Proiect 2 KRR

Preda Alexandru-Florin

Subject 1

The first step in solving requirements of laboratory 5, is to create our own Rules, Questions and The Goal. I've inspired myself by how you can determine if a computer has system failures. The Rules that I created are:

1. If cpu temperature is greater than 85 then it is overheating.
2. If cpu is overheating and has fan failure, then it has thermal shutdown.
3. If it is used more than 9 hours and has thermal shutdown then it has system instability.
4. If it has system instability and has data corruption then it has system failure.

The questions that you need to answer to find if your system has a failure are:

1. What is the CPU temperature?
2. For how many hours has the system been running continuously?
3. Is the cooling fan failing?
4. Is there data corruption detected?

And the goal is to see, based on the answers to the question, if the system has a failure.

To solve the laboratory, I split the algorithm into several parts:

1. Reading the Rules from the file
2. Rule parsing grammar
3. Horn KB
4. Reading answers from Java + turning them into Facts
5. Forward chaining
6. Backward chaining v1 (DFS)
7. "Second approach" using Prolog backward mechanism

1. Reading the Rules from the file

To read the knowledge base from a file, I implemented the function **read_scneraion/4**. This receives as input the path to a .txt file that contains the rules, questions and the goal, written in a natural language like format.

The function's purpose is to process the raw textual input and extract the information needed for inference.

First, the entire file is read as a single string. This string is then split into individual lines and normalized using trimming, splitting and removing extra whitespaces.

Then the string is divided into sections (Rules, Questions and the Goal) by identifying fixed headers. In addition, during this processing part, the function idetifies comparison terms, such as **gt(id, value)** and **ge(id, value)** that appears in the rule premises. These terms are collected separately because they depend on numeric user inputs and must be evaluated dinamically during runtime, after the user's answers are received.

2. Rule Parsing Grammar

This is the part where text becomes logic. I implemented a restricted grammar-based parser where each rule is required to follow the controller format:

- **If <condition> [and <condition>]* then <condition>.**

First, the string rule is normalized and validated to ensure it starts with the keyword **If** and contains a **then** separator. The left side of the rule is split into individual conditions connected by **and,** while the right side represents the rule's conclusion.

Each condition is then parsed using a Grammar that supports boolean predicates and numeric comparison predicates. The Grammar converts valid conditions into Prolog terms.

As a results, every string rule is mapped into an representation of this form:

- **Rule([Premise1, Premise2, ...], Conclusion)**

This structure will be later used to construct Horn Clauses and apply forward and backward chaining algorithms.

3. Horn KB

After parsing the string rules, I created another function **roles_to_horn_kb** that converts the rules into a Horn clause representation used for logical inference. Each parsed rule is transformed into a propositional Horn clause represented as a list, where all premises are negated, and the conclusion remains positive, similarly, to converting from FOL. This representation is equivalent to Horn clauses used in logical inference, where premises are treated as conditions that must already hold.

4. Reading answers from Java + turning them into Facts

User input is received from GUI as a sequence of answer lines send throught a socket connection. Each answer is parsed and classified to its type. Numeric answers are stored as value associations between identifiers and numbers, while boolean answers directly generate propositional facts when their value is true.

Numeric values are not immediately treated as facts. Later, they are evaluated against comparison predicates that appear in the rule of premises. Only when a numeric condition is satisfied, it is then added to the set of facts.

At the end, all boolean facts and derived comparison facts are combined into a single initial fact set. This set represents the known true information provided by the user and represents the starting point for reasoning algorithms.

5. Forward chaining

Forward chaining is used to infer new facts from a knowledge base expressed as Horn clauses. The algorithm starts from an initial set of known true facts and applies rules whose premises are satisified, deriving new facts until no further information can be inferred.

The reasoning process continues, until an interation produces no new facts. Then the goal is considered entailed if it appears in this closure.

6. Backward Chaining (Depth-First Search)

Backward chaining is an algorithm that proves if a given goal entailes from answers and kb by recursively proving the premises of rules that could derive it. This algorithm, instead of deriving everything that can be derived, itstarts from the goal and tries to prove it by recursively proving the premises of rules that could conclude it. So instead of trying to derive from specific facts, it tried to prove a specific goal. A visited list is used to prevent infinite loops caused by cyclic rule dependencies. This prevents infinite recursion in the presence of cyclic rule dependencies. The goal is considered entailed if a complete proof can be structured from the initial facts and the knowledge base.

