Prolog Project Code

GitHub Copilot

September 29, 2025

Contents

1 api_server.pl

```
/** <module> Full-featured API Server for Cognitive Modeling
 *
 * This module provides a comprehensive HTTP server that exposes the full
 * capabilities of the cognitive modeling system. It integrates various
 * components, including the core execution handler for the ORR (Observe,
 * Reorganize, Reflect) cycle, logging, semantic analysis, and student
 * strategy analysis.
 *
 * This server is intended for development and provides a richer set of
 * endpoints compared to `working_server.pl`.
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- use_module(library(http/thread_httpd)).
:- use_module(library(http/http_dispatch)).
:- use_module(library(http/http_json)).
:- use_module(library(http/json_convert)).
:- use_module(library(http/http_cors)).
:- use_module(library(http/http_header)).

% Load the core application logic
:- use_module(execution_handler).
:- use_module(reorganization_log).
:- use_module(reflective_monitor).
:- use_module(object_level).
:- use_module(incompatibility_semantics).
:- use_module(hermeneutic_calculator).

% Define the REST API endpoints
:- http_handler(root(solve), solve_handler, [method(post)]).
:- http_handler(root(log), log_handler, [method(get)]).
:- http_handler(root(knowledge), knowledge_handler, [method(get)]).
:- http_handler(root(analyze_semantics), analyze_semantics_handler, [method(post)]).
:- http_handler(root(analyze_strategy), analyze_strategy_handler, [method(post)]).

% Enable CORS for all endpoints
:- set_setting(http:cors, [*]).

%!      server(+Port:integer) is det.
%
%       Starts the HTTP server on the specified Port.
%
%       @param Port The port number for the server to listen on.
server(Port) :-
    http_server(http_dispatch, [port(Port)]).

% --- Endpoint Handlers ---

%!      solve_handler(+Request:list) is det.
%
%       Handles POST requests to the `/solve` endpoint.
%       It expects a JSON object with a `goal` key, e.g., `{"goal": "add(s(0),s(0),X)"}`.
%       It runs the full ORR (Observe, Reorganize, Reflect) cycle for the given
%       goal and returns the final result.
%
%       @param Request The incoming HTTP request.
```

```prolog
solve_handler(Request) :-
    http_read_json_dict(Request, In),
    term_string(Goal, In.goal),
    % Run the query, which performs the full ORR cycle
    run_query(Goal),
    % After the cycle, find the result
    (   clause(object_level:Goal, true) ->
        Result = Goal,
        Status = 'success'
    ;   Result = 'failed to find solution',
        Status = 'failure'
    ),
    term_string(ResultString, Result),
    reply_json_dict(_{status: Status, result: ResultString}).


%!      log_handler(+Request:list) is det.
%
%       Handles GET requests to the `/log` endpoint.
%       It generates and returns the full reorganization log as a JSON object,
%       detailing the cognitive steps taken by the system.
%
%       @param _Request The incoming HTTP request (unused).
log_handler(_Request) :-
    generate_report(Report),
    reply_json_dict(_{report: Report}).


%!      knowledge_handler(+Request:list) is det.
%
%       Handles GET requests to the `/knowledge` endpoint.
%       It returns the current state of the system's knowledge base, including
%       all clauses in the `object_level` module and the current conceptual
%       stress map.
%
%       @param _Request The incoming HTTP request (unused).
knowledge_handler(_Request) :-
    findall(
        Clause,
        (clause(object_level:Head, Body), Clause = (Head :- Body)),
        Clauses
    ),
    get_stress_map(StressMap),
    prolog_to_json(_{clauses: Clauses, stress_map: StressMap}, JSON_Object),
    reply_json(JSON_Object).


%!      analyze_semantics_handler(+Request:list) is det.
%
%       Handles POST requests to the `/analyze_semantics` endpoint.
%       It expects a JSON object with a `statement` key, e.g., `{"statement": "The object is red"}`.
%       It performs a semantic analysis of the statement based on incompatibility semantics.
%
%       @param Request The incoming HTTP request.
analyze_semantics_handler(Request) :-
    cors_enable(Request, [methods([post, options])]),
    (   http_read_json_dict(Request, In) ->
        Statement = In.statement,
        analyze_statement_semantics(Statement, Analysis),
```

```prolog
        reply_json_dict(Analysis)
    ;   reply_json_dict(_{error: "Invalid JSON input"})
    ).


%!      analyze_strategy_handler(+Request:list) is det.
%
%       Handles POST requests to the `/analyze_strategy` endpoint.
%       It expects a JSON object with `problemContext` and `strategy` keys,
%       e.g., `{"problemContext": "Math-JRU", "strategy": "student counted all"}`.
%       It returns a CGI/Piagetian analysis of the described student strategy.
%
%       @param Request The incoming HTTP request.
analyze_strategy_handler(Request) :-
    cors_enable(Request, [methods([post, options])]),
    (   http_read_json_dict(Request, In) ->
        ProblemContext = In.problemContext,
        StrategyDescription = In.strategy,
        analyze_cgi_strategy(ProblemContext, StrategyDescription, Analysis),
        reply_json_dict(Analysis)
    ;   reply_json_dict(_{error: "Invalid JSON input"})
    ).


% --- Helper for JSON conversion ---

%!      json_convert:prolog_to_json(+Term, -JSON) is multi.
%
%       A multifile predicate that extends the default JSON conversion library.
%       This implementation is needed to handle the conversion of complex Prolog
%       terms (like rule bodies) into a structured JSON format.
%
%       @param Term The Prolog term to convert.
%       @param JSON The resulting JSON object.
:- multifile json_convert:prolog_to_json/2.
json_convert:prolog_to_json(Term, JSON) :-
    is_list(Term), !,
    maplist(json_convert:prolog_to_json, Term, JSON).
json_convert:prolog_to_json(Term, JSON) :-
    compound(Term),
    Term =.. [Functor | Args],
    maplist(json_convert:prolog_to_json, Args, JSONArgs),
    JSON = _{functor: Functor, args: JSONArgs}.
json_convert:prolog_to_json(Term, JSON) :-
    \+ compound(Term),
    term_string(Term, JSON).

% --- Helper Predicates for Analysis ---

%!      analyze_statement_semantics(+Statement:string, -Analysis:dict) is det.
%
%       Performs semantic analysis on a given statement.
%       It finds all implications and incompatibilities for the normalized
%       (lowercase) statement.
%
%       @param Statement The input string to analyze.
%       @param Analysis A dict containing the original statement, a list of
%       implications, and a list of incompatibilities.
analyze_statement_semantics(Statement, Analysis) :-
```

```
    atom_string(StatementAtom, Statement),
    downcase_atom(StatementAtom, Normalized),

    % Basic semantic analysis based on statement content
    findall(Implication, get_implications(Normalized, Implication), Implies),
    findall(Incompatibility, get_incompatibilities(Normalized, Incompatibility), IncompatibleWith),

    Analysis = _{
        statement: Statement,
        implies: Implies,
        incompatibleWith: IncompatibleWith
