# Knowledge Representation and Reasoning

University of Bucharest

Chirobocea Mihail-Bogdan

Project 1

## 1. Resolution

I will start defining my knowledge base.

**Natural language**: If a person travels to a place with someone, that someone travels to that place too.

**FOL**: $\forall X\ \forall Y\ \forall Z\ [\ (\ TravelTo(X, Z)\ \wedge\ TravelWith(Y, X)\ )\ \supset\ TravelTo(Y, Z)\ ]$

**CNF**: $\neg TravelTo(X, Z)\ \vee\ \neg TravelWith(Y, X)\ \vee\ TravelTo(Y, Z)$

**Propositional**: { [¬TravelTo(X, Z), ¬TravelWith(Y, X), TravelTo(Y, Z)] }

**Natural language**: If a person is rich, than he/she travels with his/her wife/husband

**FOL**: $\exists Y\ \forall X\ [\ Rich(X)\ \supset\ (\ TravelWith(X, Y)\ \wedge\ MarriedTo(X, Y)\ )\ ]$

**CNF**: $(\neg Rich(X)\ \vee\ TravelWith(X, skolemtravel(Y))\ )\ \wedge\ (\neg Rich(X)\ \vee\ MarriedTo(X\ ,\ skolemmarried(Y))\ )$

**Propositional**: { [¬Rich(X), TravelWith(X, skolem_travel(Y))], [¬Rich(X), MarriedTo(X, skolem_married(Y))] }

**Natural language**:  Any person that travels with his/her husband/wife is happy.

**FOL**:  $\forall X \ \forall Y \ [ ( \text{TravelWith}(X, Y) \land \text{MarriedTo}(X, Y) ) \supset \text{Happy}(X) ]$

**CNF**: $\neg\text{TravelWith}(X, Y) \lor \neg\text{MarriedTo}(X, Y) \lor \text{Happy}(X)$

**Propositional**:  { [¬TravelWith(X, Y), ¬MarriedTo(X, Y), Happy(X)] }

**Natural language**:  Emma is Robert's husband.

**FOL**: MarriedTo(emma, robert)

**CNF**: MarriedTo(emma, robert)

**Propositional**:  { [MarriedTo(emma, robert)] }

**Natural language**:  Robert is rich

**FOL**: Rich( robert)

**CNF**: Rich( robert)

**Propositional**:  { [Rich(robert)] }

**Natural language**: If a person is married to someone, that someone is married to the initial person too.

**FOL**: $\forall X \ \forall Y \ [ \text{MarriedTo}(X, Y) \supset \text{TravelTo}(Y, X) ]$

**CNF**:  $\neg\text{MarriedTo}(X, Y) \lor \text{MarriedTo}(Y, X)$

**Propositional**: { [¬MarriedTo(X, Y), MarriedTo(Y, X)] }

**Natural language**: If a person travels with someone, that someone travels with that person too.

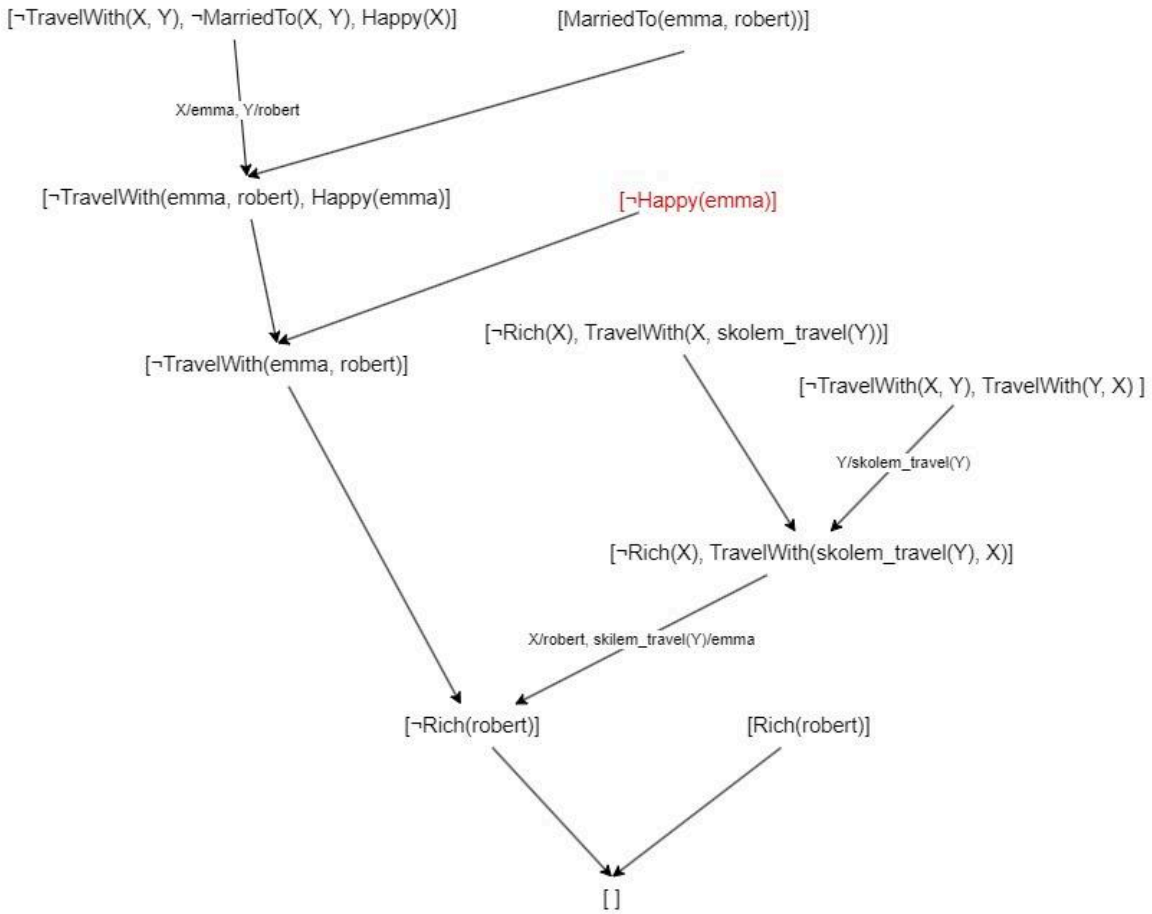
