

        for neigh, action, cost in neighbors_fn(current):
            tentative_g = g_score[current] + cost
            if neigh in g_score and tentative_g >= g_score[neigh]:
                continue
            came_from[neigh] = (current, action)
            g_score[neigh] = tentative_g
            f_neigh = tentative_g + h_fn(neigh)
            heapq.heappush(open_heap, (f_neigh, entry_count, neigh))
            entry_count += 1
            # show candidate neighbor insertion
            print(f"  PUSH: neighbor={neigh}  action={action}  g={tentative_g}  h={h_fn(neigh)}  f={f_neigh}")

    return None, None

# 8-puzzle helpers
def neighbors_8p(state):
    state = list(state)
    idx = state.index(0)
    r, c = divmod(idx, 3)
    result = []
    moves = [(-1,0,"Up"), (1,0,"Down"), (0,-1,"Left"), (0,1,"Right")]
    for dr, dc, name in moves:
        nr, nc = r + dr, c + dc
        if 0 <= nr < 3 and 0 <= nc < 3:
            ni = nr*3 + nc
            new = state.copy()
            new[idx], new[ni] = new[ni], new[idx]
            result.append((tuple(new), name, 1))
    return result

def h_misplaced(state, goal):
    return sum(1 for i,t in enumerate(state) if t != 0 and t != goal[i])

def h_manhattan(state, goal_pos):
    total = 0
    for i, tile in enumerate(state):
        if tile == 0:
            continue
        goal_i = goal_pos[tile]
        r1, c1 = divmod(i, 3)
        r2, c2 = divmod(goal_i, 3)
        total += abs(r1 - r2) + abs(c1 - c2)
    return total

if __name__ == "__main__":
    # Provide initial and goal states here:
    start = (1,2,3,4,5,6,0,7,8)   # example initial (tuple)
    goal = (1,2,3,4,5,6,7,8,0)    # goal state

    # choose heuristic: misplaced or manhattan
    use = "manhattan"  # set to "misplaced" or "manhattan"

    if use == "misplaced":
        h_fn = lambda s: h_misplaced(s, goal)
    else:
        goal_pos = {val:i for i,val in enumerate(goal)}
        h_fn = lambda s: h_manhattan(s, goal_pos)
```

```
    path, actions = a_star(start, lambda s: s == goal, neighbors_8p, h_fn)

    if path:
        print("\nSOLUTION FOUND:")
        for i, state in enumerate(path):
            h = h_fn(state)
            g = sum(1 for _ in range(i))  # each step cost=1 -> g == i
            print(f"Step {i}: state={state}  g={i}  h={h}  f={i+h}")
            for r in range(3):
                print(state[3*r:3*r+3])
            if i < len(actions):
                print("  action ->", actions[i])
            print()
        print("Total moves:", len(actions))
    else:
        print("No solution found.")
```

```
POP: state=(1, 2, 3, 4, 5, 6, 0, 7, 8)  g=0  h=2  f=2
  PUSH: neighbor=(1, 2, 3, 0, 5, 6, 4, 7, 8)  action=Up  g=1  h=3  f=4
  PUSH: neighbor=(1, 2, 3, 4, 5, 6, 7, 0, 8)  action=Right  g=1  h=1  f=2
POP: state=(1, 2, 3, 4, 5, 6, 7, 0, 8)  g=1  h=1  f=2
  PUSH: neighbor=(1, 2, 3, 4, 0, 6, 7, 5, 8)  action=Up  g=2  h=2  f=4
  PUSH: neighbor=(1, 2, 3, 4, 5, 6, 7, 8, 0)  action=Right  g=2  h=0  f=2
POP: state=(1, 2, 3, 4, 5, 6, 7, 8, 0)  g=2  h=0  f=2

SOLUTION FOUND:
Step 0: state=(1, 2, 3, 4, 5, 6, 0, 7, 8)  g=0  h=2  f=2
(1, 2, 3)
(4, 5, 6)
(0, 7, 8)
  action -> Right

Step 1: state=(1, 2, 3, 4, 5, 6, 7, 0, 8)  g=1  h=1  f=2
(1, 2, 3)
(4, 5, 6)
(7, 0, 8)
  action -> Right

Step 2: state=(1, 2, 3, 4, 5, 6, 7, 8, 0)  g=2  h=0  f=2
(1, 2, 3)
(4, 5, 6)
(7, 8, 0)

Total moves: 2
```

```
#A* FOR NORMAL GRAPH

# a_star_graph_input.py
import heapq

def reconstruct_path(came_from, current):
    path = [current]
    actions = []
    while current in came_from:
        current, action = came_from[current]
        path.append(current)
        actions.append(action)
    path.reverse()
    actions.reverse()
    return path, actions

def a_star_graph(start, goal, neighbors_fn, h_fn):
    open_heap = []
    entry_count = 0
    g_score = {start: 0}
    f_start = g_score[start] + h_fn(start)
    heapq.heappush(open_heap, (f_start, entry_count, start))
    entry_count += 1
    came_from = {}
    closed = set()

    while open_heap:
        f, _, current = heapq.heappop(open_heap)
        g = g_score.get(current, float('inf'))
        h = h_fn(current)
        print(f"POP node={current}  g={g}  h={h}  f={g+h}")

        if current == goal:
            return reconstruct_path(came_from, current)

        if current in closed:
            continue
        closed.add(current)
```

```
            for neigh, cost in neighbors_fn(current):
                tentative_g = g + cost
                if neigh in g_score and tentative_g >= g_score[neigh]:
                    continue
                came_from[neigh] = (current, f"{current}->{neigh}")
                g_score[neigh] = tentative_g
                f_neigh = tentative_g + h_fn(neigh)
                heapq.heappush(open_heap, (f_neigh, entry_count, neigh))
                entry_count += 1
                print(f"  PUSH neighbor={neigh}  g={tentative_g}  h={h_fn(neigh)}  f={f_neigh}")

        return None, None

    def build_graph_from_input():
        print("Enter nodes (space-separated):")
        nodes = input().strip().split()
        graph = {n: [] for n in nodes}
        print("Enter edges one per line in format: node1 node2 cost  (enter blank line to stop)")
        while True:
            line = input().strip()
            if not line:
                break
            u, v, w = line.split()
            w = float(w)
            graph[u].append((v, w))
            # ask if undirected
            # for now we keep edges as directed; to make undirected, also add reverse
        return graph

    if __name__ == "__main__":
        print("Build graph:")
        graph = build_graph_from_input()

        print("Nodes:", list(graph.keys()))
        print("Enter start node:")
        start = input().strip()
        print("Enter goal node:")
        goal = input().strip()

        # read heuristic values for each node (user supplies a non-negative estimate)
        print("Enter heuristic value for each node in format: node value  (blank line stops)")
        heur = {}
        while True:
            line = input().strip()
            if not line:
                break
            node, val = line.split()
            heur[node] = float(val)

        # fallback for missing heuristics: 0
        def h_fn(n):
            return heur.get(n, 0.0)

        def neighbors_fn(n):
            return graph.get(n, [])

        path, actions = a_star_graph(start, goal, neighbors_fn, h_fn)

        if path:
            print("\nSOLUTION PATH:")
            for i, node in enumerate(path):
                g = i if i < 1 else "see g-score"  # we don't track per-step g here in print; below we'll compute g
                h = h_fn(node)
                print(f"  Step {i}: {node}  h={h}")
            print("Moves:", len(actions))
            print("Full path (nodes):", path)
        else:
            print("No path found.")
```

```
Build graph:
Enter nodes (space-separated):
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
/tmp/ipython-input-2124041372.py in <cell line: 0>()
     69 if __name__ == "__main__":
     70     print("Build graph:")
---> 71     graph = build_graph_from_input()
     72
     73     print("Nodes:", list(graph.keys()))

                                 2 frames
/usr/local/lib/python3.12/dist-packages/ipykernel/kernelbase.py in _input_request(self, prompt, ident, parent,
password)
   1217             except KeyboardInterrupt:
   1218                 # re-raise KeyboardInterrupt, to truncate traceback
-> 1219                 raise KeyboardInterrupt("Interrupted by user") from None
   1220             except Exception:
   1221                 self.log.warning("Invalid Message:", exc_info=True)

KeyboardInterrupt: Interrupted by user
```

```
#ENTIALMENT

from itertools import product

# Evaluate a single clause
def eval_clause(clause, model):
    # unary NOT: 1P
    if clause.startswith("1"):
        return not model[clause[1:]]

    # binary: A2B, A3B, A4B, A5B
    for i, ch in enumerate(clause):
        if ch in "2345":
            A = clause[:i]
            op = ch
            B = clause[i+1:]

            p = model[A]
            q = model[B]

            if op == "2": return p and q
            if op == "3": return p or q
            if op == "4": return (not p) or q
            if op == "5": return p == q

    # single symbol
    return model[clause]

# Evaluate KB = conjunction of all clauses
def eval_KB(KB, model):
    return all(eval_clause(c, model) for c in KB)

# Truth table entailment
def tt_entails(KB, alpha, symbols):
    for vals in product([False, True], repeat=len(symbols)):
        model = dict(zip(symbols, vals))
        if eval_KB(KB, model) and not eval_clause(alpha, model):
            print("Counterexample:", model)
            return False
    return True

# ---------------- MAIN ----------------

print("Enter KB clauses separated by commas ONLY. Example:")
print("   Q4P, P41Q, R3Q\n")

KB = [c.strip() for c in input("KB: ").split(",")]

alpha = input("Enter alpha clause: ").strip()

# Collect symbols
symbols = set()

def collect(cl):
    if cl.startswith("1"):
        symbols.add(cl[1:])
    else:
        temp = ""
        for ch in cl:
            if ch in "2345":
```

```
                symbols.add(temp)
                temp = ""
            else:
                temp += ch
        symbols.add(temp)

for c in KB:
    collect(c)
collect(alpha)

symbols = sorted(symbols)

