# Assignment 2
# Knowledge Representation and Reasoning

Iordache Adrian-Răzvan

January 2021

## 1 Forward and Backward Chaining

The following rules and questions are given:

**Rules:**

- If a person is smart and healthy then he is fulfilled.
- If a person eats well, sleeps fine and has an active life then he is healthy.
- If a person goes to the gym at least 3 times a week then he has an active life.
- If a person sleeps at least 7 hours per night then he sleeps fine.
- If a person eats vegetables and drinks water he eats well.
- If a person reads books then he is smart.

**Questions:**

- Do you read books? (answer is yes/no)
- Do you eat vegetables? (answer is yes/no)
- How many ours do you sleep per night? (answer is a number)
- Do you drink water? (answer is yes/no)
- How many times you go to gym each week? (answer is a number)

### 1.1 Input Representation

- $[n(smart), n(healthy), fulfilled]$,
- $[n(eats), n(sleeps), n(active), healthy]$,
- $[n(gym), active]$,
- $[n(hours), sleeps]$,
- $[n(vegetables), n(water), eats]$,
- $[n(reads), smart]$,

And based on the answers from the previous questions we might dynamically add:

- $[reads], [gym], [hours], [vegetables], [water]$

## 1.2 Backward Chaining Output

### 1.2.1 Positive Case

```
?- main_backward.
```

```
Do you read books?
Type answer (yes/no): yes.
Do you eat vegetables?
Type answer (yes/no): yes.
How many hours do you sleep per night?
Type the number of hours (answer is a number): 9.
Do you drink water?
Type answer (yes/no): yes.
How many times you go to gym per week?
Type the number of times (answer is a number): 4.
```

**Backward Chaining Solution**

```
[[reads],[vegetables],[hours],[water],[gym],
[n(smart),n(healthy),fulfilled],
[n(eats),n(sleeps),n(active),healthy],
[n(gym),active],
[n(hours),sleeps],
[n(vegetables),n(water),eats],
[n(reads),smart]]
```

**Steps**

```
[n(smart),n(healthy),fulfilled]
[n(smart),n(healthy)]
[smart,healthy]
[n(reads),smart]
[n(reads)]
[reads,healthy]
[reads]
[]
[healthy]
[n(eats),n(sleeps),n(active),healthy]
[n(eats),n(sleeps),n(active)]
[eats,sleeps,active]
[n(vegetables),n(water),eats]
[n(vegetables),n(water)]
[vegetables,water,sleeps,active]
[vegetables]
[]
[water,sleeps,active]
[water]
[]
[sleeps,active]
[n(hours),sleeps]
[n(hours)]
[hours,active]
[hours]
[]
```

```
[active]
[n(gym),active]
[n(gym)]
[gym]
[gym]
[]
[]
Answer: YES
true .
```

### 1.2.2 Negative Case

```
?- main_backward.
Do you read books?
Type answer (yes/no): yes.
Do you eat vegetables?
Type answer (yes/no): yes.
How many hours do you sleep per night?
Type the number of hours (answer is a number): 8.
Do you drink water?
Type answer (yes/no): yes.
How many times you go to gym per week?
Type the number of times (answer is a number): 2.
```

#### Backward Chaining Solution

```
[[reads],[vegetables],[hours],[water],
[n(smart),n(healthy),fulfilled],
[n(eats),n(sleeps),n(active),healthy],
[n(gym),active],
[n(hours),sleeps],
[n(vegetables),n(water),eats],
[n(reads),smart]]
```

#### Steps

```
[n(smart),n(healthy),fulfilled]
[n(smart),n(healthy)]
[smart,healthy]
[n(reads),smart]
[n(reads)]
[reads,healthy]
[reads]
[]
[healthy]
[n(eats),n(sleeps),n(active),healthy]
[n(eats),n(sleeps),n(active)]
[eats,sleeps,active]
[n(vegetables),n(water),eats]
[n(vegetables),n(water)]
[vegetables,water,sleeps,active]
[vegetables]
[]
```

```
[water,sleeps,active]
[water]
[]
[sleeps,active]
[n(hours),sleeps]
[n(hours)]
[hours,active]
[hours]
[]
[active]
[n(gym),active]
[n(gym)]
[gym]
Answer: NO
true .
```

## 1.3 Forward Chaining Output

### 1.3.1 Positive Case

```
?- main_forward.
Do you read books?
Type answer (yes/no): yes.
Do you eat vegetables?
Type answer (yes/no): yes.
How many hours do you sleep per night?
Type the number of hours (answer is a number): 9.
Do you drink water?
Type answer (yes/no): yes.
How many times you go to gym per week?
Type the number of times (answer is a number): 4.
```

**Forward Chaining Solution**

```
[[fulfilled], [reads],[vegetables],[hours],[water],[gym],
[n(smart),n(healthy),fulfilled],
[n(eats),n(sleeps),n(active),healthy],
[n(gym),active],[n(hours),sleeps],
[n(vegetables),n(water),eats],
[n(reads),smart]]
```