    }.

%!      get_implications(+Statement:atom, -Implication:string) is nondet.
%
%       Generates implications for a given statement.
%       This predicate defines the semantic entailments based on keywords
%       found in the statement. It is a multi-clause predicate where each
%       clause represents a different implication rule.
%
%       @param Statement The normalized (lowercase) input atom.
%       @param Implication A string describing what the statement implies.
get_implications(Statement, 'The object is colored') :-
    sub_atom(Statement, _, _, _, red).
get_implications(Statement, 'The shape is a rectangle') :-
    sub_atom(Statement, _, _, _, square).
get_implications(Statement, 'The shape is a polygon') :-
    sub_atom(Statement, _, _, _, square).
get_implications(Statement, 'The shape has 4 sides of equal length') :-
    sub_atom(Statement, _, _, _, square).
get_implications(Statement, 'This statement has semantic content') :-
    Statement \= ''.

%!      get_incompatibilities(+Statement:atom, -Incompatibility:string) is nondet.
%
%       Generates incompatibilities for a given statement.
%       This predicate defines what a statement semantically rules out based
%       on keywords. It is a multi-clause predicate where each clause
%       represents a different incompatibility rule.
%
%       @param Statement The normalized (lowercase) input atom.
%       @param Incompatibility A string describing what the statement is incompatible with.
get_incompatibilities(Statement, 'The object is entirely blue') :-
    sub_atom(Statement, _, _, _, red).
get_incompatibilities(Statement, 'The object is monochromatic and green') :-
    sub_atom(Statement, _, _, _, red).
get_incompatibilities(Statement, 'The shape is a circle') :-
    sub_atom(Statement, _, _, _, square).
get_incompatibilities(Statement, 'The shape has exactly 3 sides') :-
    sub_atom(Statement, _, _, _, square).
get_incompatibilities(Statement, 'The negation of this statement') :-
    Statement \= ''.

%!      analyze_cgi_strategy(+ProblemContext:string, +StrategyDescription:string, -Analysis:dict) is
%
%       Analyzes a student's problem-solving strategy within a given context.
%       It normalizes the strategy description and uses `classify_strategy/7`
%       to get a detailed analysis.
%
```

```
%       @param ProblemContext The context of the problem (e.g., "Math-Addition").
%       @param StrategyDescription A text description of the student's strategy.
%       @param Analysis A dict containing the classification, developmental stage,
%       implications, incompatibilities, and pedagogical recommendations.
analyze_cgi_strategy(ProblemContext, StrategyDescription, Analysis) :-
    atom_string(StrategyAtom, StrategyDescription),
    downcase_atom(StrategyAtom, Normalized),

    classify_strategy(ProblemContext, Normalized, Classification, Stage, Implications, Incompatibili

    Analysis = _{
        classification: Classification,
        stage: Stage,
        implications: Implications,
        incompatibility: Incompatibility,
        recommendations: Recommendations
    }.

%!      classify_strategy(+Context:string, +Strategy:atom, -Classification:string, -Stage:string, -I
%
%       Classifies a student's strategy based on context and description.
%       This multi-clause predicate uses keyword matching on the strategy
%       description to determine the CGI classification, Piagetian stage,
%       and associated pedagogical insights for various domains (Math, Science).
%
%       @param Context The problem context (e.g., "Math-Addition", "Science-Float").
%       @param Strategy The normalized student strategy description.
%       @param Classification The CGI classification of the strategy.
%       @param Stage The associated Piagetian developmental stage.
%       @param Implications What the strategy implies about the student's understanding.
%       @param Incompatibility The conceptual conflict this strategy might lead to.
%       @param Recommendations Pedagogical suggestions to advance the student's understanding.
classify_strategy(Context, Strategy, Classification, Stage, Implications, Incompatibility, Recommend
    atom_string(Context, ContextStr),
    sub_atom(ContextStr, 0, 4, _, "Math"),
    (   (sub_atom(Strategy, _, _, _, 'count all') ;
         sub_atom(Strategy, _, _, _, 'starting from one') ;
         sub_atom(Strategy, _, _, _, '1, 2, 3')) ->
        Classification = "Direct Modeling: Counting All",
        Stage = "Preoperational (Piaget)",
        Implications = "The student needs to represent the quantities concretely and cannot treat th
        Incompatibility = "A commitment to 'Counting All' is incompatible with the concept of 'Cardi
        Recommendations = "Encourage 'Counting On'. Ask: 'You know there are 5 here. Can you start c
    ;   (sub_atom(Strategy, _, _, _, 'count on') ;
         sub_atom(Strategy, _, _, _, 'started at 5')) ->
        Classification = "Counting Strategy: Counting On",
        Stage = "Concrete Operational (Early)",
        Implications = "The student understands the cardinality of the first number. This is a signi
        Incompatibility = "Reliance on 'Counting On' is incompatible with the immediate retrieval re
        Recommendations = "Work on derived facts. Ask: 'If you know 5 + 5 = 10, how can that help yo
    ;   (sub_atom(Strategy, _, _, _, 'known fact') ;
         sub_atom(Strategy, _, _, _, 'just knew')) ->
        Classification = "Known Fact / Fluency",
        Stage = "Concrete Operational",
        Implications = "The student has internalized the number relationship.",
        Incompatibility = "",
        Recommendations = "Introduce more complex problem structures (e.g., Join Change Unknown or m
    ;
        Classification = "Unclassified",
```

```
        Stage = "Unknown",
        Implications = "Could not clearly identify the strategy based on the description. Please pro
        Incompatibility = "",
        Recommendations = ""
    ).

classify_strategy("Science-Float", Strategy, Classification, Stage, Implications, Incompatibility, R
    (   (sub_atom(Strategy, _, _, _, heavy) ; sub_atom(Strategy, _, _, _, big)) ->
        Classification = "Perceptual Reasoning: Weight/Size as defining factor",
        Stage = "Preoperational",
        Implications = "The student is focusing on salient perceptual features (size, weight) rather
        Incompatibility = "The concept that 'heavy things sink' is incompatible with observations of
        Recommendations = "Introduce an incompatible observation (disequilibrium). Show a very large
    ;
        Classification = "Unclassified",
        Stage = "Unknown",
        Implications = "Could not clearly identify the strategy based on the description. Please pro
        Incompatibility = "",
        Recommendations = ""
    ).

% Default case for unmatched contexts
classify_strategy(_, _, "Unclassified", "Unknown", "Could not clearly identify the strategy based on

% To run the server from the command line:
% swipl api_server.pl -g "server(8000)"
:- initialization(server(8000), main).
```