# Print truth table header
print("\nTruth Table:")
header = symbols + KB + ["KB", alpha]
print(" | ".join(header))
print("-" * 80)

# Print rows
for vals in product([False, True], repeat=len(symbols)):
    model = dict(zip(symbols, vals))
    sym_vals = [("T" if model[s] else "F") for s in symbols]
    clause_vals = [("T" if eval_clause(c, model) else "F") for c in KB]
    kb_val = "T" if eval_KB(KB, model) else "F"
    alpha_val = "T" if eval_clause(alpha, model) else "F"

    print(" | ".join(sym_vals + clause_vals + [kb_val, alpha_val]))

# Final entailment result
print("\nDoes KB entail alpha? →", tt_entails(KB, alpha, symbols))
```

```
Enter KB clauses separated by commas ONLY. Example:
   Q4P, P41Q, R3Q

---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
/tmp/ipython-input-857495725.py in <cell line: 0>()
     43 print("   Q4P, P41Q, R3Q\n")
     44
---> 45 KB = [c.strip() for c in input("KB: ").split(",")]
     46
     47 alpha = input("Enter alpha clause: ").strip()

                                    1 frames
/usr/local/lib/python3.12/dist-packages/ipykernel/kernelbase.py in _input_request(self, prompt, ident, parent,
password)
   1217             except KeyboardInterrupt:
   1218                 # re-raise KeyboardInterrupt, to truncate traceback
-> 1219                 raise KeyboardInterrupt("Interrupted by user") from None
   1220             except Exception:
   1221                 self.log.warning("Invalid Message:", exc_info=True)

KeyboardInterrupt: Interrupted by user
```

```
#UNIFIFCATION

# robust_unifier.py
import re
from typing import List, Union, Dict, Optional, Tuple
from dataclasses import dataclass

# ---------- term classes ----------
@dataclass(frozen=True)
class Var:
    name: str
    def __repr__(self): return f"?{self.name}"

@dataclass(frozen=True)
class Const:
    name: str
    def __repr__(self): return self.name

@dataclass(frozen=True)
class Func:
    name: str
    args: tuple
    def __repr__(self):
        return f"{self.name}({', '.join(map(repr, self.args))})"

Term = Union[Var, Const, Func]
Subst = Dict[Var, Term]
```

```
# ---------- helper: normalize uppercase vars (X -> ?X) ----------
def normalize_vars_line(line: str) -> str:
    # replace standalone uppercase-start tokens with ?Token, but avoid replacing function names followed by '('
    def repl(m):
        tok = m.group(0)
        # if followed by '(' it's a function name; keep as is
        return "?" + tok
    # replace tokens that are standalone identifiers and start with uppercase letter
    return re.sub(r'\b([A-Z][A-Za-z0-9_]*)\b', lambda m: ("?" + m.group(1)), line)

# ---------- split top-level by commas (ignore commas inside parentheses) ----------
def split_top_level_commas(s: str) -> List[str]:
    parts = []
    buf = []
    depth = 0
    for ch in s:
        if ch == ',' and depth == 0:
            part = ''.join(buf).strip()
            if part: parts.append(part)
            buf = []
            continue
        buf.append(ch)
        if ch == '(':
            depth += 1
        elif ch == ')':
            depth -= 1
            if depth < 0:
                raise ValueError("Unmatched closing parenthesis ')'")
    if depth > 0:
        raise ValueError("Missing closing parenthesis ')'")
    last = ''.join(buf).strip()
    if last:
        parts.append(last)
    return parts

# ---------- tokenizer & parser ----------
TOKEN_RE = re.compile(r'\s*([A-Za-z0-9_?]+|\(|\)|,)\s*')

def tokenize(s: str) -> List[str]:
    return [m.group(1) for m in TOKEN_RE.finditer(s)]

def parse_term(tokens: List[str], pos: int = 0) -> Tuple[Term, int]:
    if pos >= len(tokens):
        raise ValueError("Unexpected end of input while parsing term")
    tok = tokens[pos]
    # variable (starts with ?)
    if tok.startswith('?'):
        return Var(tok[1:]), pos + 1
    # function or constant
    if pos + 1 < len(tokens) and tokens[pos + 1] == '(':
        name = tok
        pos += 2  # skip name and '('
        args = []
        # empty argument list
        if pos < len(tokens) and tokens[pos] == ')':
            return Func(name, tuple()), pos + 1
        while True:
            arg, pos = parse_term(tokens, pos)
            args.append(arg)
            if pos >= len(tokens):
                raise ValueError("Missing closing parenthesis ')'")
            if tokens[pos] == ',':
                pos += 1
                continue
            if tokens[pos] == ')':
                pos += 1
                break
            raise ValueError(f"Unexpected token '{tokens[pos]}' in function arguments")
        return Func(name, tuple(args)), pos
    # constant
    return Const(tok), pos + 1

def parse_from_string(s: str) -> Term:
    tokens = tokenize(s)
    term, pos = parse_term(tokens, 0)
    if pos != len(tokens):
        extra = " ".join(tokens[pos:])
        raise ValueError(f"Extra tokens after parsing: {extra}")
    return term

# ---------- substitution utilities ----------
def apply_subst(term: Term, subst: Subst) -> Term:
```

```
        if isinstance(term, Var):
            if term in subst:
                return apply_subst(subst[term], subst)
            return term
        if isinstance(term, Const):
            return term
        return Func(term.name, tuple(apply_subst(a, subst) for a in term.args))

    def occurs_check(var: Var, term: Term, subst: Subst) -> bool:
        term = apply_subst(term, subst)
        if term == var:
            return True
        if isinstance(term, Func):
            return any(occurs_check(var, a, subst) for a in term.args)
        return False

    def compose_subst(s1: Subst, s2: Subst) -> Subst:
        new = {}
        for v, t in s1.items():
            new[v] = apply_subst(t, s2)
        for v, t in s2.items():
            new[v] = t
        return new

    # ---------- unify core ----------
    def unify_var(var: Var, term: Term, subst: Subst) -> Optional[Subst]:
        if var in subst:
            return unify(subst[var], term, subst)
        if isinstance(term, Var) and term in subst:
            return unify(var, subst[term], subst)
        if occurs_check(var, term, subst):
            return None
        new_subst = dict(subst)
        new_subst[var] = term
        # normalize existing mappings
        for v in list(new_subst.keys()):
            new_subst[v] = apply_subst(new_subst[v], {var: term})
        return new_subst

    def unify(x: Term, y: Term, subst: Optional[Subst]=None) -> Optional[Subst]:
        if subst is None:
            subst = {}
        x = apply_subst(x, subst)
        y = apply_subst(y, subst)
        if x == y:
            return subst
        if isinstance(x, Var):
            return unify_var(x, y, subst)
        if isinstance(y, Var):
            return unify_var(y, x, subst)
        if isinstance(x, Func) and isinstance(y, Func):
            if x.name != y.name or len(x.args) != len(y.args):
                return None
            cur = subst
            for a, b in zip(x.args, y.args):
                s = unify(a, b, cur)
                if s is None:
                    return None
                cur = s
            return cur
        return None

    # unify n-ary list of terms
    def unify_all(terms: List[Term]) -> Optional[Subst]:
        if not terms:
            return {}
        terms = [t for t in terms]
        subst: Subst = {}
        base = terms[0]
        for i in range(1, len(terms)):
            base = apply_subst(base, subst)
            nextt = apply_subst(terms[i], subst)
            s = unify(base, nextt, subst)
            if s is None:
                return None
            subst = compose_subst(subst, s)
            base = apply_subst(base, s)
        return subst

    def subst_to_string(subst: Subst) -> str:
        if not subst:
            return "{}"
```

```
        parts = []
        for v, t in subst.items():
            parts.append(f"{repr(v)}/{repr(t)}")
        return "{" + ", ".join(parts) + "}"

    # ---------- CLI ----------
    if __name__ == "__main__":
        print("Unifier (variables start with '?' or uppercase identifiers will be treated as variables).")
        print("Enter expressions separated by top-level commas.")
        print("Examples:")
        print("  Knows(John, ?x), Knows(?y, Mary)")
        print("  f(?x, a), f(b, ?y), f(?z, ?z)")
        print("  p(b,X,f(g(Z))), p(Z,f(Y),f(Y))   (uppercase vars auto-normalized)\n")

        raw = input("Enter terms: ").strip()
        if not raw:
            print("No input. Exiting."); raise SystemExit

        # normalize uppercase vars (X -> ?X)
        normalized = normalize_vars_line(raw)

        # split by top-level commas and parse each term
        try:
            parts = split_top_level_commas(normalized)
        except ValueError as e:
            print("Parse error:", e)
            raise SystemExit

        try:
            terms = [parse_from_string(p) for p in parts]
        except Exception as e:
            print("Parse error:", e)
            raise SystemExit

        # unify all
        result = unify_all(terms)
        if result is None:
            print("\nUNIFICATION FAILED")
        else:
            print("\nUNIFICATION SUCCESS")
            print("Substitution:", subst_to_string(result))
            print("All terms become:", apply_subst(terms[0], result))
```

```
Unifier (variables start with '?' or uppercase identifiers will be treated as variables).
Enter expressions separated by top-level commas.
Examples:
  Knows(John, ?x), Knows(?y, Mary)
  f(?x, a), f(b, ?y), f(?z, ?z)
  p(b,X,f(g(Z))), p(Z,f(Y),f(Y))   (uppercase vars auto-normalized)

Enter terms: p(b,X,f(g(Z))), p(Z,f(Y),f(Y))

UNIFICATION SUCCESS
Substitution: {?Z/b, ?X/f(g(b)), ?Y/g(b)}
All terms become: p(b, f(g(b)), f(g(b)))
```

```
# fol_forward_chaining.py
import re
from typing import List, Tuple, Dict, Optional, Union
from dataclasses import dataclass
from itertools import product
import sys