**FOL**: $\forall X \ \forall Y \ [ \text{TravelWith}(X, Y) \supset \text{TravelWith}(Y, X) ]$

**CNF**:  $\neg\text{TravelWith}(X, Y) \lor \text{TravelWith}(Y, X)$

**Propositional**: { [¬TravelWith(X, Y), TravelWith(Y, X)] }

**Manual Resolution**

I want to prove that our knowledge base entails: Emma is happy. Iwill use the principle of reduction to absurdity.




**Automatic Resolution**

As expected, with the automatic resolution we get the same result, Emma is happy.

**Automatic Resolution - Given sets of propositional clauses**

Running test: [[not(a), b], [c, d], [not(d), b], [not(c), b], [not(b)]]...
Knowledge Base 1 Status: **Unsatisfiable**

Running test: [[not(b), a], [not(a), b, e], [a, not(e)], [not(a)], [e]]...
Knowledge Base 2 Status: **Unsatisfiable**

Running test: [[not(a), b], [c, f], [not(c)], [not(f), b], [not(c), b]]...
Knowledge Base 3 Status: **Satisfiable**

Running test: [[a, b], [not(a), not(b)]]...
Knowledge Base 4 Status: **Satisfiable**

## 2. SAT Solver - David Putnam

As choosing atom strategies, I implemented:

- Most balanced
- Most frequent

However, the most frequent strategies do not finish for the given sets of propositional clauses, most likely ending up into an infinite loop.

For the most balanced I got the following results:

1. [[toddler], [not(toddler), child], [not(child), not(male), boy], [not(infant), child], [not(child), not(female), girl], [female], [girl]]
   ->
   **Solution**:
[girl=true,child=true,toddler=true,female=true,boy=true,male=true,infant=true]

2. [[toddler], [not(toddler), child], [not(child), not(male), boy], [not(infant), child], [not(child), not(female), girl], [female], [not(girl)]]
   -> **NO**

3. [[not(a), b], [c, d], [not(d), b], [not(c), b], [not(b)], [e], [a, b, not(f), f]]
   -> **NO**