**Steps**

```
Literals: [active,eats,fulfilled,gym,healthy,hours,reads,sleeps,smart,vegetables,water]
Init Unsolved:
[(active,0),(eats,0),(fulfilled,0),(gym,0),(healthy,0),(hours,0),(reads,0),(sleeps,0),(smart,0),(vegetables,0),(water,0)]
Selected Clause: [reads]
Positive Literal: reads
Unsolved:
[(active,0),(eats,0),(fulfilled,0),(gym,0),(healthy,0),(hours,0),(sleeps,0),(smart,0),(vegetables,0),(water,0),(reads,1)]
Selected Clause: [vegetables]
Positive Literal: vegetables
Unsolved:
[(active,0),(eats,0),(fulfilled,0),(gym,0),(healthy,0),(hours,0),(sleeps,0),(smart,0),(wat er,0),(reads,1),(vegetables,1)]
Selected Clause: [hours]
```

```
Positive Literal: hours
Unsolved:
[(active,0),(eats,0),(fulfilled,0),(gym,0),(healthy,0),(sleeps,0),(smart,0),(water,0),(reads,1),(vegetables,1),(hours,1)]
Selected Clause: [water]
Positive Literal: water
Unsolved:
[(active,0),(eats,0),(fulfilled,0),(gym,0),(healthy,0),(sleeps,0),(smart,0),(reads,1),(vegetables,1),(hours,1),(water,1)]
Selected Clause: [gym]
Positive Literal: gym
Unsolved:
[(active,0),(eats,0),(fulfilled,0),(healthy,0),(sleeps,0),(smart,0),(reads,1),(vegetables,1),(hours,1),(water,1),(gym,1)]
Selected Clause: [n(gym),active]
Positive Literal: active
Unsolved:
[(eats,0),(fulfilled,0),(healthy,0),(sleeps,0),(smart,0),(reads,1),(vegetables,1),(hours,1),(water,1),(gym,1),(active,1)]
Selected Clause: [n(hours),sleeps]
Positive Literal: sleeps
Unsolved:
[(eats,0),(fulfilled,0),(healthy,0),(smart,0),(reads,1),(vegetables,1),(hours,1),(water,1),(gym,1),(active,1),(sleeps,1)]
Selected Clause: [n(vegetables),n(water),eats]
Positive Literal: eats
Unsolved:
[(fulfilled,0),(healthy,0),(smart,0),(reads,1),(vegetables,1),(hours,1),(water,1),(gym,1),(active,1),(sleeps,1),(eats,1)]
Selected Clause: [n(eats),n(sleeps),n(active),healthy]
Positive Literal: healthy
Unsolved:
[(fulfilled,0),(smart,0),(reads,1),(vegetables,1),(hours,1),(water,1),(gym,1),(active,1),(sleeps,1),(eats,1),(healthy,1)]
Selected Clause: [n(reads),smart]
Positive Literal: smart
Unsolved:
[(fulfilled,0),(reads,1),(vegetables,1),(hours,1),(water,1),(gym,1),(active,1),(sleeps,1),(eats,1),(healthy,1),(smart,1)]
Selected Clause: [n(smart),n(healthy),fulfilled]
Positive Literal: fulfilled
Unsolved:
[(reads,1),(vegetables,1),(hours,1),(water,1),(gym,1),(active,1),(sleeps,1),(eats,1),(healthy,1),(smart,1),(fulfilled,1)]
Answer: YES
true.
```

### 1.3.2 Negative Case

```
?- main_forward.
Do you read books?
Type answer (yes/no): yes.
Do you eat vegetables?
Type answer (yes/no): yes.
How many hours do you sleep per night?
Type the number of hours (answer is a number): 9.
Do you drink water?
Type answer (yes/no): yes.
How many times you go to gym per week?
Type the number of times (answer is a number): 2.
```