# --------------------------
# Term & Predicate classes
# --------------------------
@dataclass(frozen=True)
class Var:
    name: str
    def __repr__(self): return f"?{self.name}"
@dataclass(frozen=True)
class Const:
    name: str
    def __repr__(self): return self.name
@dataclass(frozen=True)
class Func:
    name: str
    args: tuple
    def __repr__(self):
        return f"{self.name}({', '.join(map(repr, self.args))})"

Term = Union[Var, Const, Func]
```

```
@dataclass(frozen=True)
class Pred:
    name: str
    args: tuple
    def __repr__(self):
        return f"{self.name}({', '.join(map(repr, self.args))})"

# -------------------------
# Parser (variables start with ?). Uppercase tokens auto-normalized to variables.
# -------------------------
TOKEN_RE = re.compile(r'\s*([A-Za-z0-9_?]+|\(|\)|,|->|=>|&|\^)\s*')

def normalize_uppercase_vars(line: str) -> str:
    # convert standalone uppercase-start tokens to ?X
    def repl(m):
        tok = m.group(0)
        # if token followed by '(' it's a function/predicate name - leave as is
        return "?" + tok
    # Replace only identifiers not followed by '('
    return re.sub(r'\b([A-Z][A-Za-z0-9_]*)\b', lambda m: "?" + m.group(1), line)

def tokenize(s: str) -> List[str]:
    tokens = [m.group(1) for m in TOKEN_RE.finditer(s)]
    return tokens

def parse_term(tokens: List[str], pos: int = 0):
    if pos >= len(tokens):
        raise ValueError("Unexpected end of term")
    tok = tokens[pos]
    if tok.startswith('?'):
        return Var(tok[1:]), pos+1
    # function or constant
    if pos+1 < len(tokens) and tokens[pos+1] == '(':
        name = tok
        pos += 2
        args = []
        if pos < len(tokens) and tokens[pos] == ')':
            return Func(name, tuple()), pos+1
        while True:
            arg, pos = parse_term(tokens, pos)
            args.append(arg)
            if pos >= len(tokens):
                raise ValueError("Missing closing ) in function")
            if tokens[pos] == ',':
                pos += 1
                continue
            if tokens[pos] == ')':
                pos += 1
                break
            raise ValueError(f"Unexpected token {tokens[pos]} in function args")
        return Func(name, tuple(args)), pos
    # constant
    return Const(tok), pos+1

def parse_pred(s: str) -> Pred:
    s = s.strip()
    s = normalize_uppercase_vars(s)
    tokens = tokenize(s)
    # predicate must be name '(' ... ')'
    if '(' not in tokens:
        # allow zero-arg predicate as name()
        if len(tokens) == 1:
            return Pred(tokens[0], tuple())
        raise ValueError(f"Invalid predicate syntax: {s}")
    pred, pos = parse_term(tokens, 0)
    if not isinstance(pred, Func):
        # allow a bare constant as 0-arg predicate? treat as predicate with zero args
        if isinstance(pred, Const):
            return Pred(pred.name, tuple())
        raise ValueError(f"Predicate must be function-like: {s}")
    if pos != len(tokens):
        extra = " ".join(tokens[pos:])
        raise ValueError(f"Extra tokens after predicate: {extra}")
    return Pred(pred.name, pred.args)

# parse a clause line: either a fact (atomic) or rule: body -> head
def parse_clause_line(line: str):
    if '->' in line:
        parts = line.split('->', 1)
    elif '=>' in line:
        parts = line.split('=>', 1)
```

```
        else:
            parts = [line.strip()]
        if len(parts) == 1:
            # fact
            head = parse_pred(parts[0])
            return ("fact", head)
        else:
            body_text = parts[0].strip()
            head_text = parts[1].strip()
            # split body by &, ^ or commas at top level (simple)
            # allow 'A & B' or 'A, B' or 'A ^ B'
            body_parts = re.split(r'\s*(?:&|\^|,)\s*', body_text)
            body_preds = [parse_pred(p) for p in body_parts if p.strip() != ""]
            head_pred = parse_pred(head_text)
            return ("rule", (body_preds, head_pred))

    # -------------------------
    # Substitution / unification utilities
    # -------------------------
    Subst = Dict[Var, Term]

    def apply_subst_term(t: Term, subst: Subst) -> Term:
        if isinstance(t, Var):
            if t in subst:
                return apply_subst_term(subst[t], subst)
            return t
        if isinstance(t, Const):
            return t
        # Func
        return Func(t.name, tuple(apply_subst_term(a, subst) for a in t.args))

    def apply_subst_pred(p: Pred, subst: Subst) -> Pred:
        return Pred(p.name, tuple(apply_subst_term(a, subst) for a in p.args))

    def occurs_check(v: Var, t: Term, subst: Subst) -> bool:
        t = apply_subst_term(t, subst)
        if t == v:
            return True
        if isinstance(t, Func):
            return any(occurs_check(v, a, subst) for a in t.args)
        return False

    def compose(s1: Subst, s2: Subst) -> Subst:
        # return s = s2 ° s1 (apply s1 then s2)
        new = {}
        for v,t in s1.items():
            new[v] = apply_subst_term(t, s2)
        for v,t in s2.items():
            new[v] = t
        return new

    def unify_terms(x: Term, y: Term, subst: Optional[Subst]=None) -> Optional[Subst]:
        if subst is None:
            subst = {}
        x = apply_subst_term(x, subst)
        y = apply_subst_term(y, subst)
        if x == y:
            return subst
        if isinstance(x, Var):
            return unify_var(x, y, subst)
        if isinstance(y, Var):
            return unify_var(y, x, subst)
        if isinstance(x, Func) and isinstance(y, Func):
            if x.name != y.name or len(x.args) != len(y.args):
                return None
            cur = subst
            for a,b in zip(x.args, y.args):
                s = unify_terms(a,b,cur)
                if s is None:
                    return None
                cur = s
            return cur
        return None

    def unify_var(v: Var, t: Term, subst: Subst) -> Optional[Subst]:
        if v in subst:
            return unify_terms(subst[v], t, subst)
        if isinstance(t, Var) and t in subst:
            return unify_terms(v, subst[t], subst)
        if occurs_check(v, t, subst):
            return None
        new = dict(subst)
```

```
        new[v] = t
        # normalize existing mappings
        for k in list(new.keys()):
            new[k] = apply_subst_term(new[k], {v:t})
        return new

    # ------------------------
    # Standardize variables apart
    # ------------------------
    _counter = 0
    def fresh_var_name(base: str="v"):
        global _counter
        _counter += 1
        return f"?{base}_{_counter}"

    def standardize_rule(body: List[Pred], head: Pred) -> Tuple[List[Pred], Pred]:
        # rename all variables in body and head so they are fresh (avoid clashes)
        varmap: Dict[str,str] = {}
        def rename_term(t: Term) -> Term:
            if isinstance(t, Var):
                if t.name not in varmap:
                    varmap[t.name] = fresh_var_name(t.name)
                return Var(varmap[t.name][1:])  # store without leading ?
            if isinstance(t, Const):
                return t
            return Func(t.name, tuple(rename_term(a) for a in t.args))
        new_body = []
        for p in body:
            new_body.append(Pred(p.name, tuple(rename_term(a) for a in p.args)))
        new_head = Pred(head.name, tuple(rename_term(a) for a in head.args))
        return new_body, new_head

    # ------------------------
    # Matching rule premises to known facts (backtracking)
    # returns list of substitutions that make all premises true given known_facts
    # ------------------------
    def find_matching_substitutions(premises: List[Pred], known_facts: List[Pred]) -> List[Subst]:
        results: List[Subst] = []
        def backtrack(i: int, cur_subst: Subst):
            if i == len(premises):
                results.append(cur_subst)
                return
            premise = apply_subst_pred(premises[i], cur_subst)
            # try unify premise with any known fact
            for fact in known_facts:
                s = unify_predicates(premise, fact, dict(cur_subst))
                if s is not None:
                    backtrack(i+1, s)
        backtrack(0, {})
        return results

    def unify_predicates(p: Pred, f: Pred, subst: Subst) -> Optional[Subst]:
        if p.name != f.name or len(p.args) != len(f.args):
            return None
        cur = subst
        for a,b in zip(p.args, f.args):
            cur = unify_terms(a,b,cur)
            if cur is None:
                return None
        return cur

    # ------------------------
    # Forward chaining main
    # ------------------------
    def forward_chain(KB_clauses: List[Tuple[str, Union[Pred, Tuple[List[Pred],Pred]]]], query: Pred, max_iters=500)
        # KB_clauses: list of ("fact", Pred) or ("rule", (bodyList, headPred))
        # known facts initially contain all facts from KB
        known_facts: List[Pred] = []
        rules = []
        for kind, payload in KB_clauses:
            if kind == "fact":
                known_facts.append(payload)
            else:
                rules.append(payload)
        trace = []  # list of inference steps

        # If query already present
        for f in known_facts:
            s = unify_predicates(query, f, {})
            if s is not None:
                return True, s, trace
```

```
    for it in range(max_iters):
        new_inferred: List[Tuple[Pred, Tuple[List[Pred],Pred], Subst, List[Pred]]] = []
        for body, head in rules:
            # standardize variables apart
            sbody, shead = standardize_rule(body, head)
            # find substitutions that satisfy all premises
            subs = find_matching_substitutions(sbody, known_facts)
            for theta in subs:
                # instantiate head
                newfact = apply_subst_pred(shead, theta)
                # check if already known (structural equality)
                if not any(eq_preds(newfact, k) for k in known_facts) and not any(eq_preds(newfact, ni[0]) for n
                    new_inferred.append((newfact, (sbody, shead), theta, []))
        if not new_inferred:
            return False, None, trace
        # add inferred to known_facts and record trace
        for nf, (sbody, shead), theta, _ in new_inferred:
            known_facts.append(nf)
            # record which facts matched (for readability, list matching facts)
            matched_facts = []
            # find facts that were used to generate nf under theta (approx)
            for prem in sbody:
                prem_inst = apply_subst_pred(prem, theta)
                # find a fact equal to prem_inst
                for kf in known_facts:
                    if eq_preds(prem_inst, kf):
                        matched_facts.append(kf)
                        break
            trace.append({
                "inferred": nf,
                "rule_body": sbody,
                "rule_head": shead,
                "substitution": theta,
                "matched_facts": matched_facts
            })
            # check query
            s = unify_predicates(query, nf, {})
            if s is not None:
                return True, theta, trace
    return False, None, trace

def eq_terms(a: Term, b: Term) -> bool:
    if type(a) != type(b):
        return False
    if isinstance(a, Var):
        return a.name == b.name
    if isinstance(a, Const):
        return a.name == b.name
    if isinstance(a, Func):
        return a.name == b.name and len(a.args) == len(b.args) and all(eq_terms(x,y) for x,y in zip(a.args,b.arg
    return False

def eq_preds(a: Pred, b: Pred) -> bool:
    return a.name == b.name and len(a.args) == len(b.args) and all(eq_terms(x,y) for x,y in zip(a.args,b.args))