4. [[not(b), a], [not(a), b, e], [e], [a, not(e)], [not(a)]]
   -> **NO**

5. [[not(a), not(e), b], [not(d), e, not(b)], [not(e), f, not(b)], [f, not(a), e], [e, f, not(b)]]
   ->
   **Solution**: [e=true,f=true,(\+e)=true,b=true,d=true,a=true]

6. [[a, b], [not(a), not(b)], [not(a), b], [a, not(b)]]
   -> **NO**

## 3. Resources

Code: Prolog-Fun/resolution.P at master · suryanarayanan/Prolog-Fun · GitHub
Code: https://www.staff.city.ac.uk/%20jacob/solver/satsolver.txt

## 4. Code - Resolution

```
kb1([
    [not(a), b],
    [c, d],
    [not(d), b],
    [not(c), b],
    [not(b)]
]).

kb2([
    [not(b), a],
    [not(a), b, e],
    [a, not(e)],
    [not(a)],
    [e]
]).

kb3([
    [not(a), b],
    [c, f],
    [not(c)],
    [not(f), b],
    [not(c), b]
]).

kb4([
    [a, b],
    [not(a), not(b)]
]).


kb_own([
    [not(travelTo(X, Z)), not(travelWith(Y, X)), travelTo(Y, Z)],
    [not(rich(X)), travelWith(X, skolem_travel(Y))],
    [not(rich(X)), marriedTo(X, skolem_married(Y))],
    [not(travelWith(X, Y)), not(marriedTo(X, Y)), happy(X)],
    [marriedTo(emma, robert)],
    [rich(robert)],
    [not(marriedTo(X, Y)), marriedTo(Y, X)],
    [not(travelWith(X, Y)), travelWith(Y, X)]
]).
```

```
find_clause_with_literal([], _, []).
find_clause_with_literal([Clause|_], Literal, Clause) :-
    member(Literal, Clause).
find_clause_with_literal([_|Rest], Literal, Clause) :-
    find_clause_with_literal(Rest, Literal, Clause).




find_clause_with_neg_literal([], _, []).
find_clause_with_neg_literal([Clause|_], Literal, Clause) :-
    member(not(Literal), Clause).
find_clause_with_neg_literal([_|Rest], Literal, Clause) :-
    find_clause_with_neg_literal(Rest, Literal, Clause).




resolve_clauses(ComplementaryLiteral, [not(ComplementaryLiteral)],
ComplementaryLiteral, []).
resolve_clauses(List1, List2, Literal, Resolved) :-
    delete(List1, Literal, Reduced1),
    delete(List2, not(Literal), Reduced2),
    union(Reduced1, Reduced2, Resolved).




remove_duplicates([], []).
remove_duplicates([Head|Tail], [Head|UniqueTail]) :-
    delete(Tail, Head, TailWithoutHead),
    remove_duplicates(TailWithoutHead, UniqueTail).




extract_literals([], []).
extract_literals([not(Literal)|Literals], [Literal|Remaining]) :-
    extract_literals(Literals, Remaining),
    !.
```

```
extract_literals([Literal|Literals], [Literal|Remaining]) :-
    extract_literals(Literals, Remaining).