**Forward Chaining Solution**

```
[[reads],[vegetables],[hours],[water],
[n(smart),n(healthy),fulfilled],
[n(eats),n(sleeps),n(active),healthy],
[n(gym),active],
[n(hours),sleeps],
[n(vegetables),n(water),eats],
[n(reads),smart]]
Literals: [active,eats,fulfilled,gym,healthy,hours,reads,sleeps,smart,vegetables,water]
Init Unsolved:
[(active,0),(eats,0),(fulfilled,0),(gym,0),(healthy,0),(hours,0),(reads,0),(sleeps,0),(smart,0),(vegetables,0),(water,0)]
Selected Clause: [reads]
Positive Literal: reads
Unsolved:
[(active,0),(eats,0),(fulfilled,0),(gym,0),(healthy,0),(hours,0),(sleeps,0),(smart,0),(vegetables,0),(water,0),(reads,1)]
Selected Clause: [vegetables]
Positive Literal: vegetables
Unsolved:
[(active,0),(eats,0),(fulfilled,0),(gym,0),(healthy,0),(hours,0),(sleeps,0),(smart,0),(water,0),(reads,1),(vegetables,1)]
Selected Clause: [hours]
Positive Literal: hours
Unsolved: [(active,0),(eats,0),(fulfilled,0),(gym,0),(healthy,0),(sleeps,0),(smart,0),(water,0),(reads,1),(vegetables,1),(ho
Selected Clause: [water]
Positive Literal: water
Unsolved:
[(active,0),(eats,0),(fulfilled,0),(gym,0),(healthy,0),(sleeps,0),(smart,0),(reads,1),(vegetables,1),(hours,1),(water,1)]
Selected Clause: [n(hours),sleeps]
Positive Literal: sleeps
Unsolved:
[(active,0),(eats,0),(fulfilled,0),(gym,0),(healthy,0),(smart,0),(reads,1),(vegetables,1),(hours,1),(water,1),(sleeps,1)]
Selected Clause: [n(vegetables),n(water),eats]
Positive Literal: eats
Unsolved:
[(active,0),(fulfilled,0),(gym,0),(healthy,0),(smart,0),(reads,1),(vegetables,1),(hours,1),(water,1),(sleeps,1),(eats,1)]
Selected Clause: [n(reads),smart]
Positive Literal: smart
Unsolved:
[(active,0),(fulfilled,0),(gym,0),(healthy,0),(reads,1),(vegetables,1),(hours,1),(water,1),(sleeps,1),(eats,1),(smart,1)]
Selected Vid -¿ Out: []
Answer: NO
true.
```

# 2 Fuzzy Aggregation and Defuzzification

The following rules are given:

**Rules:**

- If the surface of the apartment is small or the center of the city is far then the price is low.
- If the surface is medium then the price is normal.
- If the surface of the apartment is big or the center of the city is close then the price is high.

## 2.1 Degree of curves

Predicates: small, medium, large, closer, far, cheap, normal, expensive.

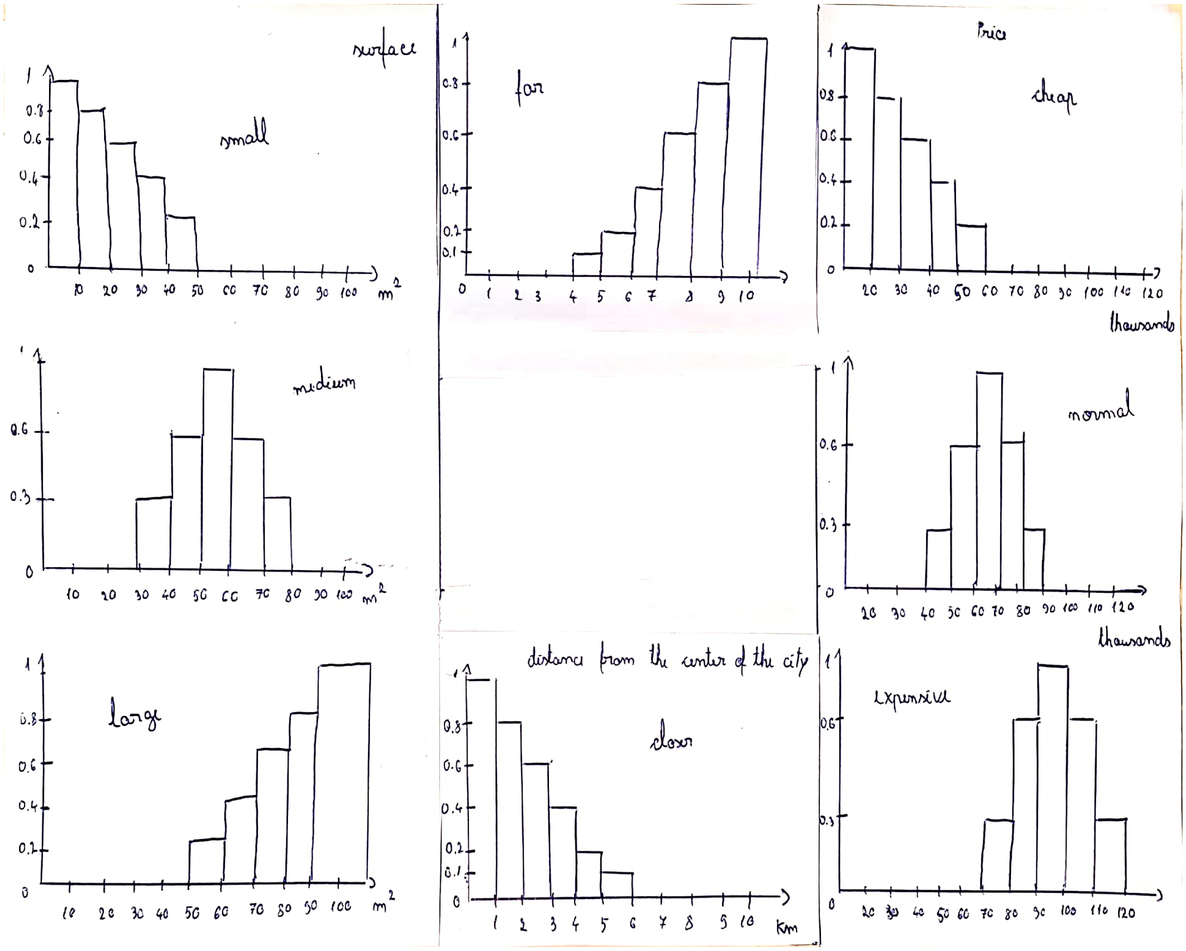

Figure 1: Degrees of curves for predicates

## 2.2 Input Representation

- $[or, [surface/small, distance/far], [price/cheap]]$
- $[and, [surface/medium], [price/normal]]$
- $[or, [surface/large, distance/closer], [price/expensive]]$