# ------------------------
# Pretty print helpers
# ------------------------
def subst_str(subst: Subst) -> str:
    if not subst:
        return "{}"
    parts=[]
    for v,t in subst.items():
        parts.append(f"{repr(v)}/{repr(t)}")
    return "{" + ", ".join(parts) + "}"

def print_trace(trace):
    for i, step in enumerate(trace, start=1):
        print(f"Step {i}: inferred {step['inferred']}")
        print("  using rule head:", step['rule_head'])
        print("  rule body (standardized):", ", ".join(map(str, step['rule_body'])))
        print("  matched facts:", ", ".join(map(str, step['matched_facts'])))
        print("  substitution:", subst_str(step['substitution']))
        print()

# ------------------------
# CLI
# ------------------------
if __name__ == "__main__":
    print("Forward-chaining FOL (definite clauses)")
    print("Syntax notes:")
    print("  - Variables start with '?' (or uppercase will be normalized). Example: ?x or X")
```

```
print("  - Functions/predicates: P(a,?x), Knows(John,?y), f(a,g(?x))")
print("  - Rules: body -> head   (use & , or ^ as conjunction). Example:")
print("       Q(?x) & P(?x,?y) -> R(?y)")
print("  - Facts: P(a,b) (just write an atom)")
print("Enter KB lines (one per line). Leave blank line to end input.\n")

lines=[]
while True:
    try:
        line = input().strip()
    except EOFError:
        break
    if line == "":
        break
    lines.append(line)

KB_clauses=[]
for ln in lines:
    try:
        parsed = parse_clause_line(ln)
        KB_clauses.append(parsed)
    except Exception as e:
        print("Parse error in line:", ln)
        print(e)
        sys.exit(1)

qline = input("Enter query (one atomic formula): ").strip()
try:
    query = parse_pred(qline)
except Exception as e:
    print("Query parse error:", e)
    sys.exit(1)

success, subst, trace = forward_chain(KB_clauses, query)
print("\n--- RESULT ---")
if success:
    print("Query is entailed by KB")
    print("One substitution that derives it:", subst_str(subst))
else:
    print("Query NOT entailed by KB (no more inferences).")
print("\n--- TRACE ---")
if trace:
    print_trace(trace)
else:
    print("No inferences made.")
```

Next steps: Explain error

```
Forward-chaining FOL (definite clauses)
Syntax notes:
```

```
#ALPHA BETA PRUNING


# alpha_beta_bottom_up_pairs_pruned_leaf_values.py
import math
import re
from typing import List, Optional

class Node:
    def __init__(self, name: str, depth: int):
        self.name = name
        self.depth = depth
        self.left: Optional['Node'] = None
        self.right: Optional['Node'] = None
        self.value: Optional[float] = None  # for leaves after input
        # tracing fields
        self.alpha = -math.inf
        self.beta = math.inf
        self.best_choice: Optional[str] = None
        self.pruned = False

    def is_leaf(self):
        return self.left is None and self.right is None

    def __repr__(self):
        if self.is_leaf():
            return f"{self.name}({self.value})"
        return f"{self.name}({self.left.name},{self.right.name})"


# build full binary tree bottom-up from leaf values (pairing left-to-right)
def build_tree_from_leaves(leaf_values: List[float]) -> Node:
    # create leaf nodes
    nodes = []
    for i, v in enumerate(leaf_values):
        n = Node(f"n{i}", depth=0)  # temporary depth; will be reassigned
        n.value = float(v)
        nodes.append(n)
    # bottom-up pairing
    level = 0
    while len(nodes) > 1:
        parents = []
        for i in range(0, len(nodes), 2):
            left = nodes[i]
            right = nodes[i+1]
            p = Node(f"n_p{level}_{i//2}", depth=level+1)
            p.left = left
            p.right = right
            parents.append(p)
        nodes = parents
        level += 1
    # final node is root; now assign correct depths (root depth = D)
    def assign_depths(node: Node, d: int):
        node.depth = d
        if not node.is_leaf():
            assign_depths(node.left, d-1)
            assign_depths(node.right, d-1)
    D = level
    assign_depths(nodes[0], D)
    return nodes[0]


# alpha-beta pruning (root = MAX). Tracks pruned nodes list and sets node.pruned True for pruned subtrees.
def alpha_beta(node: Node, maximizing: bool, alpha: float, beta: float, pruned_nodes: List[str]) -> float:
    node.alpha = alpha
    node.beta = beta
    if node.is_leaf():
        node.best_choice = None
        return node.value

    if maximizing:
        v = -math.inf
        # evaluate left child
        left_val = alpha_beta(node.left, False, alpha, beta, pruned_nodes)
        v = max(v, left_val)
        # update best choice if left gives v
        node.best_choice = node.left.name if left_val >= v else node.best_choice
        alpha = max(alpha, v)
        node.alpha = alpha
```

```
            # cutoff test
            if v >= beta:
                # prune right subtree entirely
                if node.right is not None:
                    mark_pruned_subtree(node.right)
                    pruned_nodes.append(node.right.name)
                node.best_choice = node.left.name
                return v
            # evaluate right child
            right_val = alpha_beta(node.right, False, alpha, beta, pruned_nodes)
            if right_val > v:
                v = right_val
                node.best_choice = node.right.name
            alpha = max(alpha, v)
            node.alpha = alpha
            return v
        else:
            v = math.inf
            left_val = alpha_beta(node.left, True, alpha, beta, pruned_nodes)
            v = min(v, left_val)
            node.best_choice = node.left.name if left_val <= v else node.best_choice
            beta = min(beta, v)
            node.beta = beta
            if v <= alpha:
                # prune right subtree
                if node.right is not None:
                    mark_pruned_subtree(node.right)
                    pruned_nodes.append(node.right.name)
                node.best_choice = node.left.name
                return v
            right_val = alpha_beta(node.right, True, alpha, beta, pruned_nodes)
            if right_val < v:
                v = right_val
                node.best_choice = node.right.name
            beta = min(beta, v)
            node.beta = beta
            return v

    # mark all nodes in subtree as pruned=True
    def mark_pruned_subtree(node: Node):
        node.pruned = True
        if not node.is_leaf():
            mark_pruned_subtree(node.left)
            mark_pruned_subtree(node.right)

    # pretty-print tree trace
    def print_trace(root: Node):
        print("\nTRACE (node : depth, alpha, beta, best_choice, pruned_flag, value_if_leaf):\n")
        def dfs(n: Node, indent=0):
            pad = "  " * indent
            if n.is_leaf():
                print(f"{pad}{n.name} : depth={n.depth}, alpha={n.alpha}, beta={n.beta}, pruned={n.pruned}, value={n
            else:
                print(f"{pad}{n.name} : depth={n.depth}, alpha={n.alpha}, beta={n.beta}, best={n.best_choice}, prune
                dfs(n.left, indent+1)
                dfs(n.right, indent+1)
        dfs(root)

    # collect pruned leaf values in left-to-right order (unique)
    def collect_pruned_leaf_values(root: Node) -> List[float]:
        pruned_values = []
        def dfs(n: Node, ancestor_pruned: bool):
            now_pruned = ancestor_pruned or n.pruned
            if n.is_leaf():
                if now_pruned:
                    pruned_values.append(n.value)
            else:
                dfs(n.left, now_pruned)
                dfs(n.right, now_pruned)
        dfs(root, False)
        # unique preserving order
        seen = set()
        unique = []
        for v in pruned_values:
            if v not in seen:
                seen.add(v)
                unique.append(v)
        return unique

    def main():
        print("Alpha-Beta on a full binary tree built from leaf pairs (bottom-up).")
        # read depth
```

```
    while True:
        try:
            D = int(input("Enter tree depth D (root depth = 0). Must be >= 0: ").strip())
            if D >= 0:
                break
        except:
            pass
        print("Invalid. Enter integer >= 0.")

    n_leaves = 2 ** D
    print(f"Depth {D} requires {n_leaves} terminal values (leaves).")
    print("Enter the leaf values left-to-right as numbers, separated by spaces or commas.")
    while True:
        s = input(f"Enter {n_leaves} numbers: ").strip()
        if not s:
            print("Please enter values.")
            continue
        tokens = [t for t in s.replace(",", " ").split() if t.strip() != ""]
        if len(tokens) != n_leaves:
            print(f"You entered {len(tokens)} values but need {n_leaves}. Try again.")
            continue
        try:
            leaf_values = [float(x) for x in tokens]
            break
        except:
            print("All values must be numbers. Try again.")