7. “Second approach” using Prolog backward mechanism

In the second approach, the backward mechanism is performed using Prolog’s own reasoning algorithm. The rules parsed from the file are stored as facts, and the known facts are obtained from user input. To check if the goal is true, the algorithm “asks” Prolog to prove it. A goal is considered proven if it is a known fact or if there exists a rule, and the whole conclusion matches the goal. Prolog automatically handles the search and backtracking, using the prolog’s built-in capabilities.

Subject 2

This exercise works with degrees of truth, allowing us to model uncertainty and vague concepts, unlike the exercise 1, where predicates were either true of false.

I designed a scenario inspired by computer performance evaluation. Instead of making a strict decision, the system provides a graded recommendation based on how well the computer performs.

The rules that I defined are:

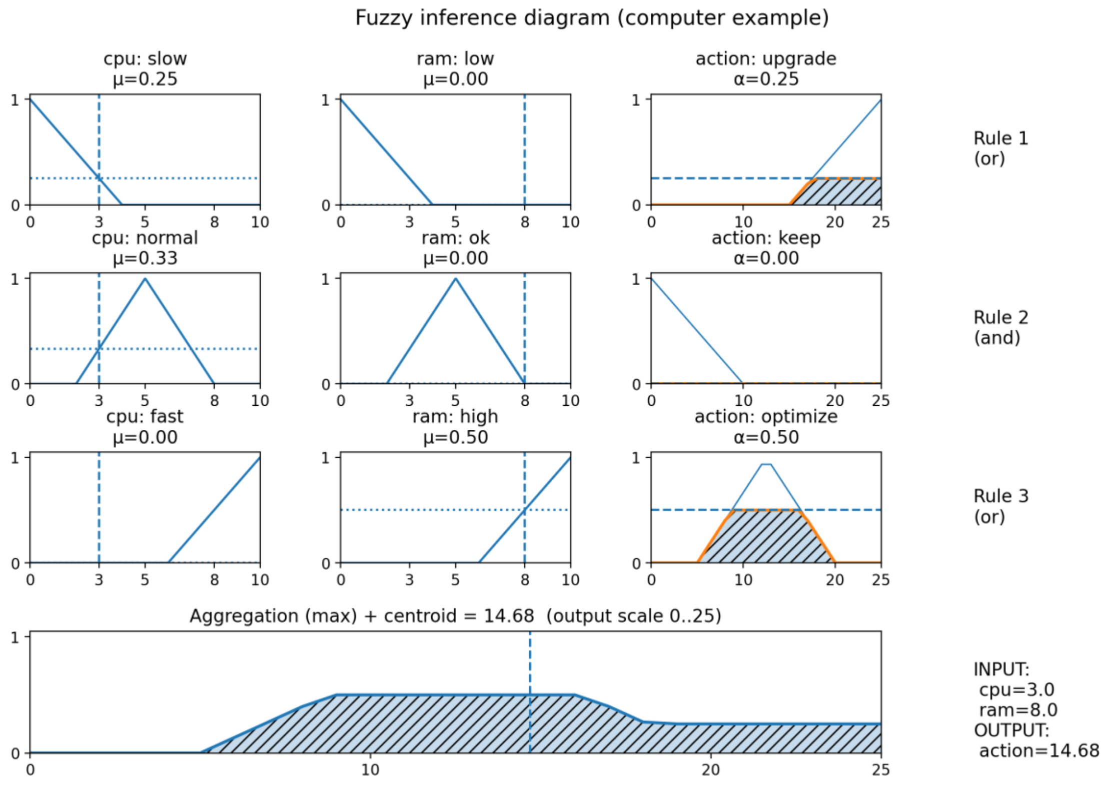
1. If the CPU is slow or the RAM is low, then the action is upgrade.
2. If the CPU is normal and the RAM is ok, then the action is keep.
3. If the CPU is fast or the RAM is high, then the action is optimize.

The questions that the user must answer are:

1. What is the CPU performance rating? (number on a 0–10 scale)
2. What is the RAM adequacy rating? (number on a 0–10 scale)

The goal of the system is to determine, based on the user's answers, the **recommended action** for the system. The result is a numerical value in the interval 0–25, which represents the degree to which the final recommendation applies.

For each predicate that is in the rules, a degree curve is defined. They define how a numerical input maps to a degree of truth between 0 and 1.



