Prolog Knowledge Base Documentation: Education and Tourism Domains

Giacomo Signorile and Marco Angelo Lillo

May 30, 2025

Abstract

This document describes two Prolog knowledge bases. The first models entities, relationships, and rules relevant to an educational domain developed by Marco Angelo Lillo, based on the provided education.gbs ontology. The second models a tourism domain, including points of interest, transportation, services, and related rules, developed by Giacomo Signorile. Both utilize a common framework for schema definition, fact representation, and rule-based reasoning. The structure of the knowledge bases was generated by the class `SchemaToProlog.java` and then implemented with facts and rules. Each knowledge base includes its specific schema, factual data, generic helper predicates, and a set of domain-specific rules for deriving insights.

1 Educational Knowledge Base (from education.gbs)

The primary motivation is to create a robust system capable of representing a rich educational domain and performing complex inferences. This involves defining a clear schema for educational concepts, populating the KB with instance data, and implementing a set of rules that derive new knowledge or suggest actions. The contribution of this work lies in the comprehensive design of the educational schema, the development of generic schema-handling utilities, and the implementation of diverse reasoning rules, including those employing search techniques and dynamic knowledge updates.

1.1 System Architecture and Methodology

The knowledge base is constructed in Prolog and follows a layered architecture:

1. **Schema Definition Layer:** This foundational layer explicitly defines the types of entities, their attributes, the relationships between them, and the class hierarchy. This includes:
    - Entity types (e.g., `User`, `Container`, `Skill`) and their direct attributes using facts like `'User'([name])`.
    - A subclass hierarchy (e.g., `subclass_of('Course', 'Container')`).
    - Relationship types (e.g., `partOf`, `hasSkill`), their domain-range constraints, and attributes using facts like `'hasSkill'(['User'-'Category'], [level])`.
    - Inverse relationships (e.g., `inverse_of(partOf, hasPart)`).
2. **Generic Schema Utilities Layer:**
    - `is_subclass/2`: Determines subclass relationships transitively.
    - `schema_entity_definition/2` and `schema_relationship_definition/3`: Retrieve schema definitions using a dispatcher pattern (e.g., `defined_entity_schema/2`) for robustness.
    - `gather_attributes/2`: Collects all attributes for an entity, including inherited ones.
    - `invert_relationship/2`: Derives the schema of an inverse relationship.
3. **Instance Data Layer:** Populates the KB with specific instances of entities and relationships.
    - `entity(ID, Type, AttributeList)` facts define individual entities.
    - `rel(RelName, SubjectID, ObjectID, AttributeList)` facts define specific relationships.
    - Helper `xxx_info/2` predicates provide human-readable names for IDs.
4. **Rule-Based Reasoning Layer:** Contains rules that infer new information, check conditions, or suggest actions.

Discontiguous predicate declarations are used. Dynamic predicates (e.g., `strong_project_candidate/3`) allow the KB to be updated. The `content_covers_skill/2` predicate is tabled for performance.

1.2 Knowledge Representation Details

1.2.1 Entities (entity/3)

Entities include `User`, `Container` (e.g., `Course`, `Module`), `AssessmentTool`, `Category` (e.g., `Skill`, `Topic`), `Event`, `Accomplishment`, and `Project`. Schemas defining direct attributes are exemplified by `'Container'([name, language, difficulty])`.

1.2.2 Relationships (rel/4)

Key relationships include `partOf`, `hasSkill`, `requires`, etc., connecting entities. Inverse relationships like `inverse_of(partOf, hasPart)` enable bidirectional querying. Relationship signatures define domains, ranges, and attributes, e.g., `'hasSkill'(['User'-'Category'], [level])`.

1.2.3 Instance Data

Specific instances are defined using facts such as `entity(u1, 'User', [attr(name, 'Alice')])` and `rel(partOf, mod_prolog, cos101, [])`. Attributes are represented as `attr(AttributeName, Value)` pairs.

1.3 Rule-Based Reasoning Engine

These rules derive new facts, check complex conditions, and enable sophisticated querying capabilities. Each rule is designed to address a specific aspect of the educational domain.

These foundational rules facilitate the interaction with the knowledge base structure and form the basis for more complex reasoning.

1.3.1 Get attribute from entity(`get_attribute_value/3`)

It queries `entity/3` facts and searches for an `attr(AttrName, Value)` term. If not found, ‘Value‘ is unified with ‘default‘.

```
get_attribute_value(EntityID, AttrName, Value) :-
entity(EntityID, _, Attributes),
member(attr(AttrName, Value), Attributes).
get_attribute_value(EntityID, AttrName, default) :-
entity(EntityID, _, _),
\+ member(attr(AttrName, _), Attributes).
```

1.3.2 Checks if Parent Class of Component is Container (`transitively_part_of/2`)

This predicate determines if a `Component` is part of a `Container`, directly or indirectly, by recursively checking `partOf` relationships.

```
transitively_part_of(Component, Container) :-
rel(partOf, Component, Container, _).
transitively_part_of(Component, Container) :-
rel(partOf, Component, Intermediate, _),
transitively_part_of(Intermediate, Container).
```

1.3.3 Validation of a learning `ContainerID` (`content_covers_skill/2`)

A tabled predicate that figures out if a learning `ContainerID` addresses a specific `SkillID`. It achieves this through several checks, primarily delegated to a helper predicate `ccs_check/3`:

- **CCS1 (Direct Assessment):** The container directly assesses the skill: `rel(assessesSkill, ContainerID, SkillID, _)`.

- **CCS2 (Assessment Part):** A part of the `ContainerID` which is itself an `AssessmentContainer` assesses the `SkillID`.
- **CCS3 (Lesson Inheritance):** If the `ContainerID` is a `Lesson`, it inherits skill coverage from its parent `Module`.
- **CCS4 (Recursive Part Coverage):** Recursively, if any sub-part of the `ContainerID` (provided it's not a `Lesson`) covers the `SkillID`.

Full rule definition:

```
content_covers_skill(ContainerID, SkillID) :-
entity(ContainerID, ContainerType, _),
is_subclass(ContainerType, 'Container'),
entity(SkillID, SkillCatType, _),
is_subclass(SkillCatType, 'Category'),
ccs_check(ContainerID, SkillID, ContainerType).

ccs_check(ContainerID, SkillID, _ContainerType) :-
rel(assessesSkill, ContainerID, SkillID, _).
ccs_check(ContainerID, SkillID, _ContainerType) :-
rel(partOf, AssessmentPartID, ContainerID, _),
entity(AssessmentPartID, PartType, _),
is_subclass(PartType, 'AssessmentContainer'),
rel(assessesSkill, AssessmentPartID, SkillID, _).
ccs_check(ContainerID, SkillID, 'Lesson') :-
rel(partOf, ContainerID, ParentModuleID, _),
entity(ParentModuleID, 'Module', _),
content_covers_skill(ParentModuleID, SkillID).
ccs_check(ContainerID, SkillID, ContainerType) :-
ContainerType \= 'Lesson',
transitively_part_of(SubPartID, ContainerID),
SubPartID \== ContainerID,
content_covers_skill(SubPartID, SkillID).
```