    # build tree
    root = build_tree_from_leaves(leaf_values)

    # run alpha-beta (root is MAX).
    pruned_nodes: List[str] = []
    value = alpha_beta(root, True, -math.inf, math.inf, pruned_nodes)

    print("\n--- RESULT ---")
    print(f"Root best utility = {value}")
    print(f"Root best child (name) = {root.best_choice}")
    print_trace(root)

    # collect pruned leaf values (unique, left-to-right)
    pruned_leaf_values = collect_pruned_leaf_values(root)
    print("\nPruned leaf values (left-to-right, unique):")
    if pruned_leaf_values:
        print(", ".join(str(int(v) if v.is_integer() else str(v)) for v in pruned_leaf_values))
    else:
        print("No leaves were pruned.")


if __name__ == "__main__":
    main()
```

```
depth = 0). Must be >= 0:  Enter tree depth D (root depth = 0). Must be >= 0:
```

```
# ============================================================
#   FOL → CNF Converter (Google Colab Version)
#   Usage:
#       convert_to_cnf("∀x (P(x) → ∃y (Q(y) ∧ R(x,y)))")
# ============================================================

import re
import itertools

# ============================================================
# AST Node Classes
# ============================================================

class Var:
    def __init__(self, name): self.name = name
    def __repr__(self): return self.name

class Const:
    def __init__(self, name): self.name = name
    def __repr__(self): return self.name

class Func:
    def __init__(self, name, args): self.name, self.args = name, args
```

```
    def __repr__(self): return f"{self.name}({', '.join(map(str,self.args))})"

class Pred:
    def __init__(self, name, args): self.name, self.args = name, args
    def __repr__(self): return f"{self.name}({', '.join(map(str,self.args))})"

class Not:
    def __init__(self, sub): self.sub = sub
    def __repr__(self): return f"¬{self.sub}"

class And:
    def __init__(self, left, right): self.left, self.right = left, right
    def __repr__(self): return f"({self.left} ∧ {self.right})"

class Or:
    def __init__(self, left, right): self.left, self.right = left, right
    def __repr__(self): return f"({self.left} ∨ {self.right})"

class Implies:
    def __init__(self, left, right): self.left, self.right = left, right
    def __repr__(self): return f"({self.left} → {self.right})"

class Iff:
    def __init__(self, left, right): self.left, self.right = left, right
    def __repr__(self): return f"({self.left} ↔ {self.right})"

class Forall:
    def __init__(self, var, sub): self.var, self.sub = var, sub
    def __repr__(self): return f"∀{self.var}.{self.sub}"

class Exists:
    def __init__(self, var, sub): self.var, self.sub = var, sub
    def __repr__(self): return f"∃{self.var}.{self.sub}"

# ============================================================
# TOKENIZER
# ============================================================

TOKEN_SPEC = [
    ('SKIP',        r'[ \t\n]+'),
    ('FORALL',      r'∀|forall\b'),
    ('EXISTS',      r'∃|exists\b'),
    ('NOT',         r'¬|~'),
    ('AND',         r'∧|&'),
    ('OR',          r'∨|\|'),
    ('IMPLIES',     r'→|=>'),
    ('IFF',         r'↔|<=>'),
    ('LPAREN',      r'\('),
    ('RPAREN',      r'\)'),
    ('COMMA',       r','),
    ('IDENT',       r"[A-Za-z_][A-Za-z0-9_]*"),
]

MASTER = re.compile('|'.join(f'(?P<{name}>{pattern})' for name, pattern in TOKEN_SPEC))

def tokenize(s):
    pos = 0
    tokens = []
    while pos < len(s):
        m = MASTER.match(s, pos)
        if not m:
            raise SyntaxError(f"Unexpected text at pos {pos}: {s[pos:pos+20]!r}")
        kind = m.lastgroup
        if kind != 'SKIP':
            tokens.append((kind, m.group(kind)))
        pos = m.end()
    tokens.append(('EOF',''))
    return tokens

# ============================================================
# PARSER (Recursive Descent)
# ============================================================

class Parser:
    def __init__(self, tokens): self.tokens, self.i = tokens, 0
    def peek(self): return self.tokens[self.i]
    def next(self): tok=self.tokens[self.i]; self.i+=1; return tok
    def match(self, kind):
        if self.peek()[0] == kind: return self.next()
        return None

    def parse(self):
```

```
            result = self.parse_iff()
            if self.peek()[0] != 'EOF': raise SyntaxError("Extra input")
            return result

        # Precedence:
        # IFF > IMPLIES > OR > AND > NOT > ATOM

        def parse_iff(self):
            left = self.parse_implies()
            while self.match('IFF'):
                right = self.parse_implies()
                left = Iff(left, right)
            return left

        def parse_implies(self):
            left = self.parse_or()
            while self.match('IMPLIES'):
                right = self.parse_or()
                left = Implies(left, right)
            return left

        def parse_or(self):
            left = self.parse_and()
            while self.match('OR'):
                right = self.parse_and()
                left = Or(left, right)
            return left

        def parse_and(self):
            left = self.parse_not()
            while self.match('AND'):
                right = self.parse_not()
                left = And(left, right)
            return left

        def parse_not(self):
            if self.match('NOT'):
                return Not(self.parse_not())
            return self.parse_atom()

        def parse_atom(self):
            tok = self.peek()

            if tok[0] == 'LPAREN':
                self.next()
                inside = self.parse_iff()
                if not self.match('RPAREN'): raise SyntaxError("Missing )")
                return inside

            if tok[0] in ('FORALL','EXISTS'):
                qtok = self.next()
                vtok = self.next()
                if vtok[0] != 'IDENT': raise SyntaxError("Expected variable")
                var = Var(vtok[1])
                sub = self.parse_atom()
                return Forall(var, sub) if qtok[0]=='FORALL' else Exists(var, sub)

            if tok[0] == 'IDENT':
                name = self.next()[1]
                if self.match('LPAREN'):
                    args = []
                    if self.peek()[0] != 'RPAREN':
                        args.append(self.parse_iff())
                        while self.match('COMMA'):
                            args.append(self.parse_iff())
                    if not self.match('RPAREN'): raise SyntaxError("Expected ) in args")
                    return Pred(name, args)
                else:
                    return Pred(name, [])

            raise SyntaxError(f"Unexpected token: {tok}")

    # ============================================================
    # TRANSFORMATION FUNCTIONS (Algorithm)
    # ============================================================

    def eliminate_implications(f):
        if isinstance(f, Implies):
            return Or(Not(eliminate_implications(f.left)), eliminate_implications(f.right))
        if isinstance(f, Iff):
            A = eliminate_implications(f.left)
            B = eliminate_implications(f.right)
```

```
            return And(Or(Not(A), B), Or(Not(B), A))
        if isinstance(f, Not): return Not(eliminate_implications(f.sub))
        if isinstance(f, And): return And(eliminate_implications(f.left), eliminate_implications(f.right))
        if isinstance(f, Or): return Or(eliminate_implications(f.left), eliminate_implications(f.right))
        if isinstance(f, Forall): return Forall(f.var, eliminate_implications(f.sub))
        if isinstance(f, Exists): return Exists(f.var, eliminate_implications(f.sub))
        return f

    def move_negation_inward(f):
        if isinstance(f, Not):
            g = f.sub
            if isinstance(g, Not): return move_negation_inward(g.sub)
            if isinstance(g, And): return Or(move_negation_inward(Not(g.left)), move_negation_inward(Not(g.right)))
            if isinstance(g, Or): return And(move_negation_inward(Not(g.left)), move_negation_inward(Not(g.right)))
            if isinstance(g, Forall): return Exists(g.var, move_negation_inward(Not(g.sub)))
            if isinstance(g, Exists): return Forall(g.var, move_negation_inward(Not(g.sub)))
            return f
        if isinstance(f, And): return And(move_negation_inward(f.left), move_negation_inward(f.right))
        if isinstance(f, Or): return Or(move_negation_inward(f.left), move_negation_inward(f.right))
        if isinstance(f, Forall): return Forall(f.var, move_negation_inward(f.sub))
        if isinstance(f, Exists): return Exists(f.var, move_negation_inward(f.sub))
        return f

    _var_counter = itertools.count()

    def standardize(f, env=None):
        if env is None: env = {}

        if isinstance(f, Var):
            return Var(env.get(f.name, f.name))

        if isinstance(f, Forall) or isinstance(f, Exists):
            newname = f"v{next(_var_counter)}"
            newenv = env.copy()
            newenv[f.var.name] = newname
            body = standardize(f.sub, newenv)
            return Forall(Var(newname), body) if isinstance(f, Forall) else Exists(Var(newname), body)

        if isinstance(f, Not): return Not(standardize(f.sub, env))
        if isinstance(f, And): return And(standardize(f.left, env), standardize(f.right, env))
        if isinstance(f, Or): return Or(standardize(f.left, env), standardize(f.right, env))
        if isinstance(f, Pred): return Pred(f.name, [standardize(a, env) for a in f.args])
        return f

    _sk_counter = itertools.count()

    def substitute(f, varname, value):
        if isinstance(f, Var): return value if f.name == varname else f
        if isinstance(f, Pred): return Pred(f.name, [substitute(a, varname, value) for a in f.args])
        if isinstance(f, Func): return Func(f.name, [substitute(a, varname, value) for a in f.args])
        if isinstance(f, Not): return Not(substitute(f.sub, varname, value))
        if isinstance(f, And): return And(substitute(f.left, varname, value), substitute(f.right, varname, value))
        if isinstance(f, Or): return Or(substitute(f.left, varname, value), substitute(f.right, varname, value))
        if isinstance(f, Forall): return Forall(f.var, substitute(f.sub, varname, value))
        if isinstance(f, Exists): return Exists(f.var, substitute(f.sub, varname, value))
        return f

    def skolemize(f, env=None):
        if env is None: env = []

        if isinstance(f, Forall):
            return Forall(f.var, skolemize(f.sub, env + [f.var.name]))

        if isinstance(f, Exists):
            sk = f"Sk{next(_sk_counter)}"
            repl = Func(sk, [Var(v) for v in env]) if env else Const(sk)
            return skolemize(substitute(f.sub, f.var.name, repl), env)

        if isinstance(f, And): return And(skolemize(f.left, env), skolemize(f.right, env))
        if isinstance(f, Or): return Or(skolemize(f.left, env), skolemize(f.right, env))
        if isinstance(f, Not): return Not(skolemize(f.sub, env))
        if isinstance(f, Pred): return Pred(f.name, [skolemize(a, env) for a in f.args])
        return f

    def drop_universal(f):
        if isinstance(f, Forall): return drop_universal(f.sub)
        if isinstance(f, And): return And(drop_universal(f.left), drop_universal(f.right))
        if isinstance(f, Or): return Or(drop_universal(f.left), drop_universal(f.right))
        if isinstance(f, Not): return Not(drop_universal(f.sub))
        return f

    def distribute(f):
```

```
    if isinstance(f, And): return And(distribute(f.left), distribute(f.right))
    if isinstance(f, Or):
        A, B = distribute(f.left), distribute(f.right)
        if isinstance(A, And):
            return And(distribute(Or(A.left, B)), distribute(Or(A.right, B)))
        if isinstance(B, And):
            return And(distribute(Or(A, B.left)), distribute(Or(A, B.right)))
        return Or(A, B)
    return f