The reasoning process was split into several clearly defined steps:

1. Reading the rules and the goal from the knowledge base file
2. Parsing the rules using a grammar
3. Reading the user's answers from the Java interface
4. Transforming crisp inputs into degrees of applicability using membership functions
5. Evaluating the antecedents of the rules using fuzzy operators (AND / OR)
6. Computing and aggregating the fuzzy consequents into a single output curve
7. Defuzzifying the aggregated curve using the centroid method to obtain a crisp result

1. Reading the rules and the goal from knwoledge base file

In this section, similarly to Subject 1, we are reading the content of a file that represents the kb. The format expected in this file is to have the Rules, Questions and The Goal well defined, with specific title and each line to start with dash followed by if and end with dot or question mark. In this part we are loading the file based on path input, splitting it into lines, trimming spaces and extracting Rules and The Goal.

2. Parsing the rules using a grammar

Using a grammar to recognize "if .. then .." structure and convert it into a structured prologue term. For example, if the input is "-If service is poor or food is rancid then tip is cheap.", the output after parsing it with the grammar will be "rule(or, [service/poor, food/rancid], tip/cheap)"

This grammar uses a strict template:

- Starts with –If
- Antecedent is either:
  - One literal :x is y
  - Two ltierals: x is y (and/or) z is w
- Then then
- Consequent: a is b
- Ends with a .

3. Reading the user's answers from the Java interface

In this section we are reading from Java interface the given answers until "done" is receive. The expected input from java is "ans:service=5.0" from which, we extract a list of values in this form "value(service, 5.0)". The expected input are numeric inputs from 0 to 10 for membership functions.

4. Transforming crips inputs into degrees of applicability using membership functions

This part represents step 1 from the algorithm presented at course. Based on the numerical crisp values provided by the user, I converted them into fuzzy degrees. Each crisp input, on a 0-10 scale, is evaluated against a predefined membership function that describes vague linguistic predicates.

I used three basic shapes: left-shoulder functions, triangular functions and right-shoulder functions.

5. Evaluating the antecedents of the rules using fuzzy opperators (AND / OR)

After each antecedent literal has been fuzzified, I evaluated the antecedent of each rule as a whole. A rule antecedent consists of one or two literals connected by a logical operator (AND or OR).

For each literal, its degree is retrieved from the fuzzification step. These degrees are combined using standard fuzzy logic operators: The AND operator is implemented using the minimum function and the OR operator is implemented using the maximum function.

The resulted value of this combination in a single value expresses how strongly the rule’s conditions are satisfied by the current inputs. This part represents step 2 from the algorithm presented at the course.

6. Computing and aggregating the fuzzy consequents into a single output curve

After computing the degree of each rule, I evaluated the consequences. Each consequent is associated with a fuzzy predicate defined over the output domain.

For each rule, their representative membership function is clipped at the rule’s applicability degree. Mathematically, this is done by taking the minimum between the rule applicability degree and the membership value of the consequent at each output point.

The result of this step is a single aggregated fuzzy output curve, which represents the combined influence of all applicable rules.

7. Defuzzifying the aggregated curve using the centroid method to obtain a crisp result

The final step converts the aggregated fuzzy output curve into a single crisp numerical value, a process called defuzzification.

The centroid method is used. It computes the weighted average of all output values, where each value is weighted by its corresponding degree in the aggregated curve. The centroid is computed as a ratio between the sum of each output value multiplied by its membership degree and the sum of all membership degrees.