extract_unique_literals(Clauses, UniqueLiterals) :-
    flatten(Clauses, FlatLiterals),
    extract_literals(FlatLiterals, LiteralList),
    remove_duplicates(LiteralList, UniqueLiterals).




resolve(KB, 'Unsatisfiable') :-
    member([], KB),
    !.

resolve(KB, Status) :-
    extract_unique_literals(KB, Literals),
    member(CurrentLiteral, Literals),
    find_clause_with_literal(KB, CurrentLiteral, PositiveClause),
    find_clause_with_neg_literal(KB, CurrentLiteral, NegativeClause),
    PositiveClause \= [],
    NegativeClause \= [],
    resolve_clauses(PositiveClause, NegativeClause, CurrentLiteral,
ResolvedClause),
    \+ member(ResolvedClause, KB),
    union(KB, [ResolvedClause], UpdatedKB),
    resolve(UpdatedKB, Status),
    !.

resolve(KB, 'Satisfiable') :-
    extract_unique_literals(KB, Literals),
    member(CurrentLiteral, Literals),
    find_clause_with_literal(KB, CurrentLiteral, PositiveClause),
    find_clause_with_neg_literal(KB, CurrentLiteral, NegativeClause),
    PositiveClause \= [],
    NegativeClause \= [],
    resolve_clauses(PositiveClause, NegativeClause, CurrentLiteral,
ResolvedClause),
    member(ResolvedClause, KB),
    !.

resolve(KB, 'Satisfiable') :-
    extract_unique_literals(KB, Literals),
```

```
    member(CurrentLiteral, Literals),
    find_clause_with_literal(KB, CurrentLiteral, PositiveClause),
    PositiveClause == [].

resolve(KB, 'Satisfiable') :-
    extract_unique_literals(KB, Literals),
    member(CurrentLiteral, Literals),
    find_clause_with_neg_literal(KB, CurrentLiteral, NegativeClause),
    NegativeClause == [].




add_negated_to_kb(KB, Literals, Result) :-
    append(KB, [NegatedLiterals], NewKB),
    resolve(NewKB, Result).




test1 :-
    kb1(KB),
    resolve(KB, Status),
    format('Knowledge Base 1 Status: ~w~n', [Status]).

test2 :-
    kb2(KB),
    resolve(KB, Status),
    format('Knowledge Base 2 Status: ~w~n', [Status]).

test3 :-
    kb3(KB),
    resolve(KB, Status),
    format('Knowledge Base 3 Status: ~w~n', [Status]).

test4 :-
    kb4(KB),
    resolve(KB, Status),
    format('Knowledge Base 4 Status: ~w~n', [Status]).

test_kb_default :-
    kb_own(KB),
    Literals=[not(happy(emma))],
```

```
    add_negated_to_kb(KB, Literals, Result),
    format('Test Default KB query: ~w~n', [Result]).

test_own_query(Literals) :-
    kb_own(KB),
    add_negated_to_kb(KB, Literals, Result),
    format('Query Result: ~w~n', [Result]).
```