# ============================================================
# CLAUSE EXTRACTION
# ============================================================

def literal_of(x):
    return f"¬{x.sub}" if isinstance(x, Not) else str(x)

def extract_clauses(cnf):
    if isinstance(cnf, And):
        return extract_clauses(cnf.left) + extract_clauses(cnf.right)
    else:
        return [collect_literals(cnf)]

def collect_literals(expr):
    if isinstance(expr, Or):
        return collect_literals(expr.left) | collect_literals(expr.right)
    return {literal_of(expr)}

# ============================================================
# MAIN FUNCTION for COLAB
# ============================================================

def convert_to_cnf(formula_string):
    print("======= INPUT FORMULA =======")
    print(formula_string)

    tokens = tokenize(formula_string)
    parsed = Parser(tokens).parse()
    print("\nAST:", parsed)

    s1 = eliminate_implications(parsed)
    print("\n1) After eliminating implications:\n", s1)

    s2 = move_negation_inward(s1)
    print("\n2) After moving negation inward:\n", s2)

    s3 = standardize(s2)
    print("\n3) After standardizing variables:\n", s3)

    s4 = skolemize(s3)
    print("\n4) After Skolemization:\n", s4)

    s5 = drop_universal(s4)
    print("\n5) After dropping universal quantifiers:\n", s5)

    s6 = distribute(s5)
    print("\n6) After distributing OR over AND (CNF):\n", s6)

    clauses = extract_clauses(s6)
    print("\n======= CNF CLAUSES =======")
    for i, c in enumerate(clauses, 1):
        print(f"Clause {i}:", c)

    return clauses
```

```
convert_to_cnf("forall x (P(x) => exists y (Q(y) & R(x,y)))")
```

```
======= INPUT FORMULA =======
forall x (P(x) => exists y (Q(y) & R(x,y)))

AST: ∀x.(P(x()) → ∃y.(Q(y()) ∧ R(x(), y())))

1) After eliminating implications:
 ∀x.(¬P(x()) ∨ ∃y.(Q(y()) ∧ R(x(), y())))

2) After moving negation inward:
 ∀x.(¬P(x()) ∨ ∃y.(Q(y()) ∧ R(x(), y())))

3) After standardizing variables:
 ∀v0.(¬P(x()) ∨ ∃v1.(Q(y()) ∧ R(x(), y())))
```

```
4) After Skolemization:
 ∀v0.(¬P(x()) ∨ (Q(y()) ∧ R(x(), y())))

5) After dropping universal quantifiers:
 (¬P(x()) ∨ (Q(y()) ∧ R(x(), y())))

6) After distributing OR over AND (CNF):
 ((¬P(x()) ∨ Q(y())) ∧ (¬P(x()) ∨ R(x(), y())))

======= CNF CLAUSES =======
Clause 1: {'Q(y())', '¬P(x())'}
Clause 2: {'R(x(), y())', '¬P(x())'}
[{'Q(y())', '¬P(x())'}, {'R(x(), y())', '¬P(x())'}]
```

```
formula = "forall x ( (forall y (Animal(y) => Loves(x,y))) => exists y (Loves(y,x)) )"
convert_to_cnf(formula)
```

```
======= INPUT FORMULA =======
forall x ( (forall y (Animal(y) => Loves(x,y))) => exists y (Loves(y,x)) )

AST: ∀x.(∀y.(Animal(y()) → Loves(x(), y())) → ∃y.Loves(y(), x()))

1) After eliminating implications:
 ∀x.(¬∀y.(¬Animal(y()) ∨ Loves(x(), y())) ∨ ∃y.Loves(y(), x()))

2) After moving negation inward:
 ∀x.(∃y.(Animal(y()) ∧ ¬Loves(x(), y())) ∨ ∃y.Loves(y(), x()))

3) After standardizing variables:
 ∀v2.(∃v3.(Animal(y()) ∧ ¬Loves(x(), y())) ∨ ∃v4.Loves(y(), x()))

4) After Skolemization:
 ∀v2.((Animal(y()) ∧ ¬Loves(x(), y())) ∨ Loves(y(), x()))

5) After dropping universal quantifiers:
 ((Animal(y()) ∧ ¬Loves(x(), y())) ∨ Loves(y(), x()))

6) After distributing OR over AND (CNF):
 ((Animal(y()) ∨ Loves(y(), x())) ∧ (¬Loves(x(), y()) ∨ Loves(y(), x())))

======= CNF CLAUSES =======
Clause 1: {'Animal(y())', 'Loves(y(), x())'}
Clause 2: {'¬Loves(x(), y())', 'Loves(y(), x())'}
[{'Animal(y())', 'Loves(y(), x())'}, {'Loves(y(), x())', '¬Loves(x(), y())'}]
```

```
# Fixed FOL Resolution with Unification — Colab / Notebook friendly
# This is the previous resolver with a corrected parser for literals/terms.
import re
import itertools
from copy import deepcopy

# ------------------------
# Term / Literal data types
# ------------------------
class Var:
    def __init__(self, name): self.name = name
    def __repr__(self): return self.name
    def __eq__(self, other): return isinstance(other, Var) and self.name==other.name
    def __hash__(self): return hash(('Var',self.name))

class Const:
    def __init__(self, name): self.name = name
    def __repr__(self): return self.name
    def __eq__(self, other): return isinstance(other, Const) and self.name==other.name
    def __hash__(self): return hash(('Const',self.name))

class Func:
    def __init__(self, name, args): self.name = name; self.args = args
    def __repr__(self): return f"{self.name}({', '.join(map(str,self.args))})"
    def __eq__(self, other): return isinstance(other, Func) and self.name==other.name and self.args==other.args
    def __hash__(self): return hash(('Func', self.name, tuple(self.args)))

class Literal:
    def __init__(self, pred, args, neg=False):
        self.pred = pred
        self.args = args
        self.neg = neg
    def __repr__(self):
        return ("¬" if self.neg else "") + (f"{self.pred}({', '.join(map(str,self.args))})" if self.args else f"
    def __eq__(self, other):
        return isinstance(other, Literal) and self.pred==other.pred and self.neg==other.neg and self.args==other
    def __hash__(self):
        return hash(('Lit', self.pred, self.neg, tuple(self.args)))

# ------------------------
```

```
# Tokenizer for clause/literal input
# -------------------------
# tokens: ~ ¬ - | v , ( ) identifiers
TOKEN_RE = re.compile(r'\s*(~|¬|-|\||v|,|\(|\)|[A-Za-z_][A-Za-z0-9_]*)\s*')

def tokenize_clause(s):
    toks = TOKEN_RE.findall(s)
    # filter out empty tokens
    return [t for t in toks if t and not t.isspace()]

# -------------------------
# Parse term: Var / Const / Func
# token list based, returns (term, next_index)
# -------------------------
def parse_term(tokens, i=0):
    if i >= len(tokens):
        raise SyntaxError("Unexpected end while parsing term")
    tok = tokens[i]
    if re.fullmatch(r'[A-Za-z_][A-Za-z0-9_]*', tok):
        # identifier maybe function if next token is '('
        if i+1 < len(tokens) and tokens[i+1] == '(':
            name = tok
            i += 2  # now at first token inside '(' or ')'
            args = []
            if i < len(tokens) and tokens[i] == ')':
                # no-arg function (treat as Func with no args)
                i += 1
                return Func(name, []), i
            while True:
                term, i = parse_term(tokens, i)
                args.append(term)
                if i < len(tokens) and tokens[i] == ',':
                    i += 1
                    continue
                if i < len(tokens) and tokens[i] == ')':
                    i += 1
                    break
                raise SyntaxError("Expected ',' or ')' in function arguments")
            return Func(name, args), i
        else:
            # identifier alone: variable if lowercase-start, else constant
            if tok[0].islower():
                return Var(tok), i+1
            else:
                return Const(tok), i+1
    else:
        raise SyntaxError(f"Invalid token in term: {tok}")

# -------------------------
# Parse literal strings like "~P(x,y)" or "Q" etc.
# Returns Literal object
# -------------------------
def parse_literal(s):
    tokens = tokenize_clause(s)
    if not tokens:
        raise SyntaxError("Empty literal")
    i = 0
    neg = False
    if tokens[i] in ('~','¬','-'):
        neg = True
        i += 1
        if i >= len(tokens):
            raise SyntaxError("Negation without atom")
    if i >= len(tokens):
        raise SyntaxError("Missing predicate after optional negation")
    # next token must be an identifier predicate name
    if not re.fullmatch(r'[A-Za-z_][A-Za-z0-9_]*', tokens[i]):
        raise SyntaxError(f"Invalid predicate name: {tokens[i]}")
    pname = tokens[i]; i += 1
    args = []
    if i < len(tokens) and tokens[i] == '(':
        i += 1
        if i < len(tokens) and tokens[i] == ')':
            i += 1
            return Literal(pname, [], neg)
        while True:
            term, i = parse_term(tokens, i)
            args.append(term)
            if i < len(tokens) and tokens[i] == ',':
                i += 1
                continue
            if i < len(tokens) and tokens[i] == ')':
```