1.3.4 Mastered Containers verification (`learner_mastered_container/2`)

This rule determines if a `UserID` has "mastered" a `ContainerID`. It succeeds if the user owns a valid `Accomplishment` that was awarded for that specific container.

```
learner_mastered_container(UserID, ContainerID) :-
entity(UserID, 'User', _),
entity(ContainerID, ContainerType, _),
is_subclass(ContainerType, 'Container'),
rel(ownsAccomplishment, UserID, AccID, Attributes),
member(attr(isValid, true), Attributes),
entity(AccID, 'Accomplishment', _),
rel(awardedFor, AccID, ContainerID, _).
```

1.3.5 Learner verification (`is_learner/1`)

Identifies if a ‘User‘ is a learner by checking if they are `involvedInEvent` with the role 'Learner'.

```
is_learner(User) :-
entity(User, 'User', _),
rel(involvedInEvent, User, _Event, Attributes),
member(attr(role, 'Learner'), Attributes).
```

1.3.6 Teacher verification (`is_teacher/1`)

Identifies if a ‘User‘ acts as an educator by checking if they are `involvedInEvent` with the role 'Teacher' or 'Speaker'.

```
is_teacher(User) :-
entity(User, 'User', _),
rel(involvedInEvent, User, _Event, Attributes),
( member(attr(role, 'Teacher'), Attributes) ;
  member(attr(role, 'Speaker'), Attributes) ).
```

1.3.7 Developed relation verification

`is_content_developer/1` Determines if a ‘User‘ has developed any ‘Container‘ or ‘Assessment-Tool‘.

```
is_content_developer(User) :-
entity(User, 'User', _),
rel(developed, User, DevelopedThing, _),
( entity(DevelopedThing, Type, _), is_subclass(Type, 'Container') ;
  entity(DevelopedThing, Type, _), is_subclass(Type, 'AssessmentTool') ).
```

1.3.8 Check of User’s skill level (`user_has_skill_level/3`)

Retrieves the proficiency ‘Level‘ a ‘User‘ has for a given ‘Skill‘ from the `hasSkill` relationship.

```
user_has_skill_level(User, Skill, Level) :-
entity(User, 'User', _),
entity(Skill, SkillType, _), is_subclass(SkillType, 'Skill'),
rel(hasSkill, User, Skill, Attributes),
member(attr(level, Level), Attributes).
```

1.3.9 Lack of skill verification (`user_lacks_skill_for_level/3`)

Checks if a ‘User‘’s skill level for a ‘Skill‘ is below a ‘RequiredLevel‘, or if the user does not possess the skill.

```
user_lacks_skill_for_level(User, Skill, RequiredLevel) :-
entity(User, 'User', _),
entity(Skill, SkillType, _), is_subclass(SkillType, 'Skill'),
(   user_has_skill_level(User, Skill, CurrentLevel) ->
    CurrentLevel < RequiredLevel
;   RequiredLevel > 0,
    \+ rel(hasSkill, User, Skill, _) ).
```

1.3.10 Check of Project’s skill required (`project_requires_skill/2`)

Identifies the ‘SkillID‘ designated as a `requiredSkill` for a ‘ProjectID‘.

```
project_requires_skill(ProjectID, SkillID) :-
entity(ProjectID, 'Project', Attributes),
member(attr(requiredSkill, SkillID), Attributes),
entity(SkillID, SkillType, _), is_subclass(SkillType, 'Skill').
```

1.3.11 Requirements verification for a User (`user_meets_project_skill_requirement/3`)

Determines if a ‘User‘’s skill level for a project’s required skill meets or exceeds a ‘MinLevel‘.

```
user_meets_project_skill_requirement(User, Project, MinLevel) :-
project_requires_skill(Project, SkillID),
user_has_skill_level(User, SkillID, UserLevel),
UserLevel >= MinLevel.
```

1.3.12 Predicting possible interests in Users (`predict_project_interest_from_skills/2`)

An **ILP-like** hypothesis suggesting ‘User‘ interest in a ‘Project‘ if they mastered content covering a skill required by the project, and are not already listed as interested.

```
predict_project_interest_from_skills(User, Project) :-
entity(User, 'User', _),
entity(Project, 'Project', _),
learner_mastered_container(User, Container),
content_covers_skill(Container, Skill),
project_requires_skill(Project, Skill),
\+ rel(interestedInProject, User, Project, _).
```

1.3.13 Container recommendation for User (suggest_learning_material_for_skill_gap/3)

Recommends a ‘Container‘ to a ‘User‘ for ‘SkillToImprove‘ if they lack proficiency, the container covers it, and they haven’t mastered it.

```
suggest_learning_material_for_skill_gap(User, Container, SkillToImprove) :-
entity(User, 'User', _),
entity(SkillToImprove, SkillType, _), is_subclass(SkillType, 'Skill'),
user_lacks_skill_for_level(User, SkillToImprove, 5),
content_covers_skill(Container, SkillToImprove),
entity(Container, CType, _), is_subclass(CType, 'Container'),
\+ learner_mastered_container(User, Container).
```

1.3.14 Find candidate team (find_project_team_candidates/2)

Collects a ‘UserList‘ of all users explicitly interested in a given ‘Project‘.

```
find_project_team_candidates(Project, UserList) :-
entity(Project, 'Project', _),
findall(User, rel(interestedInProject, User, Project, _), UserList).
```

1.3.15 User Participation in Event (user_attended_container_event/2)

Checks if a ‘User‘ participated in any ‘Event‘ that is an instance of the specified ‘Container‘.

```
user_attended_container_event(User, Container) :-
entity(User, 'User', _),
entity(Container, CType, _), is_subclass(CType, 'Container'),
rel(instanceOf, Event, Container, _),
rel(involvedInEvent, User, Event, _).
```

1.3.16 Instructor validation for Container (is_instructor_for_container/2)

Identifies an ‘Instructor‘ for a ‘Container‘ if they taught an event instance of it or developed it.

```
is_instructor_for_container(Instructor, Container) :-
entity(Instructor, 'User', _),
entity(Container, CType, _), is_subclass(CType, 'Container'),
(   rel(instanceOf, Event, Container, _),
    rel(involvedInEvent, Instructor, Event, Attributes),
    member(attr(role, 'Teacher'), Attributes)
;   rel(developed, Instructor, Container, _) ).
```

1.3.17 Gathers all distinct sub-components of a ‘TopContainer‘ using transitively_part_of/2 (get_recursive_container_parts/2)

```
get_recursive_container_parts(TopContainer, PartList) :-
entity(TopContainer, TopCType, _), is_subclass(TopCType, 'Container'),
findall(Part, (transitively_part_of(Part, TopContainer),
                Part \== TopContainer), DupPartList),
list_to_set(DupPartList, PartList).
```

1.3.18 Accomplishment validation for User (is_user_accomplishment_active/2)

Checks if a specific ‘AccomplishmentID‘ owned by a ‘User‘ has its isValid attribute set to true.

```
is_user_accomplishment_active(User, AccomplishmentID) :-
entity(User, 'User', _),
entity(AccomplishmentID, 'Accomplishment', _),
rel(ownsAccomplishment, User, AccomplishmentID, Attributes),
member(attr(isValid, true), Attributes).
```

1.3.19 Checks expertise of a User (`is_skill_expert/3`)

Determines if a ‘User‘ is an expert in a ‘Skill‘ if their skill level meets or exceeds an ‘Expert-Threshold‘.

```
is_skill_expert(User, Skill, ExpertThreshold) :-
user_has_skill_level(User, Skill, UserLevel),
UserLevel >= ExpertThreshold.
```