$$\frac{\sum_{y=0}^{25}\left(y \cdot \mu(y)\right)}{\sum_{y=0}^{25} \mu(y)}$$

Exercise1.pl

```prolog
:- use_module(library(socket)).
:- use_module(library(readutil)).
:- use_module(library(lists)).
:- use_module(library(apply)).
:- dynamic fact/1.
:- dynamic rule_impl/2.

:- initialization(main, main).

main(Argv) :-
    ( Argv = [PortAtom, ScenarioPathAtom | _] ->
        atom_number(PortAtom, Port),
        atom_string(ScenarioPathAtom, ScenarioPath),
        run_client(Port, ScenarioPath)
    ; format(user_error, "Usage: swipl -s engine.pl -- <PORT> <SCENARIO_FILE>~n", []),
      halt(2)
    ).

run_client(Port, ScenarioPath) :-
    tcp_socket(Sock),
    tcp_connect(Sock, '127.0.0.1':Port),
    tcp_open_socket(Sock, In, Out),
    handle_session(In, Out, ScenarioPath),
    close(In),
    close(Out).

handle_session(In, Out, ScenarioPath) :-
    read_scenario(ScenarioPath, RuleLines, GoalTerm, ComparisonTerms0),
    maplist(parse_rule_line, RuleLines, ParsedRules),
    rules_to_horn_kb(ParsedRules, HornKB),
    sort(ComparisonTerms0, ComparisonTerms),

    retractall(fact(_)),
    retractall(rule_impl(_,_)),
    read_answers(In, Values, BoolFacts0),
    derive_comparison_facts(ComparisonTerms, Values, CmpFacts),
    append(BoolFacts0, CmpFacts, Facts0),
    sort(Facts0, Facts),

    ( forward_chaining(HornKB, Facts, GoalTerm) -> FC=entailed ; FC=not_entailed ),
    ( backward_chaining(HornKB, Facts, GoalTerm) -> BC=entailed ; BC=not_entailed ),

    assert_kb_as_prolog(ParsedRules),
    assert_facts_as_prolog(Facts),
    ( prove_goal_builtin(GoalTerm) -> PBC=entailed ; PBC=not_entailed ),

    format(Out, "Facts: ~w~n", [Facts]),
    format(Out, "Goal: ~w~n", [GoalTerm]),
    format(Out, "Forward chaining: ~w~n", [FC]),
    format(Out, "Backward chaining v1: ~w~n", [BC]),
    format(Out, "Prolog built-in BC: ~w~n", [PBC]),
    format(Out, "~w~n", [FC]),
    flush_output(Out).

read_scenario(Path, RuleLines, GoalTerm, ComparisonTerms) :-
    read_file_to_string(Path, S, []),
    split_string(S, "\n", "\r", Lines0),
    maplist(string_trim, Lines0, Lines),
    extract_rules(Lines, RuleLines),
    extract_goal(Lines, GoalStr),
    parse_condition_string(GoalStr, GoalTerm),
    extract_comparisons(RuleLines, ComparisonTerms).

string_trim(In, Out) :-
    normalize_space(string(Out), In).

extract_rules(Lines, RuleLines) :-
    drop_until("Rules:", Lines, AfterRules),
    take_until_any(["Questions:", "The goal:"], AfterRules, RuleSection),
    include(is_bullet, RuleSection, Bullets),
    maplist(strip_bullet, Bullets, RuleLines).

extract_goal(Lines, GoalStr) :-
    drop_until("The goal:", Lines, AfterGoal),
    include(is_bullet, AfterGoal, [First|_]),
    strip_bullet(First, GoalStr).

is_bullet(Line) :-
    string_length(Line, L), L > 1,
    sub_string(Line, 0, 1, _, "-").

strip_bullet(Line, Out) :-
    sub_string(Line, 1, _, 0, Out0),
    string_trim(Out0, Out).

drop_until(_, [], []).
drop_until(Marker, [Marker|Rest], Rest) :- !.
drop_until(Marker, [_|Rest], Out) :- drop_until(Marker, Rest, Out).

take_until_any(_, [], []).
take_until_any(Markers, [H|_], []) :- member(H, Markers), !.
take_until_any(Markers, [H|T], [H|Out]) :- take_until_any(Markers, T, Out).