```
                i += 1
                break
            raise SyntaxError("Missing ) in literal")
    # ensure no leftover tokens
    if i != len(tokens):
        raise SyntaxError(f"Extra tokens in literal: {tokens[i:]}")
    return Literal(pname, args, neg)

# -------------------------
# Parse a clause string: disjunction of literals separated by '|' or 'v'
# -------------------------
def parse_clause(s):
    parts = []
    buf = ""
    depth = 0
    for ch in s:
        if ch == '(':
            depth += 1
        elif ch == ')':
            depth -= 1
        if (ch == '|' or ch == 'v') and depth == 0:
            parts.append(buf.strip()); buf = ""
        else:
            buf += ch
    if buf.strip():
        parts.append(buf.strip())
    literals = [parse_literal(p) for p in parts]
    return set(literals)

# -------------------------
# Substitution / apply functions (same as before)
# -------------------------
def apply_subst_term(term, subst):
    if isinstance(term, Var):
        return subst.get(term.name, term)
    if isinstance(term, Func):
        return Func(term.name, [apply_subst_term(a, subst) for a in term.args])
    return term

def apply_subst_lit(lit, subst):
    return Literal(lit.pred, [apply_subst_term(a, subst) for a in lit.args], neg=lit.neg)

# -------------------------
# Unification (Robinson)
# -------------------------
def occurs_check(varname, term, subst):
    t = apply_subst_term(term, subst)
    if isinstance(t, Var):
        return t.name == varname
    if isinstance(t, Func):
        return any(occurs_check(varname, a, subst) for a in t.args)
    return False

def unify_terms(t1, t2, subst):
    t1 = apply_subst_term(t1, subst)
    t2 = apply_subst_term(t2, subst)
    if isinstance(t1, Var):
        if isinstance(t2, Var) and t1.name == t2.name:
            return True
        if occurs_check(t1.name, t2, subst):
            return False
        subst[t1.name] = t2
        return True
    if isinstance(t2, Var):
        return unify_terms(t2, t1, subst)
    if isinstance(t1, Const) and isinstance(t2, Const):
        return t1.name == t2.name
    if isinstance(t1, Func) and isinstance(t2, Func):
        if t1.name != t2.name or len(t1.args) != len(t2.args):
            return False
        for a,b in zip(t1.args, t2.args):
            if not unify_terms(a, b, subst):
                return False
        return True
    return False

def unify_literals(l1, l2):
    if l1.pred != l2.pred or len(l1.args) != len(l2.args):
        return None
    subst = {}
    for a,b in zip(l1.args, l2.args):
        if not unify_terms(a, b, subst):
```

```
            return None
    return subst

# ------------------------
# Standardize variables apart (rename variables in a clause)
# ------------------------
_std_counter = itertools.count()
def standardize_clause(clause):
    mapping = {}
    def std_term(t):
        if isinstance(t, Var):
            if t.name not in mapping:
                mapping[t.name] = f"v{next(_std_counter)}"
            return Var(mapping[t.name])
        if isinstance(t, Func):
            return Func(t.name, [std_term(a) for a in t.args])
        return t
    new_clause = set()
    for lit in clause:
        new_args = [std_term(a) for a in lit.args]
        new_clause.add(Literal(lit.pred, new_args, neg=lit.neg))
    return new_clause

# ------------------------
# Resolution loop (same logic as earlier)
# ------------------------
def resolution(kb_clauses, query_clause, max_iters=10000, verbose=True):
    clause_db = []
    for c in kb_clauses:
        sc = standardize_clause(c)
        clause_db.append({'clause': sc, 'parents': None, 'info': 'KB'})
    negated_query_clauses = []
    for lit in query_clause:
        neg_lit = Literal(lit.pred, lit.args, neg=not lit.neg)
        negated_query_clauses.append({neg_lit})
    for nq in negated_query_clauses:
        clause_db.append({'clause': standardize_clause(nq), 'parents': None, 'info': 'Negated query (added)'})
    if verbose:
        print("Initial clauses (numbered):")
        for idx, entry in enumerate(clause_db,1):
            print(f" C{idx} : {sorted(map(str, entry['clause']))}    -- {entry['info']}")
    existing = [frozenset(c['clause']) for c in clause_db]
    next_id = len(clause_db)+1
    iteration = 0
    new_generated = True
    while iteration < max_iters and new_generated:
        iteration += 1
        new_generated = False
        n = len(clause_db)
        for i in range(n):
            for j in range(i+1, n):
                Ci = clause_db[i]['clause']
                Cj = clause_db[j]['clause']
                for li in list(Ci):
                    for lj in list(Cj):
                        if li.pred == lj.pred and li.neg != lj.neg and len(li.args) == len(lj.args):
                            subst = unify_literals(li, lj)
                            if subst is None:
                                continue
                            new_clause = set()
                            for l in Ci:
                                if l == li: continue
                                new_clause.add(apply_subst_lit(l, subst))
                            for l in Cj:
                                if l == lj: continue
                                new_clause.add(apply_subst_lit(l, subst))
                            # tautology check
                            taut = False
                            for a in new_clause:
                                for b in new_clause:
                                    if a.pred == b.pred and a.neg != b.neg and len(a.args) == len(b.args):
                                        if all(str(x)==str(y) for x,y in zip(a.args, b.args)):
                                            taut = True; break
                                if taut: break
                            if taut:
                                continue
                            new_clause_std = standardize_clause(new_clause)
                            frozen = frozenset(new_clause_std)
                            if frozen in existing:
                                continue
                            info = f"resolvent of C{ i+1 } and C{ j+1 } on {li} / {lj} with subst {subst}"
                            clause_db.append({'clause': new_clause_std, 'parents': (i+1, j+1), 'info': info})
```

```
                            existing.append(frozen)
                            new_generated = True
                            if verbose:
                                print(f" Derived C{next_id} : {sorted(map(str,new_clause_std))}    -- {info}")
                            if len(new_clause_std) == 0:
                                if verbose:
                                    print("\nEMPTY CLAUSE derived: Query proved by refutation.")
                                return True, clause_db
                            next_id += 1
    if verbose:
        print("\nNo new clauses can be generated (or reached iteration limit). Query NOT proved.")
    return False, clause_db

# --------------------------
# Wrapper to accept user input strings
# --------------------------
def resolution_prove(kb_string, query_string, verbose=True):
    kb_parts = [part.strip() for part in kb_string.split(',') if part.strip()]
    kb_clauses = []
    for p in kb_parts:
        try:
            kb_clauses.append(parse_clause(p))
        except Exception as e:
            raise SyntaxError(f"Error parsing KB clause '{p}': {e}")
    try:
        query_clause = parse_clause(query_string)
    except Exception as e:
        raise SyntaxError(f"Error parsing query '{query_string}': {e}")
    proved, db = resolution(kb_clauses, query_clause, verbose=verbose)
    if proved:
        print("\n--- PROOF TRACE (final) ---")
        for idx, entry in enumerate(db,1):
            clause_str = sorted(map(str, entry['clause']))
            print(f"C{idx}: {clause_str}   Parents: {entry['parents']}   {entry['info']}")
    else:
        print("\n--- FINAL CLAUSES (no refutation) ---")
        for idx, entry in enumerate(db,1):
            clause_str = sorted(map(str, entry['clause']))
            print(f"C{idx}: {clause_str}   Parents: {entry['parents']}   {entry['info']}")
    return proved, db
```

```
kb = "Humidity(a) | Cloudy(a), ~Cloudy(a) | Rain(a), ~Humidity(a) | Hot(a), ~Hot(a)"
query = "Rain(a)"
proven, db = resolution_prove(kb, query)
```