2 cognition_viz.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Cognitive Reorganization Visualization (Prolog/WASM/D3)</title>
    <script src="https://d3js.org/d3.v7.min.js"></script>
    <script src="https://cdn.jsdelivr.net/npm/swipl-wasm@3.3.1/dist/swipl-web.js"></script>
    <style>
        body { font-family: Arial, sans-serif; display: flex; margin: 0; height: 100vh; }
        #sidebar { width: 350px; padding: 20px; background-color: #f4f4f4; display: flex; flex-direc
        #visualization { flex-grow: 1; }

        /* D3 Visualization Styles */
        .link { stroke: #999; stroke-opacity: 0.6; }

        /* Node Styles: Differentiating concepts and entities */
        .node-entity { fill: #2ca02c; } /* Green for entities */
        .node-predicate { fill: #1f77b4; } /* Blue for predicates/facts */

        /* Visualizing Disequilibrium (Incompatibility Conflict) */
        .inconsistent {
            stroke: #d62728; /* Red border */
            stroke-width: 4px;
        }

        /* Interface Styles */
        #controls { margin-bottom: 20px; }
        input[type="text"] { padding: 8px; width: 70%; font-size: 14px; }
        button { padding: 8px 12px; margin-left: 5px; cursor: pointer; font-size: 14px; }
```

```
        #output { flex-grow: 1; white-space: pre-wrap; background: #333; color: #f0f0f0; padding: 15
    </style>
</head>
<body>

<div id="sidebar">
    <h2>Cognitive Model Control</h2>
    <p>Visualize the synthesis of Incompatibility Semantics and Piagetian Constructivism.</p>

    <div id="controls">
        <label for="newFact">Introduce Information:</label><br>
        <input type="text" id="newFact" placeholder="e.g., penguin(tweety)">
        <button onclick="introduceInformation()">Learn</button>
        <p><i>Try introducing conflicting information (e.g., <code>penguin(tweety)</code> or <code>m
    </div>

    <h3>Engine Output (Equilibration Process)</h3>
    <div id="output">Initializing Prolog WASM engine...</div>
</div>

<div id="visualization">
    <svg width="100%" height="100%"></svg>
</div>

<script type="text/prolog" id="cognitionCode">
% ----------------------------------------------------------------
% Cognitive Model: Incompatibility, Constructivism, Embodiment
% ----------------------------------------------------------------

% Ensure facts can be dynamically added/removed during reorganization
:- dynamic fact/1.

% Initial knowledge base (Example)
fact(flies(tweety)).
fact(bird(tweety)).
fact(swims(willy)).
fact(fish(willy)).
fact(breathes_air(willy)).

% Incompatibility Semantics (Brandom)
% Defining what cannot be materially true simultaneously.
incompatible(flies(X), penguin(X)).
incompatible(fish(X), mammal(X)).
% Example incorporating embodiment: physical constraints
incompatible(breathes_air(X), lives_underwater(X)).

% ----------------------------------------------------------------
% Reasoning Mechanisms (Piaget)
% ----------------------------------------------------------------

% Check for inconsistencies (Cognitive Disequilibrium)
find_inconsistency(Entity, Fact1, Fact2) :-
    fact(Fact1),
    fact(Fact2),
    Fact1 \= Fact2,
    % Check incompatibility in both directions
    (incompatible(Fact1, Fact2); incompatible(Fact2, Fact1)),
    % Ensure they apply to the same entity (simplified unification check)
    Fact1 =.. [_, Entity],
```

```prolog
    Fact2 =.. [_, Entity].

% Equilibration Process: Assimilation and Accommodation
learn(NewFact) :-
    % 1. Attempt Assimilation: Add the fact to the knowledge base
    assertz(fact(NewFact)),
    write('Assimilating: '), write(NewFact), nl,

    % 2. Check for Disequilibrium
    findall((E, F1, F2), find_inconsistency(E, F1, F2), Inconsistencies),

    ( Inconsistencies \= [] ->
        % Disequilibrium detected
        write('Disequilibrium detected. Initiating accommodation...\n'),
        % 3. Initiate Accommodation: Reorganize the structure
        resolve_inconsistencies(Inconsistencies),
        write('Accommodation complete: Structure reorganized.\n')
    ;
        % No conflict
        write('Assimilation successful: Knowledge structure stable.\n')
    ).

% Accommodation Logic (Resolution Strategy)
% This defines the prioritization of beliefs and how the system adapts.
resolve_inconsistencies([]).

% Specific resolution rule 1: If we learn X is a penguin, we prioritize this over the default belief
resolve_inconsistencies([(E, flies(E), penguin(E))|T]) :-
    retract(fact(flies(E))),
    format('  Resolved: Retracted flies(~w) due to new evidence penguin(~w).\n', [E, E]),
    resolve_inconsistencies(T).
resolve_inconsistencies([(E, penguin(E), flies(E))|T]) :-
    retract(fact(flies(E))),
    format('  Resolved: Retracted flies(~w) due to new evidence penguin(~w).\n', [E, E]),
    resolve_inconsistencies(T).