extract_comparisons(RuleLines, ComparisonTerms) :-
    findall(Cmp,
        ( member(Line, RuleLines),
          parse_rule_line(Line, rule(Ps,_)),
          member(Cmp, Ps),
          is_comparison(Cmp)
        ),
        Cmps),
    sort(Cmps, ComparisonTerms).

is_comparison(gt(_, _)).
is_comparison(ge(_, _)).

parse_rule_line(Line, rule(Premises, Head)) :-
    string_trim(Line, L1),
    ensure_period(L1, L2),
    ( sub_string(L2, 0, _, _, "If ") -> sub_string(L2, 3, _, 0, Body)
    ; throw(bad_rule_missing_if(Line))
    ),
    split_on_then(Body, Left, Right),
    split_on_and(Left, PremStrs),
    maplist(parse_condition_string, PremStrs, Premises),
    parse_condition_string(Right, Head).

ensure_period(Line, Out) :-
    ( sub_string(Line, _, 1, 0, ".") -> Out = Line
    ; string_concat(Line, ".", Out)
    ).

split_on_then(S, Left, Right) :-
    ( sub_string(S, ThenPos, _, After, " then ") ->
        sub_string(S, 0, ThenPos, _, Left0),
        sub_string(S, _, After, 0, Right0),
        string_trim(Left0, Left),
        string_trim(Right0, Right)
    ; throw(bad_rule_missing_then(S))
    ).

split_on_and(S, PartsTrimmed) :-
    split_on_substring(" and ", S, Parts0),
    maplist(string_trim, Parts0, PartsTrimmed).

split_on_substring(Delim, S, [Left|Rest]) :-
    ( sub_string(S, Pos, _, After, Delim) ->
        sub_string(S, 0, Pos, _, Left0),
        sub_string(S, _, After, 0, Right0),
        string_trim(Left0, Left),
        split_on_substring(Delim, Right0, Rest)
    ; string_trim(S, Left),
      Rest = []
    ).

parse_condition_string(S0, Term) :-
    string_trim(S0, S1),
    remove_trailing_period(S1, S),
    string_codes(S, Cs),
    phrase(condition(Term), Cs), !.
parse_condition_string(S, _) :-
    throw(bad_condition(S)).
```

```
remove_trailing_period(S, Out) :-
    ( sub_string(S, _, 1, 0, ".") ->
        sub_string(S, 0, _, 1, Out)
    ; Out = S
    ).

condition(has(A)) --> "has(", atom_name(A), ")".
condition(gt(A,N)) --> "gt(", atom_name(A), ",", ws0, number_string(N), ")".
condition(ge(A,N)) --> "ge(", atom_name(A), ",", ws0, number_string(N), ")".

ws0 --> [C], { code_type(C, space) }, !, ws0.
ws0 --> [].

atom_name(A) --> atom_chars(Cs), { atom_codes(A, Cs) }.
atom_chars([C|Cs]) --> [C], { valid_atom_char(C) }, !, atom_chars(Cs).
atom_chars([]) --> [].
valid_atom_char(C) :- code_type(C, alnum) ; C=:=0'_; C=:=0'-.

number_string(N) --> number_chars(Cs), { number_codes(N, Cs) }.
number_chars([C|Cs]) --> [C], { (C>=0'0, C=<0'9) ; C=:=0'. ; C=:=0'- }, !,
number_chars(Cs).
number_chars([]) --> [].

rules_to_horn_kb(ParsedRules, HornKB) :-
    maplist(rule_to_clause, ParsedRules, HornKB).

rule_to_clause(rule(Premises, Head), Clause) :-
    maplist(neg_lit, Premises, NegPremises),
    append(NegPremises, [Head], Clause).

neg_lit(P, n(P)).

read_answers(In, Values, BoolFacts) :-
    read_line_to_string(In, Line0),
    ( Line0 == end_of_file -> Values=[], BoolFacts=[]
    ; string_trim(Line0, Line),
      ( Line = "done" -> Values=[], BoolFacts=[]
      ; parse_answer_line(Line, Values1, BoolFacts1),
        read_answers(In, ValuesRest, BoolFactsRest),
        append(Values1, ValuesRest, Values),