## 5. Code - SAT

```
kb1([
    [toddler],
    [\+toddler, child],
    [\+child, \+male, boy],
    [\+infant, child],
    [\+child, \+female, girl],
    [female],
    [girl]
]).

kb2([
    [toddler],
    [\+toddler, child],
    [\+child, \+male, boy],
    [\+infant, child],
    [\+child, \+female, girl],
    [female],
    [\+girl]
]).

kb3([
    [\+a, b],
    [c, d],
    [\+d, b],
    [\+c, b],
    [\+b],
    [e],
    [a, b, \+f, f]
]).

kb4([
    [\+b, a],
    [\+a, b, e],
```

```
    [e],
    [a, \+e],
    [\+a]
]).

kb5([
    [\+a, \+e, b],
    [\+d, e, \+b],
    [\+e, f, \+b],
    [f, \+a, e],
    [e, f, \+b]
]).

kb6([
    [a, b],
    [\+a, \+b],
    [\+a, b],
    [a, \+b]
]).




count_occurrences(_, [], 0).

count_occurrences(Literal, [Clause | Rest], Count) :-
    (member(Literal, Clause); member(\+Literal, Clause)),
    count_occurrences(Literal, Rest, RestCount),
    Count is RestCount + 1.

count_occurrences(Literal, [Clause | Rest], Count) :-
    \+member(Literal, Clause),
    \+member(\+Literal, Clause),
    count_occurrences(Literal, Rest, Count).




statistics([], [], []).

statistics([Clause | Rest], Literals, Stats) :-
    statistics(Rest, RestLiterals, _),
```

```
    findall(Literal, (member(Literal, Clause); member(\+Literal,
Clause)), ClauseLiterals),
    union(RestLiterals, ClauseLiterals, Literals),
    findall(Literal-PosCount-NegCount, (
        member(Literal, Literals),
        count_occurrences(Literal, [Clause | Rest], PosCount),
        count_occurrences(\+Literal, [Clause | Rest], NegCount)
    ), Stats).




most_frequent(Literal, Clauses) :-
    sort(Clauses, ClausesSorted),
    statistics(ClausesSorted, _, Stats),
    max_member(_-Literal, Stats).


most_balanced(Literal, Clauses) :-
    sort(Clauses, ClausesSorted),
    statistics(ClausesSorted, _, Stats),
    maplist(balance_metric, Stats, Balances),
    min_member(_-Literal, Balances).

balance_metric(Literal-PosCount-NegCount, Balance-Literal) :-
    Balance is abs(PosCount - NegCount).







remove_literal([], _, []).

remove_literal([Clause | RestClauses], Literal, [NewClause | NewRest])
:-
    member(Literal, Clause),
    subtract(Clause, [Literal], NewClause),
    remove_literal(RestClauses, Literal, NewRest).

remove_literal([Clause | RestClauses], Literal, [Clause | NewRest]) :-
    \+member(Literal, Clause),
    remove_literal(RestClauses, Literal, NewRest).
```

```
resolve_clauses(Clauses, Literal, Result) :-
    findall(C, (member(C, Clauses), \+member(Literal, C),
member(\+Literal, C)), NegativeClauses),
    findall(C, (member(C, Clauses), \+member(Literal, C),
\+member(\+Literal, C)), PositiveClauses),
    remove_literal(NegativeClauses, \+Literal, CleanNegativeClauses),
    append(PositiveClauses, CleanNegativeClauses, Result).




davis_putnam_solver([], Assignment, _, _) :-
    writeln('YES'),
    format('Solution: ~w~n', [Assignment]),
    !.

davis_putnam_solver(Clauses, _, _, _) :-
    member([], Clauses),
    writeln('NO'),
    !.

davis_putnam_solver(Clauses, Assignment, Strategy, Conflict) :-
    (Strategy = most_frequent -> most_frequent(Literal, Clauses);
     Strategy = most_balanced -> most_balanced(Literal, Clauses)),

    (member(Literal = true, Assignment) -> \+member(\+Literal = true,
Assignment), !;
     member(\+Literal = true, Assignment) -> \+member(Literal = true,
Assignment), !;
     resolve_clauses(Clauses, Literal, ResolvedClauses),
     davis_putnam_solver(ResolvedClauses, [Literal = true |
Assignment], Strategy, Conflict)),

    (member(Literal = false, Assignment) -> \+member(\+Literal = false,
Assignment), !;
```

```
    member(\+Literal = false, Assignment) -> \+member(Literal = false,
Assignment), !;
    resolve_clauses(Clauses, \+Literal, ResolvedClauses),
    davis_putnam_solver(ResolvedClauses, [Literal = false |
Assignment], Strategy, Conflict)).




test1_mostbalanced :- kb1(Clauses), davis_putnam_solver(Clauses, [],
most_balanced, _).
test2_mostbalanced :- kb2(Clauses), davis_putnam_solver(Clauses, [],
most_balanced, _).
test3_mostbalanced :- kb3(Clauses), davis_putnam_solver(Clauses, [],
most_balanced, _).
test4_mostbalanced :- kb4(Clauses), davis_putnam_solver(Clauses, [],
most_balanced, _).
test5_mostbalanced :- kb5(Clauses), davis_putnam_solver(Clauses, [],
most_balanced, _).
test6_mostbalanced :- kb6(Clauses), davis_putnam_solver(Clauses, [],
most_balanced, _).

test1_mostfrequent :- kb1(Clauses), davis_putnam_solver(Clauses, [],
most_frequent, _).
test2_mostfrequent :- kb2(Clauses), davis_putnam_solver(Clauses, [],
most_frequent, _).
test3_mostfrequent :- kb3(Clauses), davis_putnam_solver(Clauses, [],
most_frequent, _).
test4_mostfrequent :- kb4(Clauses), davis_putnam_solver(Clauses, [],
most_frequent, _).
test5_mostfrequent :- kb5(Clauses), davis_putnam_solver(Clauses, [],
most_frequent, _).
test6_mostfrequent :- kb6(Clauses), davis_putnam_solver(Clauses, [],
most_frequent, _).
```