1.3.20 Tool validation (`is_tool_current_version/1`)

Determines if an ‘AssessmentTool‘ (`ToolID`) is the latest version by checking it does not `evolvesTo` another tool.

```
is_tool_current_version(ToolID) :-
entity(ToolID, 'AssessmentTool', _),
\+ rel(evolvesTo, ToolID, _AnotherTool, _).
```

1.3.21 DifficultyAtom grouping(`find_content_by_difficulty_rating/2`)

Collects a list of ‘Container‘s that have a specific ‘difficulty‘ attribute.

```
find_content_by_difficulty_rating(DifficultyAtom, ContainerList) :-
atom(DifficultyAtom),
findall(ContainerID,
        (   entity(ContainerID, CType, Attributes),
            is_subclass(CType, 'Container'),
            member(attr(difficulty, DifficultyAtom), Attributes) ),
        ContainerList).
```

1.3.22 Prerequisites grouping (`get_immediate_prerequisites_for_container/2`)

Retrieves a list of all direct prerequisites for a given ‘Container‘.

```
get_immediate_prerequisites_for_container(Container, PrerequisiteContainers) :-
entity(Container, CType, _), is_subclass(CType, 'Container'),
findall(Prereq, rel(requires, Container, Prereq, _), PrerequisiteContainers).
```

1.3.23 Prerequisites getter for a Container (`get_all_prerequisites_for_container/2`)

Finds all direct and indirect container prerequisites using `collect_all_prereqs_recursive/3`, which avoids cycles.

```
get_all_prerequisites_for_container(Container, AllPrerequisiteContainers) :-
entity(Container, CType, _), is_subclass(CType, 'Container'),
findall(P, collect_all_prereqs_recursive(Container, P, [Container]),
        PrereqsListDups),
list_to_set(PrereqsListDups, AllPrerequisiteContainers).
% Helper: collect_all_prereqs_recursive/3 (see full Prolog code for its definition)
```

1.3.24 Expert Users identification (`find_expert_content_creator_for_skill/3`)

Identifies users expert in a ‘Skill‘ who also developed ‘CreatedContent‘ (Container/Assessment-Tool) covering that same ‘Skill‘.

```
find_expert_content_creator_for_skill(User, Skill, CreatedContent) :-
is_skill_expert(User, Skill, 8),
rel(developed, User, CreatedContent, _),
( entity(CreatedContent, CCType, _), is_subclass(CCType, 'Container') ;
  entity(CreatedContent, CCType, _), is_subclass(CCType, 'AssessmentTool') ),
content_covers_skill(CreatedContent, Skill).
```

1.3.25 Assertion of User interested in Project (identify_and_record_strong_candidates/0)

If a user is interested in a project and has its required skill, and is not already marked, this rule asserts a strong_project_candidate/3 fact.

```
identify_and_record_strong_candidates :-
rel(interestedInProject, User, Project, _),
project_requires_skill(Project, RequiredSkill),
rel(hasSkill, User, RequiredSkill, _),
(   \+ strong_project_candidate(User, Project, RequiredSkill) ->
    assertz(strong_project_candidate(User, Project, RequiredSkill)),
    user_info(User, UserName),
    project_info(Project, ProjectName),
    category_info(RequiredSkill, SkillName),
    format('INFO: ~w (~w) is now a strong candidate for project ~w (~w) requiring skill
    ~w (~w).~n',
           [UserName, User, ProjectName, Project, SkillName, RequiredSkill])
;   true ).
```

1.3.26 User stuck identification (is_learner_potentially_stuck/3)

Identifies a 'Learner' active in InContainer but has not mastered one of its immediate PrereqContainer prerequisites.

```
is_learner_potentially_stuck(Learner, InContainer,
                             missing_prerequisite(PrereqContainer)) :-
entity(Learner, 'User', _),
entity(InContainer, ICType, _), is_subclass(ICType, 'Container'),
rel(instanceOf, EventID, InContainer, _),
rel(involvedInEvent, Learner, EventID, EventAttrs),
member(attr(role, 'Learner'), EventAttrs),
rel(requires, InContainer, PrereqContainer, _),
entity(PrereqContainer, PCType, _), is_subclass(PCType, 'Container'),
\+ learner_mastered_container(Learner, PrereqContainer).
```

1.3.27 Possible Collaboration for a Container between Users (find_peer_for_container_activity/3)

Suggests a 'Peer' for a 'User' if both are 'Learner's in the same event instance of a 'Container'.

```
find_peer_for_container_activity(User, Container, Peer) :-
entity(User, 'User', _),
entity(Container, CType, _), is_subclass(CType, 'Container'),
entity(Peer, 'User', _),
User \== Peer,
rel(instanceOf, EventID, Container, _),
rel(involvedInEvent, User, EventID, UserEventAttrs),
member(attr(role, 'Learner'), UserEventAttrs),
rel(involvedInEvent, Peer, EventID, PeerEventAttrs),
member(attr(role, 'Learner'), PeerEventAttrs).
```

1.3.28 Average rating for a Container (container_average_rating/3)

Computes 'AverageRating' and 'NumRatings' for a 'ContainerID' using aggregate_all/3.

```
container_average_rating(ContainerID, AverageRating, NumRatings) :-
entity(ContainerID, CType, _), is_subclass(CType, 'Container'),
aggregate_all(
    count + sum(RatingValue),
    (   rel(rated, _User, ContainerID, RateAttrs),
        member(attr(ratingValue, RatingValue), RateAttrs) ),
    NumRatings + SumOfRatings ),
(NumRatings > 0 -> AverageRating is SumOfRatings / NumRatings ;
                  AverageRating = 'not_rated', NumRatings = 0).
```

1.3.29 Reccomendation of a Container for a skill (`suggest_direct_next_step_for_skill/3`)

Recommends a ‘SuggestedContainer‘ to ‘User‘ for ‘TargetSkill‘ if it covers the skill, user hasn’t mastered it, and all its container prerequisites are mastered by the user (checked via `forall/2`).

```
suggest_direct_next_step_for_skill(User, TargetSkill, SuggestedContainer) :-
entity(User, 'User', _),
entity(TargetSkill, SKType, _), is_subclass(SKType, 'Skill'),
entity(SuggestedContainer, SCType, _), is_subclass(SCType, 'Container'),
content_covers_skill(SuggestedContainer, TargetSkill),
\+ learner_mastered_container(User, SuggestedContainer),
forall(
    (   rel(requires, SuggestedContainer, Prereq, _),
        entity(Prereq, PrereqEntityClass, _),
        is_subclass(PrereqEntityClass, 'Container') ),
    learner_mastered_container(User, Prereq) ).
```

1.4 Implementation Highlights and Demonstration

Key Prolog features utilized include:

- List processing for managing attributes and schema components.
- Unification for pattern matching and data retrieval.
- Meta-predicates like `findall/3`, `setof/3`, `maplist/2`, `aggregate_all/3`, and `forall/2`.
- Dynamic predicates (`assertz/1`) for evolving the KB.
- Tabling (`:- table/1`) for optimizing recursive queries like `content_covers_skill/2`.