## 2.3 Experiments

1. surface = 20 squared and distance from the center of the city = 8 km ⟶ 20 thousands.
2. surface = 50 squared and distance from the center of the city = 5 km ⟶ 60 thousands.
3. surface = 80 squared and distance from the center of the city = 2 km ⟶ 90 thousands.

### 2.3.1 Output Experiment 1

```
?- main_fuzzy.
    What surface do you want the apartment to be?
    Type the surface in squared meters (answer is a number between 0 - 100): 20.
    How far from the center of the city you want the apartment?
    Type the distance from the center in km (answer is a number 0 - 10): 8.

    [[or,[surface/small,distance/far],[price/cheap]],
    [and,[surface/medium],[price/normal]],
    [or,[surface/large,distance/closer],[price/expensive]]]
```

**Steps**

```
    [or,[surface/small,distance/far],[price/cheap]]
    Function Applied: small
    X: 20
    Y: 0.6
    Function Applied: far
    X: 8
    Y: 0.8
    Operator Applied: or
    Result: 0.8
    [and,[surface/medium],[price/normal]]
    Function Applied: medium
    X: 20
    Y: 0.0
    Operator Applied: and
    Result: 0.0
    [or,[surface/large,distance/closer],[price/expensive]]
    Function Applied: large
    X: 20
    Y: 0.0
    Function Applied: closer
    X: 8
    Y: 0.0
    Operator Applied: or
    Result: 0.0
    Thresholds for the cut: [0.8,0.0,0.0]
    Price Domain: [20,30,40,50,60,70,80,90,100,110,120]
    Points After Remapping:
    [(20,0.8),(30,0.6),(40,0.4),(50,0.2),(60,0.0),(70,0.0),(80,0.0),(90,0.0),(100,0.0),(110,0.0),(120,0.0)]
    Centroid: 1
    Distances from centroid: [(20,0.2),(30,0.4),(40,0.6),(50,0.8),(60,1),(70,1),(80,1),(90,1),(100,1),(110,1),(120,1)]
    Recommended Price: 20
    true .
```

### 2.3.2 Output Experiment 2

```
?- main_fuzzy.
    What surface do you want the apartment to be?
    Type the surface in squared meters (answer is a number between 0 - 100): 50.
    How far from the center of the city you want the apartment?
    Type the distance from the center in km (answer is a number 0 - 10): 5.

    [[or,[surface/small,distance/far],[price/cheap]],
    [and,[surface/medium],[price/normal]],
    [or,[surface/large,distance/closer],[price/expensive]]]
```

**Steps**

```
    [or,[surface/small,distance/far],[price/cheap]]
    Function Applied: small
    X: 50
    Y: 0.0
    Function Applied: far
    X: 5
    Y: 0.2
    Operator Applied: or
    Result: 0.2
    [and,[surface/medium],[price/normal]]
    Function Applied: medium
    X: 50
    Y: 1.0
    Operator Applied: and
    Result: 1.0
    [or,[surface/large,distance/closer],[price/expensive]]
    Function Applied: large
    X: 50
    Y: 0.2
    Function Applied: closer
    X: 5
    Y: 0.1
    Operator Applied: or
    Result: 0.2
    Thresholds for the cut: [0.2,1.0,0.2]
    Price Domain: [20,30,40,50,60,70,80,90,100,110,120]
    Points After Remapping:
    [(20,0.2),(30,0.2),(40,0.3),(50,0.6),(60,1.0),(70,0.6),(80,0.3),(90,0.2),(100,0.2),(110,0.2),(120,0.0)]
    Centroid: 1.9
    Distances from centroid:
    [(20,1.7),(30,1.7),(40,1.6),(50,1.3),(60,0.9),(70,1.3),(80,1.6),(90,1.7),(100,1.7),(110,1.7),(120,1.9)]
    Recommended Price: 60
    true .
```

### 2.3.3 Output Experiment 3

```
?- main_fuzzy.
    What surface do you want the apartment to be?
    Type the surface in squared meters (answer is a number between 0 - 100): 80.
    How far from the center of the city you want the apartment?
    Type the distance from the center in km (answer is a number 0 - 10): 2.
```

```
    [[or,[surface/small,distance/far],[price/cheap]],
    [and,[surface/medium],[price/normal]],
    [or,[surface/large,distance/closer],[price/expensive]]]
```