# Project 2

## 1. Exercise 1

I will start defining my knowledge base.

**Rules**:

**Natural Language**: If a plant receives sunlight, then it can perform photosynthesis.
**Horn Clause**: ["nS", "P"]

**Natural Language**: If a plant performs photosynthesis and it is watered regularly, then the plant will grow.
**Horn Clause**: ["nP", "nW", "G"]

**Natural Language**:If a plant grows and is protected from pests, then the plant will produce flowers.
**Horn Clause**: ["nG", "nE", "F"]

**Questions**:

Does the plant receive sunlight? (yes/no)
Is the plant watered regularly? (yes/no)
Is the plant protected from pests? (yes/no)

With this, we can check for example if the plant will produce flowers (i.e. the atom [“F”]). This is true only if all the answers were “yes”.

Bellow are some results for answers: “yes”, “yes”, ”no” (in this order):

Goal: P, nG
Backward result is NO
Forward result is NO

Goal: F
Backward result is NO
Forward result is NO

Goal: G
Backward result is YES
Forward result is YES

Goal: P

Backward result is YES
Forward result is YES

## 2. Exercise 2

I will start defining my knowledge base.

**Rules**:

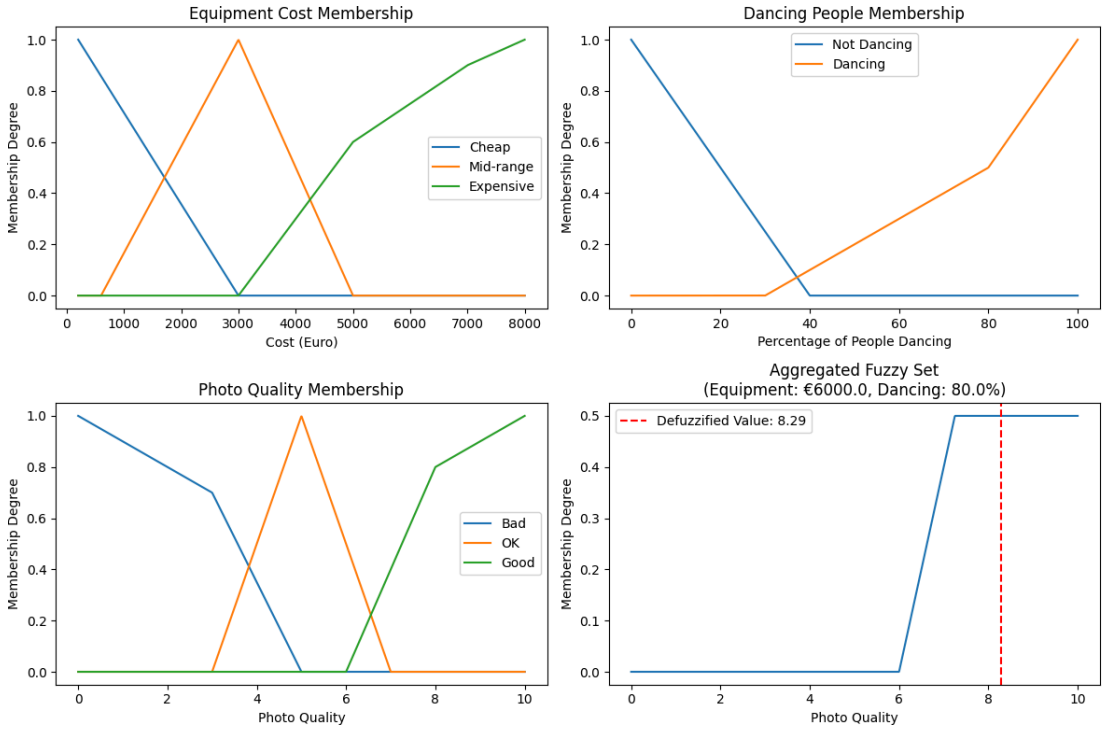
If the equipment is cheap and people are not dancing then photos will be bad.
If the equipment is mid-range or people are dancing, photos will be ok.
If the equipment is expensive and people are dancing that photos will be good

**Questions**:

Enter the cost of photography equipment (200-8000 Euro)
Enter the percentage of people dancing (0-100%)

The vague predicates are cheap, mid-range, expensive, dancing, not_dancing, bad, ok, good.

I used linear interpolation for the degree curves and the center of gravity method to pick our final prediction for the aggregate results.