```
Initial clauses (numbered):
 C1 : ['Cloudy(v0)', 'Humidity(v0)']    -- KB
 C2 : ['Rain(v1)', '¬Cloudy(v1)']    -- KB
 C3 : ['Hot(v2)', '¬Humidity(v2)']    -- KB
 C4 : ['¬Hot(v3)']    -- KB
 C5 : ['¬Rain(v4)']    -- Negated query (added)
 Derived C6 : ['Humidity(v5)', 'Rain(v5)']    -- resolvent of C1 and C2 on Cloudy(v0) / ¬Cloudy(v1) with subst
 Derived C7 : ['Cloudy(v6)', 'Hot(v6)']    -- resolvent of C1 and C3 on Humidity(v0) / ¬Humidity(v2) with subst
 Derived C8 : ['¬Cloudy(v7)']    -- resolvent of C2 and C5 on Rain(v1) / ¬Rain(v4) with subst {'v1': v4}
 Derived C9 : ['¬Humidity(v8)']    -- resolvent of C3 and C4 on Hot(v2) / ¬Hot(v3) with subst {'v2': v3}
 Derived C10 : ['Humidity(v9)', 'Rain(v9)']    -- resolvent of C1 and C2 on Cloudy(v0) / ¬Cloudy(v1) with subst
 Derived C11 : ['Cloudy(v10)', 'Hot(v10)']    -- resolvent of C1 and C3 on Humidity(v0) / ¬Humidity(v2) with sul
 Derived C12 : ['Humidity(v11)']    -- resolvent of C1 and C8 on Cloudy(v0) / ¬Cloudy(v7) with subst {'v0': v7}
 Derived C13 : ['Cloudy(v12)']    -- resolvent of C1 and C9 on Humidity(v0) / ¬Humidity(v8) with subst {'v0': v8
 Derived C14 : ['¬Cloudy(v13)']    -- resolvent of C2 and C5 on Rain(v1) / ¬Rain(v4) with subst {'v1': v4}
 Derived C15 : ['Hot(v14)', 'Rain(v14)']    -- resolvent of C2 and C7 on ¬Cloudy(v1) / Cloudy(v6) with subst {'v
 Derived C16 : ['¬Humidity(v15)']    -- resolvent of C3 and C4 on Hot(v2) / ¬Hot(v3) with subst {'v2': v3}
 Derived C17 : ['Hot(v16)', 'Rain(v16)']    -- resolvent of C3 and C6 on ¬Humidity(v2) / Humidity(v5) with subs
 Derived C18 : ['Cloudy(v17)']    -- resolvent of C4 and C7 on ¬Hot(v3) / Hot(v6) with subst {'v3': v6}
 Derived C19 : ['Humidity(v18)']    -- resolvent of C5 and C6 on ¬Rain(v4) / Rain(v5) with subst {'v4': v5}
 Derived C20 : ['Rain(v19)']    -- resolvent of C6 and C9 on Humidity(v5) / ¬Humidity(v8) with subst {'v5': v8}
 Derived C21 : ['Hot(v20)']    -- resolvent of C7 and C8 on Cloudy(v6) / ¬Cloudy(v7) with subst {'v6': v7}
 Derived C22 : ['Humidity(v21)', 'Rain(v21)']    -- resolvent of C1 and C2 on Cloudy(v0) / ¬Cloudy(v1) with subs
 Derived C23 : ['Cloudy(v22)', 'Hot(v22)']    -- resolvent of C1 and C3 on Humidity(v0) / ¬Humidity(v2) with sul
 Derived C24 : ['Humidity(v23)']    -- resolvent of C1 and C8 on Cloudy(v0) / ¬Cloudy(v7) with subst {'v0': v7}
 Derived C25 : ['Cloudy(v24)']    -- resolvent of C1 and C9 on Humidity(v0) / ¬Humidity(v8) with subst {'v0': v8
 Derived C26 : ['Humidity(v25)']    -- resolvent of C1 and C14 on Cloudy(v0) / ¬Cloudy(v13) with subst {'v0': v1
 Derived C27 : ['Cloudy(v26)']    -- resolvent of C1 and C16 on Humidity(v0) / ¬Humidity(v15) with subst {'v0':
 Derived C28 : ['¬Cloudy(v27)']    -- resolvent of C2 and C5 on Rain(v1) / ¬Rain(v4) with subst {'v1': v4}
 Derived C29 : ['Hot(v28)', 'Rain(v28)']    -- resolvent of C2 and C7 on ¬Cloudy(v1) / Cloudy(v6) with subst {'v
 Derived C30 : ['Hot(v29)', 'Rain(v29)']    -- resolvent of C2 and C11 on ¬Cloudy(v1) / Cloudy(v10) with subst
 Derived C31 : ['Rain(v30)']    -- resolvent of C2 and C13 on ¬Cloudy(v1) / Cloudy(v12) with subst {'v1': v12}
 Derived C32 : ['Rain(v31)']    -- resolvent of C2 and C18 on ¬Cloudy(v1) / Cloudy(v17) with subst {'v1': v17}
 Derived C33 : ['¬Humidity(v32)']    -- resolvent of C3 and C4 on Hot(v2) / ¬Hot(v3) with subst {'v2': v3}
 Derived C34 : ['Hot(v33)', 'Rain(v33)']    -- resolvent of C3 and C6 on ¬Humidity(v2) / Humidity(v5) with subs
 Derived C35 : ['Hot(v34)', 'Rain(v34)']    -- resolvent of C3 and C10 on ¬Humidity(v2) / Humidity(v9) with subs
 Derived C36 : ['Hot(v35)']    -- resolvent of C3 and C12 on ¬Humidity(v2) / Humidity(v11) with subst {'v2': v1
 Derived C37 : ['Hot(v36)']    -- resolvent of C3 and C19 on ¬Humidity(v2) / Humidity(v18) with subst {'v2': v1
```

```
 Derived C38 : ['Cloudy(v37)']    -- resolvent of C4 and C7 on ¬Hot(v3) / Hot(v6) with subst {'v3': v6}
 Derived C39 : ['Cloudy(v38)']    -- resolvent of C4 and C11 on ¬Hot(v3) / Hot(v10) with subst {'v3': v10}
 Derived C40 : ['Rain(v39)']    -- resolvent of C4 and C15 on ¬Hot(v3) / Hot(v14) with subst {'v3': v14}
 Derived C41 : ['Rain(v40)']    -- resolvent of C4 and C17 on ¬Hot(v3) / Hot(v16) with subst {'v3': v16}
 Derived C42 : []    -- resolvent of C4 and C21 on ¬Hot(v3) / Hot(v20) with subst {'v3': v20}

EMPTY CLAUSE derived: Query proved by refutation.

--- PROOF TRACE (final) ---
C1: ['Cloudy(v0)', 'Humidity(v0)']   Parents: None   KB
C2: ['Rain(v1)', '¬Cloudy(v1)']   Parents: None   KB
C3: ['Hot(v2)', '¬Humidity(v2)']   Parents: None   KB
C4: ['¬Hot(v3)']   Parents: None   KB
C5: ['¬Rain(v4)']   Parents: None   Negated query (added)
C6: ['Humidity(v5)', 'Rain(v5)']   Parents: (1, 2)   resolvent of C1 and C2 on Cloudy(v0) / ¬Cloudy(v1) with sul
C7: ['Cloudy(v6)', 'Hot(v6)']   Parents: (1, 3)   resolvent of C1 and C3 on Humidity(v0) / ¬Humidity(v2) with s
C8: ['¬Cloudy(v7)']   Parents: (2, 5)   resolvent of C2 and C5 on Rain(v1) / ¬Rain(v4) with subst {'v1': v4}
C9: ['¬Humidity(v8)']   Parents: (3, 4)   resolvent of C3 and C4 on Hot(v2) / ¬Hot(v3) with subst {'v2': v3}
C10: ['Humidity(v9)', 'Rain(v9)']   Parents: (1, 2)   resolvent of C1 and C2 on Cloudy(v0) / ¬Cloudy(v1) with s
```

```
#TIC TAC TOE


# tic_tac_toe_ai.py
# Human vs AI Tic-Tac-Toe (Minimax + Alpha-Beta)

from typing import List, Optional, Tuple
import math

EMPTY = ' '
HUMAN = 'X'
AI = 'O'

def init_board() -> List[List[str]]:
    return [[EMPTY]*3 for _ in range(3)]

def display(board: List[List[str]]) -> None:
    print()
    for r in range(3):
        print(' ' + ' | '.join(board[r]))
        if r < 2:
            print("---+---+---")
    print()

def is_board_full(board: List[List[str]]) -> bool:
    return all(board[r][c] != EMPTY for r in range(3) for c in range(3))

def check_winner(board: List[List[str]]) -> Optional[str]:
    # rows & cols
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != EMPTY:
            return board[i][0]
        if board[0][i] == board[1][i] == board[2][i] != EMPTY:
            return board[0][i]
    # diagonals
    if board[0][0] == board[1][1] == board[2][2] != EMPTY:
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] != EMPTY:
        return board[0][2]
    return None

def available_moves(board: List[List[str]]) -> List[Tuple[int,int]]:
    return [(r,c) for r in range(3) for c in range(3) if board[r][c] == EMPTY]

def minimax(board: List[List[str]], depth: int, maximizing: bool,
            alpha: int, beta: int) -> Tuple[int, Optional[Tuple[int,int]]]:
    """
    Returns (score, move)
    score: +1 if AI wins, -1 if human wins, 0 draw (from perspective where AI wants maximize)
    """
    winner = check_winner(board)
    if winner == AI:
        return 1, None
    if winner == HUMAN:
        return -1, None
    if is_board_full(board):
        return 0, None

    best_move = None
    if maximizing:
        max_eval = -math.inf
        for (r,c) in available_moves(board):
            board[r][c] = AI
            eval_score,   = minimax(board, depth+1, False, alpha, beta)
```

```
                board[r][c] = EMPTY
                if eval_score > max_eval:
                    max_eval = eval_score
                    best_move = (r,c)
                alpha = max(alpha, eval_score)
                if beta <= alpha:
                    break
            return int(max_eval), best_move
        else:
            min_eval = math.inf
            for (r,c) in available_moves(board):
                board[r][c] = HUMAN
                eval_score, _ = minimax(board, depth+1, True, alpha, beta)
                board[r][c] = EMPTY
                if eval_score < min_eval:
                    min_eval = eval_score
                    best_move = (r,c)
                beta = min(beta, eval_score)
                if beta <= alpha:
                    break
            return int(min_eval), best_move

    def ai_move(board: List[List[str]]) -> Tuple[int,int]:
        # If board is empty, pick center or a corner for variety
        if all(board[r][c] == EMPTY for r in range(3) for c in range(3)):
            return (1,1)  # center
        _, move = minimax(board, 0, True, -math.inf, math.inf)
        assert move is not None
        return move

    def human_move(board: List[List[str]]) -> Tuple[int,int]:
        while True:
            try:
                s = input("Enter your move as 'row col' (0 1 2): ").strip()
                if not s:
                    continue
                parts = s.split()
                if len(parts) != 2:
                    print("Please enter two integers: row and column.")
                    continue
                r, c = int(parts[0]), int(parts[1])
                if not (0 <= r <= 2 and 0 <= c <= 2):
                    print("Row and column must be 0, 1 or 2.")
                    continue
                if board[r][c] != EMPTY:
                    print("Cell not empty — choose another move.")
                    continue
                return r, c
            except ValueError:
                print("Invalid input. Use integers like: 0 2")

    def play_game():
        board = init_board()
        current = HUMAN  # human starts; change to AI if you want AI to start
        print("Tic-Tac-Toe: you are 'X', AI is 'O'.")
        display(board)

        while True:
            if current == HUMAN:
                r,c = human_move(board)
                board[r][c] = HUMAN
            else:
                print("AI is thinking...")
                r,c = ai_move(board)
                board[r][c] = AI
                print(f"AI plays: {r} {c}")
            display(board)

            winner = check_winner(board)
            if winner is not None:
                if winner == HUMAN:
                    print("You win! 🎉")
                else:
                    print("AI wins. Better luck next time.")
                break
            if is_board_full(board):
                print("It's a draw!")
                break
            current = AI if current == HUMAN else HUMAN

    if __name__ == "__main__":
        while True:
            play_game()
```