        append(BoolFacts1, BoolFactsRest, BoolFacts)
      )
    ).

parse_answer_line(Line, Values, BoolFacts) :-
    ( sub_string(Line, 0, _, _, "ans:") ->
        sub_string(Line, 4, _, 0, Rest),
        ( sub_string(Rest, EqPos, _, After, "=") ->
            sub_string(Rest, 0, EqPos, _, IdStr0),
            sub_string(Rest, _, After, 0, ValStr0),
            string_trim(IdStr0, IdStr),
            string_trim(ValStr0, ValStr),
            atom_string(Id, IdStr),
            classify_value(Id, ValStr, Values, BoolFacts)
        ; throw(bad_answer(Line))
        )
    ; throw(bad_answer(Line))
    ).

classify_value(Id, ValStr, [value(Id, Num)], []) :-
    catch(number_string(Num, ValStr), _, fail), !.
classify_value(Id, ValStr, [], [has(Id)]) :-
    string_lower(ValStr, L),
    ( L = "yes" ; L = "true" ; L = "1" ), !.
classify_value(_Id, _ValStr, [], []) :-
    true.

derive_comparison_facts(ComparisonTerms, Values, Facts) :-
    findall(Cmp,
        ( member(Cmp, ComparisonTerms),
          comparison_holds(Cmp, Values)
        ),
        Facts0),
    sort(Facts0, Facts).

comparison_holds(gt(Id, K), Values) :-
    member(value(Id, V), Values),
    V > K.
comparison_holds(ge(Id, K), Values) :-
    member(value(Id, V), Values),
    V >= K.

forward_chaining(HornKB, Facts, Goal) :-
    closure(HornKB, Facts, Closure),
    member(Goal, Closure).

closure(HornKB, Facts, Closure) :-
    forward_step(HornKB, Facts, Facts1),
    ( Facts1 == Facts -> Closure = Facts
    ; closure(HornKB, Facts1, Closure)
    ).

forward_step(HornKB, Facts, FactsOut) :-
    findall(Head,
        ( member(Clause, HornKB),
          clause_head(Clause, Head),
          \+ member(Head, Facts),
          clause_premises(Clause, Premises),
          subset_list(Premises, Facts)
        ),
        NewHeads0),
    sort(NewHeads0, NewHeads),
    append(Facts, NewHeads, Facts1),
    sort(Facts1, FactsOut).

clause_head(Clause, Head) :- last(Clause, Head).

clause_premises(Clause, Premises) :-
    append(NegPremises, [_Head], Clause),
    maplist(unwrap_neg, NegPremises, Premises).

unwrap_neg(n(P), P).

subset_list([], _).
subset_list([X|Xs], Set) :- member(X, Set), subset_list(Xs, Set).

backward_chaining(HornKB, Facts, Goal) :-
    bc_prove(HornKB, Facts, Goal, []).

bc_prove(_KB, Facts, Goal, _Visited) :-
    member(Goal, Facts), !.
bc_prove(KB, Facts, Goal, Visited) :-
    \+ member(Goal, Visited),
    member(Clause, KB),
    clause_head(Clause, Goal),
    clause_premises(Clause, Premises),
    bc_all(KB, Facts, Premises, [Goal|Visited]).

bc_all(_KB, _Facts, [], _Visited).
bc_all(KB, Facts, [P|Ps], Visited) :-
    bc_prove(KB, Facts, P, Visited),
    bc_all(KB, Facts, Ps, Visited).

assert_kb_as_prolog(ParsedRules) :-
    retractall(rule_impl(_,_)),
    forall(member(rule(Ps,H), ParsedRules),
           assertz(rule_impl(H, Ps))).

assert_facts_as_prolog(Facts) :-
    retractall(fact(_)),
    forall(member(F, Facts),
           assertz(fact(F))).

prove_goal_builtin(Goal) :-
    prove(Goal).

prove(Goal) :-
    fact(Goal), !.
prove(Goal) :-
    rule_impl(Goal, Premises),
    prove_all(Premises).
```

```prolog
prove_all([]).
prove_all([P|Ps]) :-
    prove(P),
    prove_all(Ps).
```