% Specific resolution rule 2: If we learn X is a mammal, we retract that X is a fish.
resolve_inconsistencies([(E, fish(E), mammal(E))|T]) :-
    retract(fact(fish(E))),
    format('  Resolved: Retracted fish(~w) due to reclassification as mammal(~w).\n', [E, E]),
    resolve_inconsistencies(T).
resolve_inconsistencies([(E, mammal(E), fish(E))|T]) :-
     retract(fact(fish(E))),
    format('  Resolved: Retracted fish(~w) due to reclassification as mammal(~w).\n', [E, E]),
    resolve_inconsistencies(T).


% Fallback resolution
resolve_inconsistencies([_|T]) :-
    resolve_inconsistencies(T).

% ----------------------------------------------------------------
% Visualization Extraction Utility
% ----------------------------------------------------------------

% Extract graph data (Nodes and Edges) for D3.js
get_graph_data(Nodes, Edges) :-
    % 1. Collect all current facts
```

```
    findall(F, fact(F), Facts),

    % 2. Identify entities currently involved in inconsistencies (if any remain after accommodation)
    findall(E, find_inconsistency(E, _, _), InconsistentEntitiesRaw),
    sort(InconsistentEntitiesRaw, InconsistentEntities),

    % 3. Process facts into raw nodes and edges
    process_facts(Facts, NodesList, EdgesList),

    % 4. Deduplicate nodes and mark those involved in conflicts
    deduplicate_and_mark(NodesList, InconsistentEntities, Nodes),
    Edges = EdgesList.

% Convert Prolog facts into graph elements
process_facts([], [], []).
process_facts([Fact|T], [NodeE, NodeP|NodesT], [Edge|EdgesT]) :-
    Fact =.. [Predicate, Entity],
    format(atom(PName), '~w', [Predicate]),
    format(atom(EName), '~w', [Entity]),

    % Define Nodes (Entity and Predicate)
    NodeE = node{id: EName, type: entity},
    NodeP = node{id: PName, type: predicate},

    % Define Edge (Connection between Entity and Predicate)
    Edge = edge{source: EName, target: PName},
    process_facts(T, NodesT, EdgesT).

% Utility to ensure unique nodes and apply the 'inconsistent' flag
deduplicate_and_mark(NodesList, InconsistentEntities, FinalNodes) :-
    % Apply the inconsistency marking to the raw list
    maplist(mark_node(InconsistentEntities), NodesList, MarkedNodes),
    % Use sort/2 to remove duplicates (Prolog standard way)
    sort(0, @<, MarkedNodes, FinalNodes).

mark_node(InconsistentEntities, Node, MarkedNode) :-
    % Check if the node's ID (the entity name) is in the list of conflicts
    ( member(Node.id, InconsistentEntities) ->
        MarkedNode = Node.put(inconsistent, true)
    ;
        MarkedNode = Node.put(inconsistent, false)
    ).

</script>

<script>
    let prolog;
    const outputDiv = document.getElementById('output');

    // Initialize SWIPL-WASM
    (async function() {
        prolog = await SWIPL({
            arguments: ["-q"],
            // Redirect Prolog output to the web console
            print: (text) => {
                outputDiv.innerHTML += text;
                outputDiv.scrollTop = outputDiv.scrollHeight; // Auto-scroll
            },
            on_error: (text) => outputDiv.innerHTML += 'ERROR: ' + text + '\n',
```

```
    });

    // Load the Prolog code into the WASM virtual filesystem
    const code = document.getElementById('cognitionCode').textContent;
    prolog.FS.writeFile('/home/web_user/model.pl', code);
    prolog.call('consult(model).');

    outputDiv.innerHTML += 'Prolog engine ready. Visualization initialized.\n';
    updateVisualization();
})();

// Function to handle user input
async function introduceInformation() {
    const fact = document.getElementById('newFact').value.trim();
    if (!fact) return;

    outputDiv.innerHTML += `\n> User introducing: ${fact}\n`;

    // Call the 'learn' predicate which handles the equilibration process
    const query = `learn(${fact}).`;
    try {
        prolog.call(query);
        updateVisualization();
    } catch (e) {
        outputDiv.innerHTML += `Error executing query: ${e}\n`;
    }
    document.getElementById('newFact').value = ''; // Clear input
}

// Function to fetch the current cognitive structure from Prolog
async function updateVisualization() {
    if (!prolog) return;

    const query = "get_graph_data(Nodes, Edges).";
    try {
        // Query Prolog and process the results
        const result = prolog.query(query).once();

        if (result) {
            // Convert Prolog data structures (lists of dicts) to JavaScript arrays of objects
            const nodes = Array.from(result.Nodes).map(n => Object.fromEntries(n));
            const edges = Array.from(result.Edges).map(e => Object.fromEntries(e));

            drawGraph(nodes, edges);
        }
    } catch (e) {
        console.error("Error querying graph data:", e);
    }
}

// D3.js Force-Directed Graph Rendering Logic
function drawGraph(nodes, links) {
    const svg = d3.select("#visualization svg");
    svg.selectAll("*").remove(); // Clear previous graph

    const width = svg.node().getBoundingClientRect().width;
    const height = svg.node().getBoundingClientRect().height;

    // Create the force simulation
```

```javascript
const simulation = d3.forceSimulation(nodes)
    .force("link", d3.forceLink(links).id(d => d.id).distance(120))
    .force("charge", d3.forceManyBody().strength(-350))
    .force("center", d3.forceCenter(width / 2, height / 2))
    .force("collision", d3.forceCollide().radius(30));

// Draw links (relationships)
const link = svg.append("g")
    .selectAll("line")
    .data(links)
    .enter().append("line")
    .attr("class", "link");

// Draw nodes (concepts/entities)
const node = svg.append("g")
    .selectAll("circle")
    .data(nodes)
    .enter().append("circle")
    .attr("r", 15)
    // Apply CSS classes based on node type and inconsistency status
    .attr("class", d => {
        let classes = `node-${d.type}`;
        // If the node is involved in a conflict, highlight it
        if (d.inconsistent) {
            classes += " inconsistent";
        }
        return classes;
    })
    // Enable dragging functionality
    .call(d3.drag()
        .on("start", dragstarted)
        .on("drag", dragged)
        .on("end", dragended));

// Draw labels
const label = svg.append("g")
    .selectAll("text")
    .data(nodes)
    .enter().append("text")
    .attr("x", 20)
    .attr("y", 5)
    .text(d => d.id)
    .style("font-size", "14px")
    .style("pointer-events", "none");