Demonstration queries:

1. To find all unique learners:

```
?- setof(User-Name, (is_learner(User), user_info(User, Name)), Learners).
```

Listing 1: Query for Unique Learners

2. To get all prerequisites for the ’Advanced Prolog Workshop’:

```
?- get_all_prerequisites_for_container(workshop_advanced_prolog, AllPrereqs).
```

Listing 2: Query for All Prerequisites

3. To suggest the next learning step:

```
?- suggest_direct_next_step_for_skill(u5, skill_expert_prolog, S),
   container_info(S,N).
```

Listing 3: Query for Learning Suggestion

4. To identify and record strong project candidates:

```
?- findall(_, identify_and_record_strong_candidates, _).
```

Listing 4: Query to Record Strong Candidates

Followed by:

```
?- listing(strong_project_candidate).
```

Listing 5: Listing Strong Candidates

5. To calculate the average rating for the 'Prolog Module':

```
?- container_average_rating(mod_prolog, AvgRating, NumRatings).
```

Listing 6: Query for Average Rating

1.5 Query Execution Examples and Outputs

This section demonstrates the execution of selected queries and explains their outputs, particularly highlighting rules with side effects..

1.5.1 Example 1: Identifying and Recording Strong Project Candidates

The rule `identify_and_record_strong_candidates/0` (Rule 21) dynamically updates the knowledge base by asserting `strong_project_candidate/3` facts. The `assertz/1` predicate adds a new clause to the end of the specified dynamic predicate. This rule checks for users interested in projects who also possess the required skill for that project. If such a candidate is found and not already recorded, the rule asserts this information and prints an informational message.

Query:

```
?- findall(_, identify_and_record_strong_candidates, _).
```

Listing 7: Executing Candidate Identification

The `findall/3` predicate is used here to ensure that the `identify_and_record_strong_candidates/0` goal is executed for all possible instantiations derived from the knowledge base, thus processing all potential candidates. The first argument `_` means we are not interested in collecting a list of specific terms from each success, and the third argument `_` means we are not binding the final list of these terms. The primary purpose here is to trigger the rule's side effects (the `assertz/1` and `format/2` calls).

Output:

```
INFO: Charlie_Learner (u3) is now a strong candidate for project AI Chatbot (p1)
      requiring skill Prolog Programming (skill_prolog).
INFO: Eve_Student (u5) is now a strong candidate for project AI Chatbot (p1)
      requiring skill Prolog Programming (skill_prolog).
true.
```

Explanation:

- The 'INFO' messages are generated by the `format/2` call within `identify_and_record_strong_candidates/0` each time a new `strong_project_candidate/3` fact is successfully asserted.
- Based on the instance data, `u3` (Charlie_Learner) is interested in `p1` ('AI Chatbot'), `p1` requires `skill_prolog`, and `u3` possesses `skill_prolog`. Thus, `u3` is identified and recorded.
- Similarly, `u5` (Eve_Student) is interested in `p1`, which requires `skill_prolog`, and `u5` also possesses `skill_prolog`. Thus, `u5` is also identified and recorded.
- After this query, the knowledge base will contain new facts:

```
strong_project_candidate(u3, p1, skill_prolog).
strong_project_candidate(u5, p1, skill_prolog).
```

Listing 8: Asserted Facts after Execution

These can be verified by querying `listing(strong_project_candidate)`.

1.5.2 Example 2: Suggesting Next Learning Step

The rule `suggest_direct_next_step_for_skill/3` (Rule 25) recommends a container for a user to develop a target skill, considering prerequisites.

Query:

```
?- setof(Name-Suggested,
        (suggest_direct_next_step_for_skill(u5, skill_expert_prolog, Suggested),
         container_info(Suggested, Name)),
        UniqueSuggestions).
```

Listing 9: Suggesting Next step for Eve

Output (based on instance data, including `u5` mastering `mod_prolog`):

```
UniqueSuggestions = ['AI Course'-cos101, 'Advanced Prolog Workshop'-
workshop_advanced_prolog]
```

Explanation of Output:

- The query asks for a `Suggested` container and its `Name` for user `u5` (Eve_Student) to learn `skill_expert_prolog`.
- `cos101` ('AI Course') is suggested because:
  - It `content_covers_skill` for `skill_expert_prolog` (indirectly, as `workshop_advanced_prolog` is part of it and covers the skill).
  - `u5` has not mastered `cos101`.
  - The prerequisite for `cos101` is `mod_prolog`, and the instance data shows `u5` has mastered `mod_prolog` (via `ac_basic_mod_prolog`).
- `workshop_advanced_prolog` is suggested because:
  - It `content_covers_skill` for `skill_expert_prolog` (directly via `assessesSkill`).
  - `u5` has not mastered `workshop_advanced_prolog`.
  - Its prerequisite `mod_prolog` is mastered by `u5`.

1.5.3 Example 3: Calculating Average Container Rating

The rule `container_average_rating/3` (Rule 24) uses `aggregate_all/3` for calculations.

Query:

```
?- container_average_rating(mod_prolog, AvgRating, NumRatings).
```

Listing 10: Average Rating for Prolog Module

Output (based on instance data):

```
AvgRating = 4.5,
NumRatings = 2.
```

Explanation of Output:

- The query asks for the average rating and number of ratings for `mod_prolog`.
- The instance data contains:

  - `rel(rated, u3, mod_prolog, [attr(ratingValue, 5)]).`
  - `rel(rated, u5, mod_prolog, [attr(ratingValue, 4)]).`
- `aggregate_all/3` in Rule 24 counts these two ratings ('NumRatings = 2') and sums their values ($5 + 4 = 9$).
- The average is then calculated as $9/2 = 4.5$ ('AvgRating = 4.5').
- If `mod_prolog` had no ratings, the output would be `AvgRating = 'not_rated', NumRatings = 0`.

2 Tourism Knowledge Base

This section details a Prolog knowledge base for a tourism domain. It includes definitions for points of interest (POIs), means of transportation, services, attractions, and visits. Rules are established to query this information, such as finding POIs by category, hotels by features, or attractions by artist.

The representation of the generic rules mirrors the generic approach using a generated Prolog form SchemaToProlog.java using the 'tourism.gbs':

- `subclass_of/2` for entity class hierarchy.
- Entity attribute schemas like `'EntityName'([attribute1, ...])`.
- Relationship inverse definitions using `inverse_of/2`.
- Relationship signature schemas like `'RelName'([Domain-Range,...], [rel_attribute1,...])`.
- `entity/3` for entity instances.
- `rel/4` for relationship instances.
- Specific `xxx_info/N` predicates for core identifying information for related concepts like places, persons, and categories.

2.1 Tourism Domain-Specific Rules

This section details domain-specific rules designed to infer information and answer queries about the tourism knowledge base. These rules operate on the defined schema and factual instances to derive higher-level information and answer common queries pertinent to the tourism domain. The following subsections detail the initial set of these rules. There are also dynamic predicates for user preference and reccomandation and a simple dfs search for searching path of connected entities.

