Expert System for Degree Program Selection (Prolog)

An **expert system** uses a *knowledge base* (facts and rules) plus an inference engine to draw conclusions. In our case, the domain knowledge (Table 1) links a student's AL stream and interest to a degree program. We encode each table row as a Prolog fact or rule. Prolog is well-suited for this: it inherently acts as an inference engine that unifies facts and backtracks through rules to answer queries[1]. In Prolog, the knowledge base consists of **clauses** (facts and rules)[2]. A **fact** is a clause with an empty body (no conditions)[3], whereas a **rule** has the form `Head :- Body`, meaning "Head is true if Body is true." In general, rules say *if the body is true, then the head is true*[4]. For example, a rule `playsAirGuitar(X) :- listens2Music(X).` means "X plays air guitar if X listens to music." Prolog uses *unification* (pattern matching) and *backtracking* to satisfy queries[5]. When we ask a query (e.g. `?- degree_program(math, scientist, D).`), Prolog attempts to match it against the facts/rules and returns a result if possible.

Knowledge Representation in Prolog

We represent each entry of Table 1 as a Prolog predicate, e.g. `degree_program(ALStream, Field, Degree)`. The facts below encode the sample knowledge: each fact links an AL stream and interest to a degree. For instance, the fact

```
degree_program(math, scientist, ai).
```

states that a student in the Math stream interested in being a Scientist should take an AI degree. In Prolog, facts follow the format `relation(entity1, entity2, …).` and form the knowledge base[6]. For our table, we can write:

```
degree_program(math, scientist, ai).
degree_program(math, computer_science, engineer).
degree_program(hardware, math, computer_engineer).
degree_program(software, any, software_developer).
degree_program(software, quality, industry).
```

Each of these lines is a fact (a clause with no conditions). We use lowercase atoms (`math`, `scientist`, `ai`, etc.) to match Prolog syntax. The set of all these facts is the **knowledge base** for our expert system. (In Prolog, a fact can be viewed as a rule with an empty body[3].) Once defined, we can query this knowledge base. For example, the query `?- degree_program(math, scientist, D).` succeeds with `D = ai`.

Inference and Queries

Prolog's inference engine will use the facts above to answer questions. When we pose a query like `degree_program(AL, Field, Degree)`, Prolog searches through the facts, unifies variables with matching values, and returns all solutions. This is a form of backward chaining: Prolog looks for facts/rules that can satisfy the query, and if it hits a rule, it ensures the rule's body is true (recursively applying modus ponens)[4]. In simple cases with only facts, Prolog just matches a fact. For instance, if the user inputs `AL = software` and `Field = any`, Prolog finds the fact `degree_program(software, any, software_developer)` and infers `Degree = software_developer`. Because facts and rules in Prolog are called clauses[2], the system can chain together inferences: if we had rules that combined conditions, Prolog would combine them until it deduces a matching head.

Prolog User Interface (I/O)

To interact with the user, we use Prolog's built-in I/O predicates. The `write/1` predicate displays text, and `read/1` reads a term from the console[7]. For example, a predicate to ask the user for inputs might be:

```
recommend_degree :-
    write('Enter AL stream (math/hardware/software): '), nl,
    read(AL),
    write('Enter field of interest: '), nl,
    read(Field),
    ( degree_program(AL, Field, Degree) ->
        write('Recommended degree program: '), write(Degree), nl
    ;   write('No matching degree found.'), nl ).
```

This uses `write` to prompt and `read` to capture the user's answer, then queries the `degree_program/3` facts. If a matching fact is found, it writes out the `Degree`. The tutorial example *cube* program illustrates a similar pattern: it uses `write('Write a number: ')` and `read(Number)` to get user input[7]. In practice, Prolog will unify the input atom (e.g. `math`) with the facts. Because Prolog backtracks on failure, we added an if-then (`-> ;`) to handle the case where no fact matches.

Key Prolog Concepts (Theory)

- **Facts and Rules:** Our knowledge base is made of *facts* (e.g. `degree_program(...)`) and possibly *rules*. A rule has the form `Head :- Body` (read "Head if Body"). If the *Body* (conditions) can be proven true from known facts/rules, then Prolog infers the *Head*[4]. A fact is a rule with an empty body[3].