// Update positions on simulation tick (animation loop)
simulation.on("tick", () => {
    link
        .attr("x1", d => d.source.x)
        .attr("y1", d => d.source.y)
        .attr("x2", d => d.target.x)
        .attr("y2", d => d.target.y);

    node
        .attr("cx", d => d.x)
        .attr("cy", d => d.y);

    label
        .attr("transform", d => `translate(${d.x}, ${d.y})`);
```

```
        });

        // Drag event handlers
        function dragstarted(event, d) {
            if (!event.active) simulation.alphaTarget(0.3).restart();
            d.fx = d.x;
            d.fy = d.y;
        }

        function dragged(event, d) {
            d.fx = event.x;
            d.fy = event.y;
        }

        function dragended(event, d) {
            if (!event.active) simulation.alphaTarget(0);
            d.fx = null;
            d.fy = null;
        }
    }
</script>
</body>
</html>
```

3 config.pl

```
/** <module> System Configuration
 *
 * This module defines configuration parameters for the ORR (Observe,
 * Reorganize, Reflect) system. These parameters control the behavior of the
 * cognitive cycle, such as resource limits.
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(config, [
    max_inferences/1,
    max_retries/1
    ]).

%!      max_inferences(?Limit:integer) is nondet.
%
%       Defines the maximum number of inference steps the meta-interpreter
%       is allowed to take before a `resource_exhaustion` perturbation is
%       triggered.
%
%       This is a key parameter for learning. It is intentionally set to a
%       low value to make inefficient strategies (like the initial `add/3`
%       implementation) fail, thus creating a "disequilibrium" that the
%       system must resolve through reorganization.
%
%       This predicate is dynamic, so it can be changed at runtime if needed.
:- dynamic max_inferences/1.
max_inferences(15).

%!      max_retries(?Limit:integer) is nondet.
%
%       Defines the maximum number of times the system will attempt to
%       reorganize and retry a goal after a failure. This prevents infinite
```

```
%       loops if the system is unable to find a stable, coherent solution.
%
%       This predicate is dynamic.
:- dynamic max_retries/1.
max_retries(5).
```

4 counting2.pl

```
/** <module> Deterministic Pushdown Automaton for Counting
 *
 * This module implements a Deterministic Pushdown Automaton (DPDA) that
 * simulates the cognitive process of counting from 0 up to a specified number.
 * It models how units, tens, and hundreds are incremented and "carry over,"
 * similar to an odometer.
 *
 * The automaton's configuration is represented by `pda(State, Stack)`. The
 * stack is used to store the current count, with separate atoms for the
 * units, tens, and hundreds places (e.g., `['U5', 'T2', 'H1', '#']` for 125).
 * The input to the automaton is a series of `tick` events, each causing the
 * counter to increment by one.
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(counting2,
          [ run_counter/2
          ]).

:- use_module(library(lists)).

%!      run_counter(+N:integer, -FinalValue:integer) is det.
%
%       Runs the counting automaton for `N` steps and returns the final value.
%
%       This predicate generates an input list of `N` `tick` atoms,
%       initializes the DPDA, runs the simulation, and then converts the
%       final stack configuration back into an integer result.
%
%       @param N The number of times to "tick" the counter, effectively the
%       number to count up to.
%       @param FinalValue The integer value represented by the automaton's
%       stack after `N` increments.
run_counter(N, FinalValue) :-
    % Generate the input sequence of N 'tick' events.
    length(Input, N),
    maplist(=(tick), Input),

    % Initial DPDA configuration: start state with an empty stack marker.
    InitialPDA = pda(q_start, ['#']),

    % Run the DPDA simulation.
    run_pda(InitialPDA, Input, FinalPDA),

    % Convert the final stack configuration to an integer value.
    FinalPDA = pda(_, FinalStack),
    stack_to_int(FinalStack, FinalValue).

% run_pda(+PDA, +Input, -FinalPDA)
%
```

```prolog
% The main recursive loop that drives the automaton.
run_pda(PDA, [], PDA).
run_pda(PDA, [Input|Rest], FinalPDA) :-
    transition(PDA, Input, NextPDA),
    run_pda(NextPDA, Rest, FinalPDA).
run_pda(pda(State, Stack), [], pda(FinalState, FinalStack)) :-
    transition(pda(State, Stack), '', pda(FinalState, FinalStack)),
    \+ transition(pda(FinalState, FinalStack), '', _), % ensure it's a final epsilon transition
    !.

% transition(+CurrentPDA, +Input, -NextPDA)
%
% Defines the state transition rules for the counting automaton.

% Epsilon transition from start to initialize the counter stack.
transition(pda(q_start, ['#']), '', pda(q_idle, ['U0', 'T0', 'H0', '#'])).

% --- Unit Transitions ---
% If units are not 9, just increment the unit counter.
transition(pda(q_idle, [U|Rest]), tick, pda(q_idle, [NewU|Rest])) :-
    atom_concat('U', N_str, U), atom_number(N_str, N), N < 9, NewN is N + 1, atom_concat('U', NewN,
% If units are 9, transition to increment the tens place.
transition(pda(q_idle, ['U9'|Rest]), tick, pda(q_inc_tens, Rest)).

% --- Tens Transitions (Epsilon) ---
% After incrementing units from 9, reset units to 0 and increment tens.
transition(pda(q_inc_tens, [T|Rest]), '', pda(q_idle, ['U0', NewT|Rest])) :-
    atom_concat('T', N_str, T), atom_number(N_str, N), N < 9, NewN is N + 1, atom_concat('T', NewN,
% If tens are also 9, transition to increment the hundreds place.
transition(pda(q_inc_tens, ['T9'|Rest]), '', pda(q_inc_hundreds, Rest)).

% --- Hundreds Transitions (Epsilon) ---
% After incrementing tens from 9, reset units/tens and increment hundreds.
transition(pda(q_inc_hundreds, [H|Rest]), '', pda(q_idle, ['U0', 'T0', NewH|Rest])) :-
    atom_concat('H', N_str, H), atom_number(N_str, N), N < 9, NewN is N + 1, atom_concat('H', NewN,
% If hundreds are also 9, we have overflowed; halt.
transition(pda(q_inc_hundreds, ['H9'|Rest]), '', pda(q_halt, ['U0', 'T0', 'H0'|Rest])).