2.1.1 Point of Interest Identification (`is_poi/1`)

The foundational predicate `is_poi(PoiID)` ascertains if a given entity identifier, `PoiID`, represents a Point of Interest (POI). This is determined by retrieving the entity's ontological type (via `entity/3`) and subsequently verifying its hierarchical relationship to the `'PointOfInterest'` class through the `is_subclass/2` helper.

```prolog
is_poi(PoiID) :-
    entity(PoiID, Type, _),
    is_subclass(Type, 'PointOfInterest').
```

Listing 11: Rule 1: is_poi/1

2.1.2 Eating and Drinking Establishment Identification (`is_eating_place/1`)

It succeeds if the entity's type is found to be a subclass of the `'EatingDrinking'` category, thereby encompassing specific instances like restaurants, cafes, and bars as defined in the schema's hierarchy.

```prolog
is_eating_place(PlaceID) :-
    entity(PlaceID, Type, _),
    is_subclass(Type, 'EatingDrinking').
```

Listing 12: Rule 2: is_eating_place/1

2.1.3 Restaurant Classification by Food Type (`restaurant_food_type/2`)

The rule `restaurant_food_type(RestaurantID, FoodType)` facilitates the classification and retrieval of restaurants based on the specific type of food they offer. This allows for querying restaurants that specialize in, for example, 'Fish' or adhere to a 'Generic' food type classification.

```
restaurant_food_type(RestaurantID, FoodType) :-
    entity(RestaurantID, 'Restaurant', Attributes),
    member(attr(foodType, FoodType), Attributes).
```

Listing 13: Rule 3: restaurant_food_type/2

2.1.4 Hotel Classification by Star Rating (`hotel_with_star_rating/3`)

This general predicate classifies hotels based on their star rating. It confirms `HotelID` is a `'Hotel'` entity, retrieves its numerical 'stars' attribute, and succeeds if this rating satisfies a comparison defined by `Operator` like '>=', '=', etc. This rule underpins more specific classifications like luxury or budget hotels.

```
hotel_with_star_rating(HotelID, Operator, Value) :-
    entity(HotelID, 'Hotel', Attributes),
    member(attr(stars, ActualStars), Attributes),
    number(ActualStars), number(Value),
    ComparisonGoal =.. [Operator, ActualStars, Value],
    call(ComparisonGoal).
```

Listing 14: General Rule: hotel_with_star_rating/3

2.1.5 Ticket Requirement Check for POIs (`poi_requires_ticket/1`)

This rule is designed to determine whether a given Point of Interest, necessitates the purchase of an admission ticket.

```
poi_requires_ticket(PoiID) :-
    is_poi(PoiID),
    get_attribute_value(PoiID, requiresTicket, true).
```

Listing 15: Rule 6: poi_requires_ticket/1

2.1.6 Free Entry Check for POIs (`poi_is_free/1`)

This rule has a dual-condition logic to define in a comprehensive way a "free" entry for a POI.

```
poi_is_free(PoiID) :-
    is_poi(PoiID),
    get_attribute_value(PoiID, requiresTicket, false).
poi_is_free(PoiID) :-
    is_poi(PoiID),
    \+ get_attribute_value(PoiID, requiresTicket, true),
    get_attribute_value(PoiID, estimatedCost, Cost),
    Cost =:= 0.
```

Listing 16: Rule 7: poi_is_free/1

2.1.7 POI Cost Comparison (`poi_cheaper_than/2`)

This rule facilitates finding PoI whose `estimatedCost` is below a specified `MaxCost`. This is useful for budget-based filtering of attractions.

```
poi_cheaper_than(PoiID, MaxCost) :-
    is_poi(PoiID),
    get_attribute_value(PoiID, estimatedCost, Cost),
    number(Cost),
    Cost < MaxCost.
```

Listing 17: Rule 8: poi_cheaper_than/2

2.1.8 Hotel Service Offering Identification (`hotel_offers_service_type/3`)

This predicate identifies hotels offering specific types of services. It check if `HotelID` is a hotel and finds a `ServiceEntityID` related to it via the `wasIn` relationship. The rule then determines if this service qualifies: either its type is formally a subclass of `'HotelService'`, or its type is recognized as a common hotel amenity (e.g., 'Spa', 'Restaurant') by the helper.

```
is_typical_hotel_amenity_type(Type) :-
    member(Type, ['Spa', 'Resort', 'Yoga', 'Pilates', 'BikeTour', 'Restaurant', 'Bar']).

hotel_offers_service_type(HotelID, ServiceEntityID, TargetServiceType) :-
    entity(HotelID, 'Hotel', _),
    rel(wasIn, ServiceEntityID, HotelID, _),
    entity(ServiceEntityID, ActualServiceEntityType, _),
    (   is_subclass(ActualServiceEntityType, 'HotelService')
    ;   is_typical_hotel_amenity_type(ActualServiceEntityType)
    ),
    is_subclass(ActualServiceEntityType, TargetServiceType).
```

Listing 18: Rule 9 (Revised): hotel_offers_service_type/3 and helper

2.1.9 POI Location by Place Identifier (`poi_at_place/2`)

The rule link a Point of Interest to its generic location identifier and can be a foundational step for more complex geographical queries.

```
poi_at_place(PoiID, PlaceID) :-
    is_poi(PoiID),
    get_attribute_value(PoiID, place, PlaceID).
```

Listing 19: Rule 10: poi_at_place/2

2.1.10 POI Location by City Name (`poi_in_city/2`)

Building upon `poi_at_place/2`, the rule `poi_in_city(PoiID, CityName)` identifies Points of Interest situated within a specific `CityName`. It first resolves the generic `PlaceID` for a `PoiID` and then consults the `place_info/5` facts to match this `PlaceID` with the provided `CityName`. This allows for queries based on human-readable city names rather than internal place identifiers.

```
poi_in_city(PoiID, CityName) :-
    poi_at_place(PoiID, PlaceID),
    place_info(PlaceID, CityName, _, _, _).
```

Listing 20: Rule 11: poi_in_city/2

2.1.11 Relaxation Spot Identification (`is_relaxation_spot/1`)

This rule identifies if a given entity ID qualifies as a relaxation spot.

```
relaxation_type(Type) :- member(Type, ['Spa', 'Park', 'Yoga', 'Pilates', 'Resort', '
    Massage']).
is_relaxation_spot(ID) :-
  entity(ID, T, _), relaxation_type(Base),
  is_subclass(T, Base).
```

Listing 21: Rule: is_relaxation_spot/1

2.1.12 Road Vehicle Classification (`is_road_vehicle/1`)

To classify means of transportation, `is_road_vehicle(VehicleID)` determines if a `VehicleID` concern to road-based transport.

```prolog
is_road_vehicle(VehicleID) :-
    entity(VehicleID, Type, _),
    is_subclass(Type, 'RoadTransportation').
```

Listing 22: Rule 14: is_road_vehicle/1

2.1.13 Eco-Friendly Transport Identification (`is_eco_friendly_transport/1`)

The rule identifies means of transportation rated as environmentally friendly.

```prolog
is_eco_friendly_transport(TransportID) :-
    entity(TransportID, Type, _),
    ( is_subclass(Type, 'Bike') ;
      is_subclass(Type, 'Skateboard');
      get_attribute_value(TransportID, typeOfFuel, 'Electric')
    ).
```

Listing 23: Rule 15: is_eco_friendly_transport/1

2.1.14 POI Relevance to Category (`poi_for_category/2`)

The predicate links Points of Interest to thematic categories like 'Art' etc. This rule is necessary for content-based filtering and recommendations.

```prolog
poi_for_category(PoiID, CategoryName) :-
    is_poi(PoiID),
    category_info(CatID, CategoryName),
    rel(relevantFor, PoiID, CatID, _).
```