**Steps**

```
    [or,[surface/small,distance/far],[price/cheap]]
    Function Applied: small
    X: 80
    Y: 0.0
    Function Applied: far
    X: 2
    Y: 0.0
    Operator Applied: or
    Result: 0.0
    [and,[surface/medium],[price/normal]]
    Function Applied: medium
    X: 80
    Y: 0.0
    Operator Applied: and
    Result: 0.0
    [or,[surface/large,distance/closer],[price/expensive]]
    Function Applied: large
    X: 80
    Y: 0.8
    Function Applied: closer
    X: 2
    Y: 0.6
    Operator Applied: or
    Result: 0.8
    Thresholds for the cut: [0.0,0.0,0.8]
    Price Domain: [20,30,40,50,60,70,80,90,100,110,120]
    Points After Remapping:
    [(20,0.0),(30,0.0),(40,0.0),(50,0.0),(60,0.0),(70,0.3),(80,0.6),(90,0.8),(100,0.6),(110,0.3),(120, 0.0)]
    Centroid: 1.3
    Distances from centroid:
    [(20,1.3),(30,1.3),(40,1.3),(50,1.3),(60,1.3),(70,1),(80,0.7),(90,0.5),(100,0.7),(110,1),(120,1.3)]
    Recommended Price: 90
    true .
```