Exercise2.pl

```prolog
:- use_module(library(socket)).
:- use_module(library(readutil)).
:- use_module(library(lists)).
:- use_module(library(dcg/basics)).

:- initialization(main, main).

main(Argv) :-
    ( Argv = [PortAtom, ScenarioPathAtom | _] ->
        atom_number(PortAtom, Port),
        atom_string(ScenarioPathAtom, ScenarioPath),
        run_client(Port, ScenarioPath)
    ; format(user_error, "Usage: swipl -s engine2.pl -- <PORT> <SCENARIO_FILE>~n",
[]),
      halt(2)
    ).

run_client(Port, ScenarioPath) :-
    tcp_socket(Sock),
    tcp_connect(Sock, '127.0.0.1':Port),
    tcp_open_socket(Sock, In, Out),
    handle_session(In, Out, ScenarioPath),
    close(In),
    close(Out).

handle_session(In, Out, ScenarioPath) :-
    read_scenario(ScenarioPath, RuleLines, GoalVar),
    maplist(parse_fuzzy_rule_line, RuleLines, Rules0),
    include(rule_matches_goal(GoalVar), Rules0, Rules),
    read_answers(In, Values),
    aggregate_output_curve(Rules, Values, Curve),
    defuzzify_centroid(Curve, CrispOut),
    format(Out, "result:~w=~2f~n", [GoalVar, CrispOut]),
    flush_output(Out).



rule_matches_goal(GoalVar, rule(_Conn, _Ante, GoalVar/_Pred)).

read_scenario(Path, RuleLines, GoalVar) :-
    read_file_to_string(Path, S, []),
    split_string(S, "\n", "\r", Lines0),
    maplist(string_trim, Lines0, Lines),
    extract_section_lines("Rules:", Lines, RuleLines0),
    include(is_rule_line, RuleLines0, RuleLines),
    extract_goal_var(Lines, GoalVar).

is_rule_line(Line) :-
    Line \= "",
    sub_string(Line, 0, 1, _, "-").

string_trim(S0, S) :-
    normalize_space(string(S), S0).

extract_section_lines(Header, Lines, SectionLines) :-
    ( append(_, [Header|Rest], Lines) ->
        take_until_next_header(Rest, SectionLines)
    ; SectionLines = []
    ).

take_until_next_header([], []).
take_until_next_header([L|_], []) :-
    is_header(L), !.
take_until_next_header([L|Ls], [L|Out]) :-
    \+ is_header(L),
    take_until_next_header(Ls, Out).

is_header("Rules:")     :- !.
is_header("Questions:") :- !.
is_header("The goal:")  :- !.

extract_goal_var(Lines, GoalVar) :-
    ( append(_, ["The goal:"|Rest], Lines) ->
        first_nonempty(Rest, GoalLine0),
        strip_leading_dash(GoalLine0, GoalLine),
        string_lower(GoalLine, GoalLower),
        atom_string(GoalVar, GoalLower)
    ; throw(no_goal_found)
    ).

first_nonempty([L|_], L) :- L \= "", !.
first_nonempty([_|Ls], L) :- first_nonempty(Ls, L).
first_nonempty([], "").

strip_leading_dash(S0, S) :-
    ( sub_string(S0, 0, 1, _, "-") ->
        sub_string(S0, 1, _, 0, S1),
        string_trim(S1, S)
    ; string_trim(S0, S)
    ).

read_answers(In, Values) :-
    read_line_to_string(In, Line0),
    ( Line0 == end_of_file ->
        Values = []
    ; string_trim(Line0, Line),
```

```prolog
      format(user_error, "[DBG] Read line: ~q~n", [Line]),
      ( is_done(Line) ->
          Values = []
      ; parse_answer_line_numeric(Line, Values1),
        read_answers(In, ValuesRest),
        append(Values1, ValuesRest, Values)
      )
    ).

is_done("done") :- !.
is_done(done)   :- !.

parse_answer_line_numeric(Line, Values) :-
    ( sub_string(Line, 0, _, _, "ans:") ->
        sub_string(Line, 4, _, 0, Rest),
        ( sub_string(Rest, EqPos, _, After, "=") ->
            sub_string(Rest, 0, EqPos, _, IdStr0),
            sub_string(Rest, _, After, 0, ValStr0),
            string_trim(IdStr0, IdStr),
            string_trim(ValStr0, ValStr),
            atom_string(Id, IdStr),
            ( catch(number_string(Num, ValStr), _, fail) ->
                Values = [value(Id, Num)]
            ;   Values = []
            )
        ; throw(bad_answer(Line))
        )
    ; throw(bad_answer(Line))
    ).

parse_fuzzy_rule_line(LineStr, Rule) :-
    string_codes(LineStr, Codes),
    phrase(fuzzy_rule(Rule), Codes).

fuzzy_rule(rule(Conn, Ante, Cons)) -->
    blanks, "-", blanks,
    ("If"; "if"), blanks,