Listing 24: Rule 16: poi_for_category/2

2.1.15 Counting Points of Interest by Category (`count_pois_for_category/2`)

This predicate is designed to quantify the number of unique Points of Interest (POIs) associated with a given `CategoryName`.

```prolog
count_pois_for_category(CategoryName, Count) :-
    category_info(_CatID, CategoryName),
    (   setof(PoiID, poi_for_category(PoiID, CategoryName), PoiList)
    ->  length(PoiList, Count)
    ;   Count = 0
    ).
```

Listing 25: Rule: count_pois_for_category/2

2.1.16 Visit Planner Identification (visit_planner/2)

This rule identifies a PersonID as the planner of a VisitID if the person is linked to the visit via a makes relationship which explicitly carries the attribute role with the value 'planner'.

```
visit_planner(VisitID, PersonID) :-
    entity(VisitID, 'Visit', _),
    person_info(PersonID, _Name),
    rel(makes, PersonID, VisitID, Attributes),
    member(attr(role, 'planner'), Attributes).
```

Listing 26: Rule 17: visit_planner/2

2.1.17 POI Inclusion in a Person's Visit (poi_in_person_visit/2)

This rule identifies Points of Interest (PoiID) included in a PersonID's visit. It operates by linking the person to their visit, then to collections forming that visit, and finally to POIs belonging to those collections, effectively reconstructing itinerary components.

```
poi_in_person_visit(PersonID, PoiID) :-
    person_info(PersonID, _),
    rel(makes, PersonID, VisitID, _),
    entity(VisitID, 'Visit', _),
    rel(makes, CollectionID, VisitID, _),
    collection_info(CollectionID, _),
    rel(belongsTo, PoiID, CollectionID, _),
    is_poi(PoiID).
```

Listing 27: Rule 18: poi_in_person_visit/2

2.1.18 Attraction Material Identification (attraction_material/2)

This rule identifies an attraction's material. If MaterialString is provided, it verifies if it's a substring of the attraction's actual material. If MaterialString is a variable, it retrieves the attraction's full material description.

```
attraction_material(AttractionID, MaterialString) :-
    entity(AttractionID, 'Attraction', Attributes),
    member(attr(material, ActualMaterial), Attributes),
    (   ground(MaterialString) ->
        sub_string(ActualMaterial, _, _, _, MaterialString)
    ;
        MaterialString = ActualMaterial
    ).
```

Listing 28: Rule 19 : attraction_material/2

2.1.19 Hotel Beauty Service Offering (hotel_offers_beauty_service/3)

The predicate identifies hotels that provide beauty services. It builds upon hotel_offers_service_type/3 to first find any service (ServiceEntityID) offered by the HotelID.

```
hotel_offers_beauty_service(HotelID, ServiceEntityID, SpecificBeautyServiceType) :-
    hotel_offers_service_type(HotelID, ServiceEntityID, _AnyHotelServiceType),
    entity(ServiceEntityID, SpecificBeautyServiceType, _),
```

```prolog
    is_subclass(SpecificBeautyServiceType, 'BeautyService').
```

Listing 29: Rule 23: hotel_offers_beauty_service/3

2.1.20 Enumerating Services of Luxury Hotels

This rules returns a list of services offered by Luxury Hotels. Uses setof for gathering a list of unique services.

```prolog
luxury_hotel_all_services_safe(LuxuryHotelID, ListOfServiceIDs) :-
    is_luxury_hotel(LuxuryHotelID),
    (   setof(ServiceID, _AnyServiceType^(hotel_offers_service_type(LuxuryHotelID,
    ServiceID, _AnyServiceType)), ListOfServiceIDs)
    ->  true
    ;   ListOfServiceIDs = []
    ).
```

Listing 30: Rule: luxury_hotel_all_services_safe/2

2.1.21 Aggregating Average Museum Costs by City (`average_museum_cost_in_city/2`)

This predicate computes the mean estimated admission cost for all museums located within a specified `CityName`.

```prolog
average_museum_cost_in_city(CityName, AverageCost) :-
    setof(Cost, MID^(_AnyOtherVar)^(
                poi_in_city(MID, CityName),
                entity(MID, 'Museum', _),
                get_attribute_value(MID, estimatedCost, Cost),
                number(Cost)
            ), CostsList),
    CostsList \= [],
    sum_list(CostsList, TotalCost),
    length(CostsList, NumberOfMuseums),
    AverageCost is TotalCost / NumberOfMuseums.
```

Listing 31: Rule: average_museum_cost_in_city/2

2.1.22 Similar POI Recommendation (`recommend_similar_poi/2`)

The rule `recommend_similar_poi(GivenPoiID, RecommendedPoiID)` provides a basic content-based recommendation. Given a `GivenPoiID`, it suggests a `RecommendedPoiID` if both are Points of Interest, they are distinct, they share the exact same ontological type (e.g., both are 'Museums'), and they are located in the same city (determined via `poi_in_city/2`).

```prolog
recommend_similar_poi(GivenPoiID, RecommendedPoiID) :-
    is_poi(GivenPoiID),
    entity(GivenPoiID, Type, _),
    poi_in_city(GivenPoiID, CityName),
    is_poi(RecommendedPoiID),
    RecommendedPoiID \== GivenPoiID,
    entity(RecommendedPoiID, Type, _),
    poi_in_city(RecommendedPoiID, CityName).
```

Listing 32: recommend_similar_poi/2

2.1.23 Visit Retrieval by Planner and Budget Range (find_visits_by_planner_and_budget/4)

The rule retrieves visits planned by a specific `PersonID` (identified via `visit_planner/2`) whose associated 'budget' attribute falls within the specified `MinBudget` and `MaxBudget` (inclusive). This allows for filtering travel plans based on planner and financial constraints.

```prolog
find_visits_by_planner_and_budget(PersonID, MinBudget, MaxBudget, VisitID) :-
    visit_planner(VisitID, PersonID),
    entity(VisitID, 'Visit', Attributes),
    member(attr(budget, Budget), Attributes),
    Budget >= MinBudget,
    Budget =< MaxBudget.
```

Listing 33: find_visits_by_planner_and_budget/4

2.1.24 Direct Local Travel Feasibility Between POIs (can_travel_directly/3)

The predicate assesses the feasibility of direct local travel between two distinct Points of Interest, `Poi1` and `Poi2`.

```prolog
can_travel_directly(Poi1, Poi2, 'local_transport') :-
    is_poi(Poi1), is_poi(Poi2), Poi1 \== Poi2,
    poi_in_city(Poi1, CityName),
    poi_in_city(Poi2, CityName).
```

Listing 34: can_travel_directly/3

2.1.25 Customizable Point of Interest Search with Multiple Filters (poi_with_custom_filters/15)

The predicate `poi_with_custom_filters/15` facilitates a flexible search for Points of Interest (`PoiID`) by allowing a combination of criteria to be dynamically applied. Each filter, corresponding to attributes such as city, entity type, category, age suitability, visit time limit, maximum cost, and entrance fee, is controlled by a respective boolean-like enable flag (e.g., `ENCity`, `ENEntityType`).