# 3 Code Implementation

```
read_pipeline_chain(X) :-
see('/home/adrian/Desktop/Python/Personal/Master/Master-Projects/First-
    Year/Knowledge-Representation-and-Reasoning/Labs/Project-2/Input-1.txt
    '),
read(X), seen.

read_pipeline_fuzzy(X) :-
see('/home/adrian/Desktop/Python/Personal/Master/Master-Projects/First-
    Year/Knowledge-Representation-and-Reasoning/Labs/Project-2/Input-2.txt
    '),
read(X), seen.

print(Msg, X) :- write(Msg), write(X), nl.
print(X) :- write(X), nl.

ask(Question, Prompt, Response) :- write(Question), prompt(_, Prompt), nl
    , read(Response), nl.

do_continue(Function) :- ask('Do␣you␣want␣to␣continue?', 'Type␣answer␣(
    yes/stop):␣', Stop), Stop \= stop, call(Function).
do_continue(_) :- ask('Are␣you␣sure?', 'Type␣answer␣(yes/no):␣', Stop),
    Stop = yes.

read_books(yes, true).
read_books(no, false).

eat_vegetables(yes, true).
eat_vegetables(no, true).

hours_slept(Hours, true) :- Hours > 7, !.
hours_slept(_, false).

drink_water(yes, true).
drink_water(no, false).

gym_per_week(Times, true) :- Times > 3, !.
gym_per_week(_, false).

update_answer_list(true, Value, List, Out) :- append(List, Value, Out),
    !.
update_answer_list(false, _, List, List).

add_answears_to_list(Books, Vegetables, HoursSlept, DrinksWater, Gym,
    List) :-
add_answears_to_list_(Books, Vegetables, HoursSlept, DrinksWater, Gym,
    [], List).

add_answears_to_list_(Books, Vegetables, HoursSlept, DrinksWater, Gym,
    Vid, List) :-
update_answer_list(Books, [[reads]], Vid, AddedBooks),
update_answer_list(Vegetables, [[vegetables]], AddedBooks,
    AddedVegetables),
update_answer_list(HoursSlept, [[hours]], AddedVegetables,
    AddedHoursSlept),
update_answer_list(DrinksWater, [[water]], AddedHoursSlept, AddedWater),
update_answer_list(Gym, [[gym]], AddedWater, AddedGym),
List = AddedGym.

backward_chaining([], _, ”YES”) :- !.
backward_chaining(Literals, KB, Answer) :- for_each_clause(Literals, KB,
    KB, Answer).

for_each_clause(_, [], _, ”NO”) :- !.

for_each_clause([Lit | Lits], [Clause | _], KB, Answer) :-
member(Lit, Clause),
select(Lit, Clause, Removed),
negate_list(Lits, Removed, Resulted),
backward_chaining(Resulted, KB, Answer).


for_each_clause([Lit | Lits], [Clause | Clauses], KB, Answer) :-
not(member(Lit, Clause)),
for_each_clause([Lit | Lits], Clauses, KB, Answer).

negate_list(Sentences, [], Sentences) :- !.
negate_list(Sentences, [n(P) | Tail], [P | Resulted]) :-
negate_list(Sentences, Tail, Resulted).

main_backward :-
ask('Do␣you␣read␣books?', 'Type␣answer␣(yes/no):␣', Books), Books \= stop
    ,
ask('Do␣you␣eat␣vegetables?', 'Type␣answer␣(yes/no):␣', Vegetables),
    Vegetables \= stop,
ask('How␣many␣hours␣do␣you␣sleep␣per␣night?', 'Type␣the␣number␣of␣hours␣(
    answer␣is␣a␣number):␣', HoursSlept), HoursSlept \= stop,
ask('Do␣you␣drink␣water?', 'Type␣answer␣(yes/no):␣', DrinksWater),
    DrinksWater \= stop,
ask('How␣many␣times␣you␣go␣to␣gym␣per␣week?', 'Type␣the␣number␣of␣times␣(
    answer␣is␣a␣number):␣', Gym), Gym \= stop,

read_books(Books, BooksAnswer),
eat_vegetables(Vegetables, VegetablesAnswer),
hours_slept(HoursSlept, HoursAnswer),
drink_water(DrinksWater, DrinkAnswer),
gym_per_week(Gym, GymAnswer),

read_pipeline_chain(ClausesKB),
add_answears_to_list(BooksAnswer, VegetablesAnswer, HoursAnswer,
    DrinkAnswer, GymAnswer, Clauses),
union(Clauses, ClausesKB, KB),

print(”Backward␣Chaining␣Solution”),
print(KB),
backward_chaining([fulfilled], KB, Ans),
print(”Answer:␣”, Ans),
do_continue(main_backward).
















get_literals_from_clause_([], Literals, Literals).
get_literals_from_clause_([n(Lit) | Lits], Literals, Aux) :-
get_literals_from_clause_(Lits, Literals, [Lit | Aux]).

get_literals_from_clause_([Lit | Lits], Literals, Aux) :-
get_literals_from_clause_(Lits, Literals, [Lit | Aux]).

get_literals_from_clause([], []).
get_literals_from_clause(Clause, Literals) :-
get_literals_from_clause_(Clause, Literals, []).

get_literals_from_clauses_([], Literals, Aux) :- sort(Aux, Literals).
get_literals_from_clauses_([Clause | Clauses], Literals, Aux) :-
get_literals_from_clause(Clause, Lits), append(Lits, Aux, Union),
get_literals_from_clauses_(Clauses, Literals, Union).

get_literals_from_clauses([], []).
get_literals_from_clauses(Clauses, Lits) :-
get_literals_from_clauses_(Clauses, Lits, []).

initialize_unsolved([], []) :- !.
initialize_unsolved([Lit | Lits], [(Lit, 0) | Result]) :-
initialize_unsolved(Lits, Result).

check_all_goals_are_solved([], _).
check_all_goals_are_solved([Lit | Lits], Solved) :-
member((Lit, 1), Solved),
check_all_goals_are_solved(Lits, Solved), !.


how_many_negatives([], 0) :- !.
how_many_negatives([n(_) | Lits], Result) :- how_many_negatives(Lits, Aux
    ), Result is Aux + 1, !.