% stack_to_int(+Stack, -Value)
%
% Converts the final stack representation back into an integer.
stack_to_int(['U0', 'T0', 'H0', '#'], 0).
stack_to_int([U, T, H, '#'], Value) :-
    atom_concat('U', U_str, U), atom_number(U_str, U_val),
    atom_concat('T', T_str, T), atom_number(T_str, T_val),
    atom_concat('H', H_str, H), atom_number(H_str, H_val),
    Value is U_val + T_val * 10 + H_val * 100.
```

5 counting_on_back.pl

```prolog
/** <module> Bidirectional Counting Automaton (Up and Down)
 *
 * This module implements a Deterministic Pushdown Automaton (DPDA) that
 * simulates counting both forwards and backwards. It extends the functionality
 * of `counting2.pl` by handling two types of input events:
 * - `tick`: Increments the counter by one.
 * - `tock`: Decrements the counter by one.
 *
```

```prolog
 * The automaton manages carrying (for `tick`) and borrowing (for `tock`)
 * across units, tens, and hundreds places, which are stored on the stack.
 * This provides a more complex model of cognitive counting processes.
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(counting_on_back,
          [ run_counter/3
          ]).

:- use_module(library(lists)).

%!      run_counter(+StartN:integer, +Ticks:list, -FinalValue:integer) is det.
%
%       Runs the bidirectional counting automaton.
%
%       This predicate initializes the DPDA's stack to represent `StartN`,
%       then processes a list of `Ticks`, where each element is either `tick`
%       (increment) or `tock` (decrement). Finally, it converts the resulting
%       stack back into an integer.
%
%       @param StartN The integer value to start counting from.
%       @param Ticks A list of `tick` and `tock` atoms.
%       @param FinalValue The final integer value after processing all ticks.
run_counter(StartN, Ticks, FinalValue) :-
    % Set up initial stack from the starting number.
    H is StartN // 100,
    T is (StartN mod 100) // 10,
    U is StartN mod 10,
    atom_concat('U', U, US), atom_concat('T', T, TS), atom_concat('H', H, HS),
    InitialStack = [US, TS, HS, '#'],
    InitialPDA = pda(q_idle, InitialStack),

    % Run the DPDA with the list of ticks/tocks.
    run_pda(InitialPDA, Ticks, FinalPDA),

    % Convert the final stack configuration to an integer.
    FinalPDA = pda(_, FinalStack),
    stack_to_int(FinalStack, FinalValue).

% run_pda(+PDA, +Input, -FinalPDA)
%
% The main recursive loop that drives the automaton.
run_pda(PDA, [], PDA).
run_pda(PDA, [Input|Rest], FinalPDA) :-
    transition(PDA, Input, NextPDA),
    run_pda(NextPDA, Rest, FinalPDA).
run_pda(pda(State, Stack), [], pda(FinalState, FinalStack)) :-
    transition(pda(State, Stack), '', pda(FinalState, FinalStack)),
    \+ transition(pda(FinalState, FinalStack), '', _), % ensure it's a final epsilon transition
    !.

% transition(+CurrentPDA, +Input, -NextPDA)
%
% Defines the state transition rules for the up/down counter.

% --- Unit Transitions ---
% Increment (tick)
```

```
transition(pda(q_idle, [U|Rest]), tick, pda(q_idle, [NewU|Rest])) :-
    atom_concat('U', N_str, U), atom_number(N_str, N), N < 9, NewN is N + 1, atom_concat('U', NewN,
transition(pda(q_idle, ['U9'|Rest]), tick, pda(q_inc_tens, Rest)).
% Decrement (tock)
transition(pda(q_idle, [U|Rest]), tock, pda(q_idle, [NewU|Rest])) :-
    atom_concat('U', N_str, U), atom_number(N_str, N), N > 0, NewN is N - 1, atom_concat('U', NewN,
transition(pda(q_idle, ['U0'|Rest]), tock, pda(q_dec_tens, Rest)).


% --- Tens Transitions (Epsilon-driven) ---
% Carry from units
transition(pda(q_inc_tens, [T|Rest]), '', pda(q_idle, ['U0', NewT|Rest])) :-
    atom_concat('T', N_str, T), atom_number(N_str, N), N < 9, NewN is N + 1, atom_concat('T', NewN,
transition(pda(q_inc_tens, ['T9'|Rest]), '', pda(q_inc_hundreds, Rest)).
% Borrow from tens
transition(pda(q_dec_tens, [T|Rest]), '', pda(q_idle, ['U9', NewT|Rest])) :-
    atom_concat('T', N_str, T), atom_number(N_str, N), N > 0, NewN is N - 1, atom_concat('T', NewN,
transition(pda(q_dec_tens, ['T0'|Rest]), '', pda(q_dec_hundreds, Rest)).


% --- Hundreds Transitions (Epsilon-driven) ---
% Carry from tens
transition(pda(q_inc_hundreds, [H|Rest]), '', pda(q_idle, ['U0', 'T0', NewH|Rest])) :-
    atom_concat('H', N_str, H), atom_number(N_str, N), N < 9, NewN is N + 1, atom_concat('H', NewN,
transition(pda(q_inc_hundreds, ['H9'|Rest]), '', pda(q_halt, ['U0', 'T0', 'H0'|Rest])).
% Borrow from hundreds
transition(pda(q_dec_hundreds, [H|Rest]), '', pda(q_idle, ['U9', 'T9', NewH|Rest])) :-
    atom_concat('H', N_str, H), atom_number(N_str, N), N > 0, NewN is N - 1, atom_concat('H', NewN,
transition(pda(q_dec_hundreds, ['H0'|Rest]), '', pda(q_underflow, ['U9', 'T9', 'H9'|Rest])).