The rule operates by first ensuring the `PoiID` represents a valid Point of Interest. Subsequently, it iterates through each potential filter: if a filter's enable flag is true, the associated condition must be met by the `PoiID`; otherwise, the filter is bypassed. This conditional application, implemented using Prolog's if-then-else construct `(Flag -> Goal ; true)`, allows for tailored queries. Specific conditions leverage existing domain rules (e.g., `poi_in_city/2`, `poi_for_category/2`, `poi_cheaper_than/2`) or direct attribute checks, with entrance fee logic encapsulated in the helper predicate `filter_entrance_fee/2`.

```prolog
poi_with_custom_filters(
    PoiID,
    CityFilterValue, EntityTypeFilterValue,
    CategoryFilterValue, AgeFilterValue,
    TimeLimitFilterValue, MaxCostFilterValue, EntranceFeeFilterValue,
    EN_City, EN_EntityType,
    EN_Category, EN_Age,
    EN_TimeLimit, EN_MaxCost, EN_EntranceFee
) :-
    is_poi(PoiID),
    (EN_City -> poi_in_city(PoiID, CityFilterValue) ; true),
    (EN_EntityType -> entity(PoiID, ActualType, _), is_subclass(ActualType,
    EntityTypeFilterValue) ; true),
    (EN_Category -> poi_for_category(PoiID, CategoryFilterValue) ; true),
    (EN_Age -> activity_for_age(_, AgeFilterValue, PoiID) ; true),
    (EN_TimeLimit -> poi_has_time_limit(PoiID, ActualTimeLimit), ActualTimeLimit =<
    TimeLimitFilterValue ; true),
    (EN_MaxCost -> poi_cheaper_than(PoiID, MaxCostFilterValue) ; true),
    (EN_EntranceFee -> filter_entrance_fee(PoiID, EntranceFeeFilterValue) ; true).

filter_entrance_fee(PoiID, free) :-
    poi_is_free(PoiID).
filter_entrance_fee(PoiID, paid) :-
    poi_requires_ticket(PoiID).
filter_entrance_fee(PoiID, MaxCost) :-
    number(MaxCost),
    get_attribute_value(PoiID, estimatedCost, Cost),
    (Cost == default -> true ; number(Cost), Cost =< MaxCost).
filter_entrance_fee(_, _).
```

Listing 35: Rule: poi_with_custom_filters/15 and helper filter_entrance_fee/2

2.2 Dynamic Predicates for Tourism: User Preference Management and Recommendation

To enable personalized experiences and recommendations, the knowledge base incorporates mechanisms for dynamically managing user preferences and suggesting relevant Points of Interest. This is achieved through a set of predicates that allow for the assertion, retraction, and querying of user-specific POI preferences, along with a rule for generating recommendations based on these stored preferences.

2.2.1 Adding a User Preference (`add_preference/2`)

The predicate `add_preference(UserID, PoiID)` is responsible for recording a user's preference for a specific Point of Interest. Before asserting it also checks if the preference already exists using `\+ user_preference(UserID, PoiID)`. If all conditions are met it is asserted into the knowledge base using `assertz/1`. The rule provides informative feedback regarding the success, prior existence, or failure of the operation.

```
:- dynamic user_preference/2.

add_preference(UserID, PoiID) :-
    person_info(UserID, _),
    is_poi(PoiID),
    \+ user_preference(UserID, PoiID),
    assertz(user_preference(UserID, PoiID)),
    format('Preference: ~w liked ~w added.~n', [UserID, PoiID]).
add_preference(UserID, PoiID) :-
    user_preference(UserID, PoiID),
    format('INFO: ~w already liked ~w.~n', [UserID, PoiID]).
add_preference(UserID, _PoiID) :-
    \+ person_info(UserID, _),
    format('ERROR: User ~w not found. Preference not added.~n', [UserID]),
    !, fail.
add_preference(_UserID, PoiID) :-
    \+ is_poi(PoiID),
    format('ERROR: POI ~w not found. Preference not added.~n', [PoiID]),
    !, fail.
```

Listing 36: Rule: add_preference/2

2.2.2 Removing a User Preference (`remove_preference/2`)

To allow users to modify their indicated preferences, the rule `remove_preference(UserID, PoiID)` facilitates the deletion of an existing preference.

```
remove_preference(UserID, PoiID) :-
    user_preference(UserID, PoiID),
    retract(user_preference(UserID, PoiID)),
    format('Preference: ~w unliked ~w removed.~n', [UserID, PoiID]).
remove_preference(UserID, PoiID) :-
    \+ user_preference(UserID, PoiID),
    format('INFO: ~w did not have ~w as a preference. Nothing removed.~n', [UserID,
    PoiID]).
```

Listing 37: Rule: remove_preference/2

2.2.3 Listing User Preferences (`list_preferences/1`)

The predicate `list_preferences(UserID)` provides a user-friendly display of all Points of Interest that a given `UserID` has marked as preferred.

```
list_preferences(UserID) :-
    person_info(UserID, UserName),
    format('Preferences for ~w (~w):~n', [UserName, UserID]),
    (   user_preference(UserID, _PoiID)
    ->  forall(user_preference(UserID, CurrentPoiID),
               (get_entity_display_name(CurrentPoiID, PoiDisplayName),
                format('  - ~w (~w)~n', [PoiDisplayName, CurrentPoiID]))
            )
    ;   format('  No preferences found for this user.~n', [])
    ).
list_preferences(UserID) :-
    \+ person_info(UserID, _),
    format('ERROR: User ~w not found.~n', [UserID]).
```

Listing 38: Rule: list_preferences/1

2.2.4 Recommendation Based on Existing Preferences (`recommend_based_on_prefs/2`)

The rule implements a basic collaborative filtering-style recommendation. It operates by first identifying a `LikedPOI` from the existing preferences. It then leverages the rule `recommend_similar_poi/2` to find a `Recommendation` that is similar to this `LikedPOI`. To ensure novelty, the rule verifies that the `Recommendation` is not the same as the `LikedPOI` and that the `User` has not already expressed a preference for the `Recommendation`. This predicate can throw multiple recommendations upon backtracking.

```
recommend_based_on_prefs(User, Recommendation) :-
    user_preference(User, LikedPOI),
    % entity(LikedPOI, _Type, _Attributes), % Validation implicitly handled by
    recommend_similar_poi
    recommend_similar_poi(LikedPOI, Recommendation),
    Recommendation \== LikedPOI,
    \+ user_preference(User, Recommendation).
```

Listing 39: Rule: recommend_based_on_prefs/2

2.3 Rule: Generic Pathfinding via Depth-First Search (go/2)

The predicate `go(Start, Goal)` implements a Depth-First Search (DFS) algorithm to find out if a path of connected entities exist. The search navigates through connections defined by a `mov(Entity1, Entity2)` predicate, which encapsulates various types of relationships and links between entities.

The core of the pathfinding logic resides in the recursive `path(CurrentState, Goal, VisitedList)` predicate. This predicate explores possible next states from the `CurrentState` using `mov/2`. To prevent cycles and redundant exploration, it maintains a `VisitedList` of entities already encountered on the current path.

The `mov(EntityA, EntityB)` It is composed of multiple clauses, each representing a specific type of link within the tourism kb. For instance, to find a path such as 'mona_lisa' → 'louvre' → 'p1' (Paris City Center), `mov/2` would include clauses to recognize:

- An `Attraction` being located `wasIn` a `PointOfInterest` (e.g., 'mona_lisa' in 'louvre').
- A `PointOfInterest` being geographically situated at a `Place` (e.g., 'louvre' at 'p1'), often via its 'place' attribute or a `wasIn` relationship to a `place_info` identifier.