how_many_negatives([_ | Lits], Result) :- how_many_negatives(Lits, Result
    ), !.

get_negative_literals([], []).
get_negative_literals([n(Lit) | Lits], [Lit | Result]) :-
    get_negative_literals(Lits, Result), !.
get_negative_literals([_ | Lits], Result) :- get_negative_literals(Lits,
    Result), !.


get_positive_literals([], []).
get_positive_literals([n(_) | Lits], Result) :- get_positive_literals(
    Lits, Result), !.
get_positive_literals([Lit | Lits], [Lit | Result]) :-
    get_positive_literals(Lits, Result), !.


get_positive_literal([], []).
get_positive_literal([n(_) | Lits], Result) :- get_positive_literal(Lits,
     Result), !.
get_positive_literal([Lit | _], Lit) :- !.

check_negatives_solved([], _, ”YES”).
check_negatives_solved([Lit | Lits], Unsolved, Ans) :- member((Lit, 1),
    Unsolved),
check_negatives_solved(Lits, Unsolved, Ans), !.

check_negatives_solved([Lit | _], Unsolved, ”NO”) :-
not(member((Lit, 1), Unsolved)), !.


check_positive_unsolved([], _, ”YES”).
check_positive_unsolved([Lit | Lits], Unsolved, Ans) :- member((Lit, 0),
    Unsolved),
check_positive_unsolved(Lits, Unsolved, Ans), !.

check_positive_unsolved([Lit | _], Unsolved, ”NO”) :-
not(member((Lit, 0), Unsolved)), !.


number_condition(Clause, ”YES”) :-
how_many_negatives(Clause, NegativesCounter), length(Clause, Counter),
AssumedNegative is Counter - 1, NegativesCounter == AssumedNegative, !.

number_condition(_, ”NO”).

solved_condition(Clause, Unsolved, ”YES”) :-
get_negative_literals(Clause, Lits), check_negatives_solved(Lits,
    Unsolved, ”YES”),
get_positive_literals(Clause, Pos), check_positive_unsolved(Pos, Unsolved
    , ”YES”), !.

solved_condition(_, _, ”NO”).

check_conditions(Clause, Unsolved, ”YES”) :-
number_condition(Clause, ”YES”), solved_condition(Clause, Unsolved, ”YES”
    ), !.
```

```
check_conditions(_, _, ”NO”).

select_clause([], _, []).
select_clause([Clause | _], Unsolved, Clause) :- check_conditions(Clause,
     Unsolved, ”YES”), !.
select_clause([_ | Clauses], Unsolved, Result) :- select_clause(Clauses,
    Unsolved, Result), !.

forward_chaining_(Goal, Unsolved, _, ”YES”) :- check_all_goals_are_solved
    (Goal, Unsolved), !.

forward_chaining_(Goal, Unsolved, KB, Ans) :-
select_clause(KB, Unsolved, Selected), Selected \= [],
get_positive_literal(Selected, Pos),
delete(Unsolved, (Pos, 0), Aux1), append(Aux1, [(Pos, 1)], Aux2),
forward_chaining_(Goal, Aux2, KB, Ans), !.

forward_chaining_(_, Unsolved, KB, ”NO”) :-  select_clause(KB, Unsolved,
    Selected), Selected == [], !.

forward_chaining(Goal, KB, Ans) :-
append([Goal], KB, KBComplete),
get_literals_from_clauses(KBComplete, Literals),
initialize_unsolved(Literals, Unsolved),
forward_chaining_(Goal, Unsolved, KB, Ans), !.


main_forward :-
ask('Do␣you␣read␣books?', 'Type␣answer␣(yes/no):␣', Books), Books \= stop
    ,
ask('Do␣you␣eat␣vegetables?', 'Type␣answer␣(yes/no):␣', Vegetables),
    Vegetables \= stop,
ask('How␣many␣hours␣do␣you␣sleep␣per␣night?', 'Type␣the␣number␣of␣hours␣(
    answer␣is␣a␣number):␣', HoursSlept), HoursSlept \= stop,
ask('Do␣you␣drink␣water?', 'Type␣answer␣(yes/no):␣', DrinksWater),
    DrinksWater \= stop,
ask('How␣many␣times␣you␣go␣to␣gym␣per␣week?', 'Type␣the␣number␣of␣times␣(
    answer␣is␣a␣number):␣', Gym), Gym \= stop,

read_books(Books, BooksAnswer),
eat_vegetables(Vegetables, VegetablesAnswer),
hours_slept(HoursSlept, HoursAnswer),
drink_water(DrinksWater, DrinkAnswer),
gym_per_week(Gym, GymAnswer),

read_pipeline_chain(ClausesKB),
add_answears_to_list(BooksAnswer, VegetablesAnswer, HoursAnswer,
    DrinkAnswer, GymAnswer, Clauses),
union(Clauses, ClausesKB, KB),

print(”Forward␣Chaining␣Solution”),
print(KB),
forward_chaining([fulfilled], KB, Ans),
print(”Answer:␣”, Ans),
do_continue(main_forward).




round(X,Y,D) :- Z is X * 10^D, round(Z, ZA), Y is ZA / 10^D.

small(Surface, Percent) :- Surface < 10, Percent is 1.0, !.
small(Surface, Percent) :- Surface < 20, Percent is 0.8, !.
small(Surface, Percent) :- Surface < 30, Percent is 0.6, !.
small(Surface, Percent) :- Surface < 40, Percent is 0.4, !.
small(Surface, Percent) :- Surface < 50, Percent is 0.2, !.
small(_, Percent) :- Percent is 0.0, !.


medium(Surface, Percent) :- Surface < 30, Percent is 0.0, !.
medium(Surface, Percent) :- Surface < 40, Percent is 0.3, !.
medium(Surface, Percent) :- Surface < 50, Percent is 0.6, !.
medium(Surface, Percent) :- Surface < 60, Percent is 1.0, !.
medium(Surface, Percent) :- Surface < 70, Percent is 0.6, !.
medium(Surface, Percent) :- Surface < 80, Percent is 0.3, !.
medium(_, Percent) :- Percent is 0.0, !.


large(Surface, Percent) :- Surface < 50,  Percent is 0.0, !.
large(Surface, Percent) :- Surface < 60,  Percent is 0.2, !.
large(Surface, Percent) :- Surface < 70,  Percent is 0.4, !.
large(Surface, Percent) :- Surface < 80,  Percent is 0.6, !.