- **Unification:** Prolog answers queries by unification: it tries to match the query against facts/rules. Variables in the query can match different constant symbols (e.g. `Degree = ai`). This is pattern matching at the core of Prolog reasoning[5].

- **Backtracking:** If one fact/rule does not satisfy the query, Prolog backtracks and tries another. This allows Prolog to find all possible answers or report failure[5].

- **Knowledge Base (KB):** The KB stores domain-specific facts and rules. In an expert system, the KB holds the expertise. Prolog itself serves as the *inference engine*, applying rules like modus ponens to derive conclusions[1][4].

- **User Interaction:** We typically write a *driver* predicate (here, `recommend_degree/0`) that asks the user questions with `write/1` and reads answers with `read/1`, then queries the KB. For example, `write('Enter field: ')` followed by `read(Field)` gets input into the variable `Field`[7]. After reading answers, we query our facts and display the result with `write/1`.

By combining these elements—facts for Table 1, Prolog's inference, and simple I/O—we implement a working expert system. The rules above capture the given knowledge base, and the UI predicates prompt the student and print out the recommended degree.

Example Code Implementation:

```
% Knowledge base facts (from Table 1)
degree_program(math, scientist, ai).
degree_program(math, computer_science, engineer).
degree_program(hardware, math, computer_engineer).
degree_program(software, any, software_developer).
degree_program(software, quality, industry).

% User interface predicate
recommend_degree :-
    write('Enter AL stream (math/hardware/software): '), nl,
    read(AL),
    write('Enter field of interest: '), nl,
    read(Field),
    ( degree_program(AL, Field, Degree)
    -> write('Recommended degree program: '), write(Degree), nl
    ;  write('No matching degree found.'), nl
    ).
```

When run (e.g. with SWI-Prolog), this will ask the student for their AL stream and interest, then use the facts to find and display the appropriate degree.

Sources: Prolog treats facts as a database of knowledge and uses backward-chaining to infer answers[6][4]. The `write/1` and `read/1` predicates handle console I/O[7]. Prolog's built-in search/backtracking makes it an easy way to build simple expert systems[1][5].

---

[1] Expert Systems in Prolog

https://www.metalevel.at/prolog/expertsystems

[2] [3] [4] Learn Prolog Now!

https://lpn.swi-prolog.org/lpnpage.php?pagetype=html&pageid=lpn-htmlse1

[5] [6] Prolog | An Introduction - GeeksforGeeks

https://www.geeksforgeeks.org/artificial-intelligence/prolog-an-introduction/

Logical Equivalences

Domination laws:	$p \vee \mathbf{T} \equiv \mathbf{T}$, $p \wedge \mathbf{F} \equiv \mathbf{F}$
Identity laws:	$p \wedge \mathbf{T} \equiv p$, $p \vee \mathbf{F} \equiv p$
Idempotent laws:	$p \wedge p \equiv p$, $p \vee p \equiv p$
Double negation law:	$\neg(\neg p) \equiv p$
Negation laws:	$p \vee \neg p \equiv \mathbf{T}$, $p \wedge \neg p \equiv \mathbf{F}$

Commutative laws:	$p \wedge q \equiv q \wedge p$, $p \vee q \equiv q \vee p$
Associative laws:	$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$
	$(p \vee q) \vee r \equiv p \vee (q \vee r)$
Distributive laws:	$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$
	$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$
Absorption laws:	$p \vee (p \wedge q) \equiv p$, $p \wedge (p \vee q) \equiv p$

Logical Equivalences cont

- $\neg(\neg P) \equiv P$
- $P \vee Q \equiv Q \vee P$
- $P \rightarrow Q \equiv \neg P \vee Q$
- $\neg(P \vee Q) \equiv \neg P \wedge \neg Q$
- $\neg(P \wedge Q) \equiv \neg P \vee \neg Q$
- $P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$
- $P \rightarrow Q \equiv \neg Q \rightarrow \neg P$