    antecedent(Conn, Ante),
    blanks, ("then"; "Then"), blanks,
    consequent(Cons),
    blanks, ".", blanks.

antecedent(Conn, [L1, L2]) -->
    literal(L1),
    blanks,
    connector(Conn),
    blanks,
    literal(L2).
antecedent(and, [L]) -->
    literal(L).

connector(or)  --> ("or"; "OR").
connector(and) --> ("and"; "AND").

literal(Id/Pred) -->
    identifier(Id), blanks, ("is"; "IS"), blanks, identifier(Pred).

consequent(Id/Pred) --> literal(Id/Pred).

identifier(Atom) -->
    word_codes(Cs),
    { Cs \= [],
      string_codes(S, Cs),
      string_lower(S, SL),
      atom_string(Atom, SL)
    }.

word_codes([C|Cs]) -->
    [C],
    { code_type(C, alnum) ; C = 0'_; C = 0'- },
    !,
    word_codes_rest(Cs).
word_codes_rest([C|Cs]) -->
    [C],
    { code_type(C, alnum) ; C = 0'_; C = 0'- },
    !,
    word_codes_rest(Cs).
word_codes_rest([]) --> [].

clamp01(X, Y) :- Y is max(0.0, min(1.0, X)).

left_shoulder(A, B, X, Mu) :-
    ( X =< A -> Mu = 1.0
    ; X >= B -> Mu = 0.0
    ; Mu0 is (B - X) / (B - A), clamp01(Mu0, Mu)
    ).

right_shoulder(A, B, X, Mu) :-
    ( X =< A -> Mu = 0.0
    ; X >= B -> Mu = 1.0
    ; Mu0 is (X - A) / (B - A), clamp01(Mu0, Mu)
    ).

triangle(A, B, C, X, Mu) :-
    ( X =< A -> Mu = 0.0
    ; X >= C -> Mu = 0.0
    ; X =< B -> Mu0 is (X - A) / (B - A), clamp01(Mu0, Mu)
    ; Mu0 is (C - X) / (C - B), clamp01(Mu0, Mu)
    ).

mu(service, poor, X, Mu)      :- left_shoulder(0.0, 4.0, X, Mu).
mu(service, good, X, Mu)      :- triangle(2.0, 5.0, 8.0, X, Mu).
mu(service, excellent, X, Mu) :- right_shoulder(6.0, 10.0, X, Mu).
mu(food, rancid, X, Mu)       :- left_shoulder(0.0, 4.0, X, Mu).
mu(food, delicious, X, Mu)    :- right_shoulder(6.0, 10.0, X, Mu).

mu(tip, cheap, Y, Mu)         :- left_shoulder(0.0, 10.0, Y, Mu).
mu(tip, normal, Y, Mu)        :- triangle(5.0, 12.5, 20.0, Y, Mu).
mu(tip, generous, Y, Mu)      :- right_shoulder(15.0, 25.0, Y, Mu).

mu(cpu, slow, X, Mu)          :- left_shoulder(0.0, 4.0, X, Mu).
mu(cpu, normal, X, Mu)        :- triangle(2.0, 5.0, 8.0, X, Mu).
mu(cpu, fast, X, Mu)          :- right_shoulder(6.0, 10.0, X, Mu).

mu(ram, low, X, Mu)           :- left_shoulder(0.0, 4.0, X, Mu).
mu(ram, ok, X, Mu)            :- triangle(2.0, 5.0, 8.0, X, Mu).
mu(ram, high, X, Mu)          :- right_shoulder(6.0, 10.0, X, Mu).

mu(action, keep, Y, Mu)       :- left_shoulder(0.0, 10.0, Y, Mu).
mu(action, optimize, Y, Mu)   :- triangle(5.0, 12.5, 20.0, Y, Mu).
mu(action, upgrade, Y, Mu)    :- right_shoulder(15.0, 25.0, Y, Mu).

get_value(Id, Values, V) :- member(value(Id, V), Values).

lit_degree(Id/Pred, Values, Deg) :-
    get_value(Id, Values, X),
    mu(Id, Pred, X, Deg).

combine_degrees(or, Degs, Out)  :- max_list(Degs, Out).
combine_degrees(and, Degs, Out) :- min_list(Degs, Out).

rule_applicability(rule(Conn, Lits, _Cons), Values, Alpha) :-
    maplist({Values}/[Lit,Deg]>>lit_degree(Lit, Values, Deg), Lits, Degs),
    combine_degrees(Conn, Degs, Alpha).

output_domain(Domain) :- findall(Y, between(0,25,Y), Domain).

rule_contrib_at_y(rule(_Conn, _Lits, OutVar/OutPred), Alpha, Y, MuY) :-
    mu(OutVar, OutPred, Y, BaseMu),
    MuY is min(Alpha, BaseMu).

aggregate_output_curve(Rules, Values, Curve) :-
    output_domain(Domain),
    findall(alpha_rule(R,Alpha),
        ( member(R, Rules),
          rule_applicability(R, Values, Alpha)
        ),
        Alphas),
    findall(Y-MuAgg,
        ( member(Y, Domain),
          findall(MuY,
              ( member(alpha_rule(R,Alpha), Alphas),
                rule_contrib_at_y(R, Alpha, Y, MuY)
              ),
              Mus),
          ( Mus = [] -> MuAgg = 0.0 ; max_list(Mus, MuAgg) )
        ),
        Curve).

defuzzify_centroid(Curve, Crisp) :-
    foldl([_Y-Mu, S0, S1]>>(S1 is S0 + Mu), Curve, 0.0, SumMu),
    ( SumMu =:= 0.0 ->
        Crisp = 0.0
    ; foldl([Y-Mu, A0, A1]>>(A1 is A0 + Y*Mu), Curve, 0.0, SumYMu),
      Crisp is SumYMu / SumMu
    ).
```