large(Surface, Percent) :- Surface < 90,  Percent is 0.8, !.
large(Surface, Percent) :- Surface < 100, Percent is 1.0, !.
large(_, Percent) :- Percent is 1.0, !.


closer(Distance, Percent) :- Distance < 1, Percent is 1.0, !.
closer(Distance, Percent) :- Distance < 2, Percent is 0.8, !.
closer(Distance, Percent) :- Distance < 3, Percent is 0.6, !.
closer(Distance, Percent) :- Distance < 4, Percent is 0.4, !.
closer(Distance, Percent) :- Distance < 5, Percent is 0.2, !.
closer(Distance, Percent) :- Distance < 6, Percent is 0.1, !.
closer(_, Percent) :- Percent is 0.0, !.


far(Distance, Percent) :- Distance < 4, Percent is 0.0, !.
far(Distance, Percent) :- Distance < 5, Percent is 0.1, !.
far(Distance, Percent) :- Distance < 6, Percent is 0.2, !.
far(Distance, Percent) :- Distance < 7, Percent is 0.4, !.
far(Distance, Percent) :- Distance < 8, Percent is 0.6, !.
far(Distance, Percent) :- Distance < 9, Percent is 0.8, !.
far(Distance, Percent) :- Distance < 10, Percent is 1.0, !.
far(_, Percent) :- Percent is 1.0, !.



cheap(Price, Percent) :- Price < 20, Percent is 1.0, !.
cheap(Price, Percent) :- Price < 30, Percent is 0.8, !.
cheap(Price, Percent) :- Price < 40, Percent is 0.6, !.
cheap(Price, Percent) :- Price < 50, Percent is 0.4, !.
cheap(Price, Percent) :- Price < 60, Percent is 0.2, !.
cheap(_, Percent) :- Percent is 0.0, !.

normal(Price, Percent) :- Price < 40, Percent is 0.0, !.
normal(Price, Percent) :- Price < 50, Percent is 0.3, !.
normal(Price, Percent) :- Price < 60, Percent is 0.6, !.
normal(Price, Percent) :- Price < 70, Percent is 1.0, !.
normal(Price, Percent) :- Price < 80, Percent is 0.6, !.
normal(Price, Percent) :- Price < 90, Percent is 0.3, !.
normal(_, Percent) :- Percent is 0.0, !.

expensive(Price, Percent) :- Price < 70, Percent is 0.0, !.
expensive(Price, Percent) :- Price < 80, Percent is 0.3, !.
expensive(Price, Percent) :- Price < 90, Percent is 0.6, !.
expensive(Price, Percent) :- Price < 100, Percent is 1.0, !.
expensive(Price, Percent) :- Price < 110, Percent is 0.6, !.
expensive(Price, Percent) :- Price < 120, Percent is 0.3, !.
expensive(_, Percent) :- Percent is 0.0, !.

interval([20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120]).

or(List, Result)  :- max_list(List, Result).
and(List, Result) :- min_list(List, Result).

decomposition([Operator, Conditions, Conclusion], Operator, Conditions,
   Conclusion).

iterate_conditions([], _, []).
iterate_conditions([(_ / Degree) | Conditions], [Score | Scores], [
   Percent | Result]) :-
call(Degree, Score, Percent), iterate_conditions(Conditions, Scores,
   Result).

argmin([], _, R, R).
argmin([(X, Y) | Points], Min, _, R):- Y <  Min, argmin(Points, Y, X, R).
argmin([(_, Y) | Points], Min, Idx, R):- Y >= Min, argmin(Points, Min,
    Idx, R).
argmin([(X, Y) | Points], R):- argmin(Points, Y, X, R).

iterate([], _, []).
iterate([Premise | Premises], Scores, [Result | Results]) :-
decomposition(Premise, Operator, Conditions, _),
iterate_conditions(Conditions, Scores, Percentages),
call(Operator, Percentages, Result),
iterate(Premises, Scores, Results).


remap_function(Function, Threshold, X, R) :-
call(Function, X, Y), R is min(Y, Threshold).

construct_function_(_, [], [], []).
construct_function_(X, [Premise | Premises], [Threshold | Thresholds], [Y
     | Ys]) :-
decomposition(Premise, _, _, [_ / Degree]),
remap_function(Degree, Threshold, X, Y),
construct_function_(X, Premises, Thresholds, Ys).


construct_function([], _, _, []).
construct_function([X | Xs], Premises, Thresholds, [(X, Y) | Points]) :-
construct_function_(X, Premises, Thresholds, Values), or(Values, Y),
construct_function(Xs, Premises, Thresholds, Points).

compute_distances([], _, []).
compute_distances([(X, Y) | Points], Objective, [(X, Rounded) | Distances
    ]) :-
Aux is Objective - Y, abs(Aux, Distance),
round(Distance, Rounded, 2),
compute_distances(Points, Objective, Distances).

extract_ys([], []).
extract_ys([(_, Y) | Points], [Y | Ys]) :- extract_ys(Points, Ys).

main_fuzzy :-
ask('What␣surface␣do␣you␣want␣the␣apartment␣to␣be?', 'Type␣the␣surface␣in
    ␣squared␣meters␣(answer␣is␣a␣number␣between␣0␣-␣100):␣', Surface),
    Surface \= stop,
ask('How␣far␣from␣the␣center␣of␣the␣city␣you␣want␣the␣apartment?', 'Type␣
    the␣distance␣from␣the␣center␣in␣km␣(answer␣is␣a␣number␣0␣-␣10):␣',
    Distance), Distance \= stop,

read_pipeline_fuzzy(Premises),
iterate(Premises, [Surface, Distance], Thresholds),
interval(Domain),
construct_function(Domain, Premises, Thresholds, Points),
extract_ys(Points, Ys),
sumlist(Ys, Sum), Objective is Sum / 2, round(Objective, _, 2),
compute_distances(Points, Objective, Distances),
argmin(Distances, Price), print("Recommended␣Price:␣", Price),
do_continue(main_fuzzy).
```