The predicate is typically defined to be symmetric or includes clauses for inverse relationships to allow traversal in either direction along these conceptual edges in the knowledge graph. Other clauses might define connections based on relationships like `developed`, `belongsTo`, or generic `rel/4` facts.

```prolog
go(Start, Goal) :-
    empty_stack(Empty_been_list),
    stack(Start, Empty_been_list, Been_list),
    path(Start, Goal, Been_list).

path(Goal, Goal, Been_list) :-
    write('Path found:'), nl,
    reverse_print_stack_with_names(Been_list),
    !.
path(State, Goal, Been_list) :-
    mov(State, Next),
    not(member_stack(Next, Been_list)),
    stack(Next, Been_list, New_been_list),
    path(Next, Goal, New_been_list).
```

Listing 40: DFS Pathfinding Core: go/2 and path/3

```prolog
mov(Entity1, Entity2) :-
    rel(_RelName, Entity1, Entity2, _Attrs).

mov(Entity1, Entity2) :-
    (   inverse_of(InvRelName, RelName)
    ;   inverse_of(RelName, InvRelName)
    ),
    rel(RelName, Entity2, Entity1, _Attrs).

mov(AttractionID, PoiID) :-
    entity(AttractionID, TypeA, _), is_subclass(TypeA, 'Attraction'),
    entity(PoiID, TypeP, _), is_subclass(TypeP, 'PointOfInterest'),
    rel(wasIn, AttractionID, PoiID, _).

mov(PoiID, AttractionID) :-
    entity(AttractionID, TypeA, _), is_subclass(TypeA, 'Attraction'),
    entity(PoiID, TypeP, _), is_subclass(TypeP, 'PointOfInterest'),
    rel(wasIn, AttractionID, PoiID, _).

mov(PoiID, PlaceID) :-
    entity(PoiID, TypeP, _), is_subclass(TypeP, 'PointOfInterest'),
    place_info(PlaceID, _, _, _, _),
    ( rel(wasIn, PoiID, PlaceID, _);
    get_attribute_value(PoiID, place, PlaceID)
    ).

mov(PlaceID, PoiID) :-
    entity(PoiID, TypeP, _), is_subclass(TypeP, 'PointOfInterest'),
    place_info(PlaceID, _, _, _, _),
    ( rel(wasIn, PoiID, PlaceID, _)
    ; get_attribute_value(PoiID, place, PlaceID)
    ).
```

Listing 41: Connection Predicate: mov/2 (Illustrative Clauses for Tourism Pathfinding)

2.4 Query Execution Examples and Outputs (Tourism Domain)

This section demonstrates the execution of selected queries within the Tourism Knowledge Base, illustrating how the defined rules and facts can be used to derive insights and manage dynamic information.

2.4.1 Example 1: Identifying Luxury Hotels

The rule `is_luxury_hotel/1` (not explicitly shown in prior rule listings but assumed to be defined similarly to `is_budget_hotel/1` or based on star ratings, e.g., >= 5 stars using `hotel_with_star_rating/3`) is used to find all hotels classified as luxury.

Query:

```
?- is_luxury_hotel(H).
```

Listing 42: Query for Luxury Hotels

Output:

```
H = grand_hotel_paris ;
H = ryokan_kyoto.
```

Explanation:

- The query asks Prolog to find all hotel identifiers `H` that satisfy the `is_luxury_hotel/1` predicate.
- The output indicates that, based on the knowledge base's facts and rules (e.g., a rule checking for hotels with a 5-star rating or a direct `subclass_of('GrandHotelParis', 'LuxuryHotel')` fact), `grand_hotel_paris` and `ryokan_kyoto` are identified as luxury hotels.
- Prolog backtracks to find all possible solutions.

2.4.2 Example 2: Finding Categories for a Point of Interest

The rule `poi_for_category/2` links Points of Interest to their relevant thematic categories.

Query:

```
?- poi_for_category(colosseum, CategoryName).
```

Listing 43: Categories for the Colosseum

Output:

```
CategoryName = 'History' ;
CategoryName = 'Architecture'.
```

Explanation:

- This query seeks all `CategoryName` values associated with the POI identified as `colosseum`.
- The rule `poi_for_category/2` would look for `rel(relevantFor, colosseum, CatID, _)` facts and then use `category_info(CatID, CategoryName)` to get the human-readable names.
- The output shows that the Colosseum is categorized under both 'History' and 'Architecture'.

2.4.3 Example 3: Managing User Preferences (Dynamic Predicates)

The `add_preference/2` rule utilizes dynamic predicates (`assertz/1`) to modify the knowledge base by adding user preferences.

Queries & Outputs:

- **Adding a new preference:**

  ```
  ?- add_preference(person1, louvre).
  ```

  Listing 44: Adding a New Preference for Louvre

  Output:

  ```
  Preference: person1 liked louvre added.
  true.
  ```

  Explanation: The rule successfully validates `person1` and `louvre`, checks that the preference doesn't already exist, asserts `user_preference(person1, louvre)`, and prints a confirmation.

- **Attempting to add an existing preference:**

  ```
  ?- add_preference(person1, louvre).
  ```

  Listing 45: Re-adding Preference for Louvre

  Output:

  ```
  INFO: person1 already liked louvre.
  true.
  ```

  Explanation: The rule detects that `user_preference(person1, louvre)` already exists in the KB (from the previous assertion) and informs the user without re-asserting.

Overall Explanation:

- This example demonstrates the ability of the KB to dynamically change by adding new facts at runtime.
- The rule includes logic to prevent duplicate assertions and provides informative feedback.

2.4.4 Example 4: Pathfinding with DFS

The `go/2` rule, utilizing a Depth-First Search (DFS) algorithm via its helper `path/3` and connection predicate `mov/2`, finds a path between two entities.

Query:

```
?- go(mona_lisa, p1).
```

Listing 46: Pathfinding from Mona Lisa to Paris City Center (p1)

Output:

```
Path found:
Mona Lisa
Louvre Museum
p1
true.
```

Explanation:

- The query asks if a path exists from the entity `mona_lisa` to `p1`.
- The `go/2` rule initiates the DFS. The `path/3` predicate recursively explores connections defined by `mov/2`.
- For this path to be found, `mov/2` would need to define connections such as:
  - `mov(mona_lisa, louvre)`: e.g., an attraction is 'in' a museum.
  - `mov(louvre, p1)`: e.g., a museum (POI) is located 'at' a place/city identifier.
- The output shows the discovered path, printed in reverse order of discovery by `reverse_print_stack_with`
- The final 'true.' indicates the successful completion of the 'go/2' goal.

3 Conclusion

This document has outlined the structure and components of two distinct Prolog knowledge bases: one for an educational LMS based on the `education.gbs` ontology and another for a tourism domain. Both utilize a common set of generic helper predicates for schema introspection while featuring domain-specific schemas, facts, and inferential rules. This approach allows for modular knowledge representation and reasoning tailored to each application area. Further extensions could involve more complex rules, integration with external data sources, or the application of Inductive Logic Programming (ILP) techniques to automatically learn new rules.