% stack_to_int(+Stack, -Value)
%
% Converts the final stack representation back into an integer.
stack_to_int(['U0', 'T0', 'H0', '#'], 0).
stack_to_int([U, T, H, '#'], Value) :-
    atom_concat('U', U_str, U), atom_number(U_str, U_val),
    atom_concat('T', T_str, T), atom_number(T_str, T_val),
    atom_concat('H', H_str, H), atom_number(H_str, H_val),
    Value is U_val + T_val * 10 + H_val * 100.
```

6 execution_handler.pl

```
/** <module> ORR Cycle Execution Handler
 *
 * This module serves as the central controller for the cognitive architecture,
 * managing the Observe-Reorganize-Reflect (ORR) cycle. It orchestrates the
 * interaction between the meta-interpreter (Observe), the reflective monitor
 * (Reflect), and the reorganization engine (Reorganize).
 *
 * The primary entry point is `run_query/1`, which initiates the ORR cycle
 * for a given goal.
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(execution_handler, [run_computation/2]).

:- use_module(meta_interpreter).
```

```prolog
:- use_module(reorganization_engine).
:- use_module(object_level).
:- use_module(more_machine_learner, [reflect_and_learn/1]).

%!      run_computation(+Goal:term, +Limit:integer) is semidet.
%
%       The main entry point for the self-reorganizing system. It attempts
%       to solve the given `Goal` within the specified `Limit` of
%       computational steps.
%
%       If the computation exceeds the resource limit, it triggers the
%       reorganization process and then retries the goal.
%
%       @param Goal The computational goal to be solved.
%       @param Limit The maximum number of inference steps allowed.
run_computation(Goal, Limit) :-
    catch(
        call_meta_interpreter(Goal, Limit, Trace),
        Error,
        handle_perturbation(Error, Goal, Trace, Limit)
    ).

%!      call_meta_interpreter(+Goal, +Limit, -Trace) is det.
%
%       A wrapper for the `meta_interpreter:solve/4` predicate. It
%       executes the goal and, upon success, reports that the computation
%       is complete.
%
%       @param Goal The goal to be solved.
%       @param Limit The inference limit.
%       @param Trace The resulting execution trace.
call_meta_interpreter(Goal, Limit, Trace) :-
    meta_interpreter:solve(Goal, Limit, _, Trace),
    writeln('Computation successful.'),
    reflect_on_success(Goal, Trace).

%!      normalize_trace(+Trace, -NormalizedTrace) is det.
%
%       Converts different trace formats into a unified dictionary format
%       for the learner. It specifically handles the `arithmetic_trace/3`
%       term, converting it to a `trace{}` dict.
% Case 1: The trace is a list containing a single arithmetic_trace term.
normalize_trace([arithmetic_trace(Strategy, _, Steps)], NormalizedTrace) :-
    !,
    NormalizedTrace = trace{strategy:Strategy, steps:Steps}.
% Case 2: The trace is a bare arithmetic_trace term.
normalize_trace(arithmetic_trace(Strategy, _, Steps), NormalizedTrace) :-
    !,
    NormalizedTrace = trace{strategy:Strategy, steps:Steps}.
% Case 3: Pass through any other format (already normalized dicts, etc.)
normalize_trace(Trace, Trace).

%!      reflect_on_success(+Goal, +Trace) is det.
%
%       After a successful computation, this predicate triggers the
%       reflective learning process. It passes the goal and the resulting
%       trace to the learning module to check for potential optimizations.
reflect_on_success(Goal, Trace) :-
    writeln('--- Proactive Reflection Cycle Initiated (Success) ---'),
```

```prolog
    normalize_trace(Trace, NormalizedTrace),
    Result = _{goal:Goal, trace:NormalizedTrace},
    reflect_and_learn(Result),
    writeln('--- Reflection Cycle Complete ---').

%!      handle_perturbation(+Error, +Goal, +Trace, +Limit) is semidet.
%
%       Catches errors from the meta-interpreter and initiates the
%       reorganization process.
%
%       This predicate specifically handles `perturbation(resource_exhaustion)`.
%       Upon catching this error, it logs the event, invokes the
%       `reorganization_engine`, and then recursively retries the original
%       goal with the same limit.
%
%       @param Error The error term thrown by `catch/3`.
%       @param Goal The original goal that was being attempted.
%       @param Trace The execution trace produced before the error occurred.
%       @param Limit The original resource limit.
handle_perturbation(perturbation(resource_exhaustion), Goal, Trace, Limit) :-
    writeln('Resource exhaustion detected. Initiating reorganization...'),
    % First, attempt to learn from the failure trace
    writeln('--- Reflective Cycle Initiated (Failure) ---'),
    normalize_trace(Trace, NormalizedTrace),
    Result = _{goal:Goal, trace:NormalizedTrace},
    reflect_and_learn(Result),
    % Then, proceed with the original reorganization logic
    reorganize_system(Goal, Trace),
    writeln('Reorganization complete. Retrying goal...'),
    run_computation(Goal, Limit).

handle_perturbation(Error, _, _, _) :-
    writeln('An unhandled error occurred:'),
    writeln(Error),
    fail.
```

7 hermeneutic_calculator.pl

```prolog
/** <module> Hermeneutic Calculator - Strategy Dispatcher
 *
 * This module acts as a high-level dispatcher for the various cognitive
 * strategy models implemented in the `sar_*` and `smr_*` modules. It provides
 * a unified interface to execute a calculation using a specific, named
 * strategy and to list the available strategies for each arithmetic operation.
 *
 * This allows the user interface or other components to abstract away the
 * details of individual strategy modules.
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(hermeneutic_calculator,
          [ calculate/6
          , list_strategies/2
          ]).

% Addition Strategies
:- use_module(sar_add_cobo).
:- use_module(sar_add_chunking).
```

```prolog
:- use_module(sar_add_rmb).
:- use_module(sar_add_rounding).

% Subtraction Strategies
:- use_module(sar_sub_cobo_missing_addend).
:- use_module(sar_sub_cbbo_take_away).
:- use_module(sar_sub_decomposition).
:- use_module(sar_sub_rounding).
:- use_module(sar_sub_sliding).
:- use_module(sar_sub_chunking_a).
:- use_module(sar_sub_chunking_b).
:- use_module(sar_sub_chunking_c).

% Multiplication Strategies
:- use_module(smr_mult_c2c).
:- use_module(smr_mult_cbo).
:- use_module(smr_mult_commutative_reasoning).
:- use_module(smr_mult_dr).

% Division Strategies
:- use_module(smr_div_cbo).
:- use_module(smr_div_dealing_by_ones).
:- use_module(smr_div_idp).
:- use_module(smr_div_ucr).

% Counting Automata
:- use_module(counting2).
:- use_module(counting_on_back).