## 3. Code 1

```
def convert(input_str):
    if "y" in input_str.strip().split()[0].lower():
        return True
    elif "n" in input_str.strip().split()[0].lower():
        return False
    return None


def get_input(prompt):
    while True:
        i = input(prompt)
        b = convert(i)
        if b is not None:
            return b
        print("Invalid input. Please enter 'y' or 'n'.")


def proces(clause):
    positive_literal = None
    negated_literals = []
    for literal in clause:
        if literal.startswith("n"):
            negated_literals.append(literal[1:])
        else:
            positive_literal = literal

    return positive_literal, negated_literals


def check_solved_literals(literals, solved_literals):
    return all(literal in solved_literals for literal in literals)


def backward(goals, rules):
    if not goals:
        return True

    goal = goals[0]
    rest_goals = goals[1:]
```

```
    if goal.startswith("n"):
        positive_goal = goal[1:]
        result = backward([positive_goal], rules)
        if not result:
            return backward(rest_goals, rules)
        return False

    for clause in rules:
        positive_literal, negated_literals = proces(clause)

        if goal == positive_literal:
            new_goals = negated_literals + rest_goals
            if backward(new_goals, rules):
                return True

    return False


def forward(goals, rules, solved_literals):
    while True:
        progress = False
        for clause in rules:
            positive_literal, negated_literals = proces(clause)
            if check_solved_literals(negated_literals, solved_literals)
and positive_literal not in solved_literals:
                solved_literals.append(positive_literal)
                progress = True

        if not progress:
            break


    for goal in goals:
        if goal.startswith("n"):
            if goal[1:] in solved_literals:
                return False
        else:
            if goal not in solved_literals:
                return False

    return True
```

```
"""
Rules:
If a plant receives sunlight, then it can perform photosynthesis.
If a plant performs photosynthesis and it is watered regularly, then
the plant will grow.
If a plant grows and is protected from pests, then the plant will
produce flowers.
Questions:
Does the plant receive sunlight? (yes/no)
Is the plant watered regularly? (yes/no)
Is the plant protected from pests? (yes/no)
"""


while True:

    rules = [["nS", "P"], ["nP", "nW", "G"], ["nG", "nE", "F"]]

    if get_input("Does the plant receive sunlight? [y/n]\n"):
        rules.append(["S"])
    if get_input("Is the plant watered regularly? [y/n]\n"):
        rules.append(["W"])
    if get_input("Is the plant protected from pests? [y/n]\n"):
        rules.append(["E"])

    goals = [["P", "nG"], ["F"], ["G"], ["P"], ["P", "G"]]

    print("    Results:\n")
    for goal in goals:
        print(f"Goal: {', '.join(goal)}")
        print("Backward result is", "YES" if backward(goal, rules) else
"NO")
        print("Forward result is", "YES" if forward(goal, rules, [])
else "NO")

    print("\n")
    if "stop" in input("To terminate the loop enter 'stop' else press
enter\n").strip().lower():
        break
```

## 4. Code 2

```
import numpy as np
from scipy.integrate import quad
import matplotlib.pyplot as plt
import os


def equipment_cheap(x):
    return np.interp(x, [200, 3000], [1, 0])

def equipment_mid_range(x):
    return np.interp(x, [600, 3000, 5000], [0, 1, 0])

def equipment_expensive(x):
    return np.interp(x, [3000, 5000, 7000, 8000], [0, 0.6, 0.9, 1])

def people_not_dancing(x):
    return np.interp(x, [0, 40], [1, 0])

def people_dancing(x):
    return np.interp(x, [30, 80, 100], [0, 0.5, 1])

def photos_bad(x):
    return np.interp(x, [0, 3, 5], [1, 0.7, 0])

def photos_ok(x):
    return np.interp(x, [3, 5, 7], [0, 1, 0])

def photos_good(x):
    return np.interp(x, [6, 8, 10], [0, 0.8, 1])



def rule1(equipment, dancing):
    return min(equipment_cheap(equipment), people_not_dancing(dancing))

def rule2(equipment, dancing):
    return equipment_mid_range(equipment)

def rule3(equipment, dancing):
    return min(equipment_expensive(equipment), people_dancing(dancing))
```

```
def aggregated_shape(x, equipment, dancing):
    return max(
        min(rule1(equipment, dancing), photos_bad(x)),
        min(rule2(equipment, dancing), photos_ok(x)),
        min(rule3(equipment, dancing), photos_good(x))
    )

def defuzzify(equipment, dancing):
    def x_times_shape(x):
        return x * aggregated_shape(x, equipment, dancing)

    numerator, _ = quad(x_times_shape, 0, 10, limit=1000)
    denominator, _ = quad(lambda x: aggregated_shape(x, equipment,
dancing), 0, 10, limit=1000)

    epsilon = 1e-10
    if denominator < epsilon:
        return 0
    return numerator / denominator



def plot_results(equipment, dancing, iteration):
    if not os.path.exists('plots'):
        os.makedirs('plots')

    plt.figure(figsize=(12, 8))
    x_equipment = np.linspace(200, 8000, 1000)
    x_dancing = np.linspace(0, 100, 1000)
    x_photos = np.linspace(0, 10, 1000)

    plt.subplot(2, 2, 1)
    plt.plot(x_equipment, [equipment_cheap(x) for x in x_equipment],
label='Cheap')
    plt.plot(x_equipment, [equipment_mid_range(x) for x in
x_equipment], label='Mid-range')
    plt.plot(x_equipment, [equipment_expensive(x) for x in
x_equipment], label='Expensive')
    plt.title('Equipment Cost Membership')
    plt.xlabel('Cost (Euro)')
    plt.ylabel('Membership Degree')
    plt.legend()
```

```
    plt.subplot(2, 2, 2)
    plt.plot(x_dancing, [people_not_dancing(x) for x in x_dancing],
label='Not Dancing')
    plt.plot(x_dancing, [people_dancing(x) for x in x_dancing],
label='Dancing')
    plt.title('Dancing People Membership')
    plt.xlabel('Percentage of People Dancing')
    plt.ylabel('Membership Degree')
    plt.legend()

    plt.subplot(2, 2, 3)
    plt.plot(x_photos, [photos_bad(x) for x in x_photos], label='Bad')
    plt.plot(x_photos, [photos_ok(x) for x in x_photos], label='OK')
    plt.plot(x_photos, [photos_good(x) for x in x_photos],
label='Good')
    plt.title('Photo Quality Membership')
    plt.xlabel('Photo Quality')
    plt.ylabel('Membership Degree')
    plt.legend()

    plt.subplot(2, 2, 4)
    x = np.linspace(0, 10, 1000)
    y = [aggregated_shape(xi, equipment, dancing) for xi in x]
    plt.plot(x, y)
    plt.title(f"Aggregated Fuzzy Set\n(Equipment: €{equipment},
Dancing: {dancing}%)")
    plt.xlabel("Photo Quality")
    plt.ylabel("Membership Degree")
    defuzzified_value = defuzzify(equipment, dancing)
    plt.axvline(x=defuzzified_value, color='r', linestyle='--',
label=f'Defuzzified Value: {defuzzified_value:.2f}')
    plt.legend()

    plt.tight_layout()
    plt.savefig(f'plots/fuzzy_result_{iteration}.png')
    plt.close()



def get_valid_equipment_cost():
    while True:
        try:
```

```
            cost = float(input("Enter the cost of photography equipment
(200-8000 Euro): "))
            if 200 <= cost <= 8000:
                return cost
            else:
                print("Invalid input. Please enter a value between 200
and 8000.")
        except ValueError:
            print("Invalid input. Please enter a numeric value.")


def get_valid_dancing_percentage():
    while True:
        try:
            percentage = float(input("Enter the percentage of people
dancing (0-100%): "))
            if 0 <= percentage <= 100:
                return percentage
            else:
                print("Invalid input. Please enter a value between 0
and 100.")
        except ValueError:
            print("Invalid input. Please enter a numeric value.")



def main():
    iteration = 1
    while True:
        equipment = get_valid_equipment_cost()
        dancing = get_valid_dancing_percentage()

        try:
            result = defuzzify(equipment, dancing)
            print(f"\nPredicted photo quality: {result:.2f}/10")

            plot_results(equipment, dancing, iteration)
            print(f"Plot saved as
'plots/fuzzy_result_{iteration}.png'")
        except Exception as e:
            print(f"An error occurred: {e}")
            print("Unable to calculate the result or generate the
plot.")
```

```
        iteration += 1

        if input("Enter 'q' to quit, or any other key to continue:
").lower() == 'q':
            break

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