% --- Strategy Lists ---

%!      list_strategies(+Op:atom, -Strategies:list) is nondet.
%
%       Provides a list of available strategy names for a given arithmetic
%       operator.
%
%       @param Op The operator (`+`, `-`, `*`, `/`).
%       @param Strategies A list of atoms representing the names of the
%       strategies available for that operator.
list_strategies(+, [
    'COBO',
    'Chunking',
    'RMB',
    'Rounding'
]).
list_strategies(-, [
    'COBO (Missing Addend)',
    'CBBO (Take Away)',
    'Decomposition',
    'Rounding',
    'Sliding',
    'Chunking A',
    'Chunking B',
    'Chunking C'
]).
list_strategies(*, [
    'C2C',
    'CBO',
    'Commutative Reasoning',
```

```prolog
    'DR'
]).
list_strategies(/, [
    'CBO (Division)',
    'Dealing by Ones',
    'IDP',
    'UCR'
]).

% --- Calculator Dispatch ---

%!      calculate(+Num1:integer, +Op:atom, +Num2:integer, +Strategy:atom, -Result:integer, -History:
%
%       Executes a calculation using a specified cognitive strategy.
%       This predicate acts as a dispatcher, calling the appropriate
%       `run_*` predicate from the various strategy modules based on the
%       `Strategy` name. It now captures and returns the execution trace.
%
%       @param Num1 The first operand.
%       @param Op The arithmetic operator (`+`, `-`, `*`, `/`).
%       @param Num2 The second operand.
%       @param Strategy The name of the strategy to use (must match one from
%       `list_strategies/2`).
%       @param Result The numerical result of the calculation. Fails if the
%       strategy does not complete successfully.
%       @param History A list of terms representing the execution trace of
%       the chosen strategy.
calculate(N1, +, N2, 'COBO', Result, History) :-
    run_cobo(N1, N2, Result, History).
calculate(N1, +, N2, 'Chunking', Result, History) :-
    run_chunking(N1, N2, Result, History).
calculate(N1, +, N2, 'RMB', Result, History) :-
    run_rmb(N1, N2, Result, History).
calculate(N1, +, N2, 'Rounding', Result, History) :-
    run_rounding(N1, N2, Result, History).

calculate(M, -, S, 'COBO (Missing Addend)', Result, History) :-
    run_cobo_ma(M, S, Result, History).
calculate(M, -, S, 'CBBO (Take Away)', Result, History) :-
    run_cbbo_ta(M, S, Result, History).
calculate(M, -, S, 'Decomposition', Result, History) :-
    run_decomposition(M, S, Result, History).
calculate(M, -, S, 'Rounding', Result, History) :-
    run_sub_rounding(M, S, Result, History).
calculate(M, -, S, 'Sliding', Result, History) :-
    run_sliding(M, S, Result, History).
calculate(M, -, S, 'Chunking A', Result, History) :-
    run_chunking_a(M, S, Result, History).
calculate(M, -, S, 'Chunking B', Result, History) :-
    run_chunking_b(M, S, Result, History).
calculate(M, -, S, 'Chunking C', Result, History) :-
    run_chunking_c(M, S, Result, History).

calculate(N, *, S, 'C2C', Result, History) :-
    run_c2c(N, S, Result, History).
calculate(N, *, S, 'CBO', Result, History) :-
    run_cbo_mult(N, S, 10, Result, History).
calculate(N, *, S, 'Commutative Reasoning', Result, History) :-
    run_commutative_mult(N, S, Result, History).
```

```
calculate(N, *, S, 'DR', Result, History) :-
    run_dr(N, S, Result, History).

calculate(T, /, S, 'CBO (Division)', Result, History) :-
    run_cbo_div(T, S, 10, Result, History).
calculate(T, /, N, 'Dealing by Ones', Result, History) :-
    run_dealing_by_ones(T, N, Result, History).
calculate(T, /, S, 'IDP', Result, History) :-
    % A default Knowledge Base is provided for demonstration.
    KB = [40-5, 16-2, 8-1],
    run_idp(T, S, KB, Result, History).
calculate(E, /, G, 'UCR', Result, History) :-
    run_ucr(E, G, Result, History).
```

8 incompatibility_semantics.pl

```
/** <module> Core logic for incompatibility semantics and automated theorem proving.
 *
 *  This module implements Robert Brandom's incompatibility semantics, providing a
 *  sequent calculus-based theorem prover. It integrates multiple knowledge
 *  domains, including geometry, number theory (Euclid's proof of the
 *  infinitude of primes), and arithmetic over natural numbers, integers, and
 *  rational numbers. The prover uses a combination of structural rules,
 *  material inferences (axioms), and reduction schemata to derive conclusions
 *  from premises.
 *
 *  Key features:
 *  - A sequent prover `proves/1` that operates on sequents of the form `Premises => Conclusions`.
 *  - A predicate `incoherent/1` to check if a set of propositions is contradictory.
 *  - Support for multiple arithmetic domains (n, z, q) via `set_domain/1`.
 *  - A rich set of logical operators and domain-specific predicates.
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(incompatibility_semantics,
          [ proves/1
          , is_recollection/2 % obj_coll/1 is deprecated
          , incoherent/1, set_domain/1, current_domain/1
          % Updated exports
          , s/1, o/1, n/1, comp_nec/1, exp_nec/1, exp_poss/1, comp_poss/1, neg/1
          , highlander/2, bounded_region/4, equality_iterator/3
          % Geometry
          , square/1, rectangle/1, rhombus/1, parallelogram/1, trapezoid/1, kite/1, quadrilateral/1
          , r1/1, r2/1, r3/1, r4/1, r5/1, r6/1
          % Number Theory (Euclid)
          , prime/1, composite/1, divides/2, is_complete/1
          % Fractions (Jason.pl)
          , rdiv/2, iterate/3, partition/3, normalize/2 % Export normalize
          ]).
% Declare predicates that are defined across different sections.
:- use_module(hermeneutic_calculator).

:- discontiguous proves_impl/2.
:- discontiguous is_incoherent/1. % Non-recursive check

% =================================================================
% Part 0: Setup and Configuration
% =================================================================
```

```prolog
% Define operators for modalities, negation, and sequents.
:- op(500, fx, comp_nec). % Compressive Necessity (Box_down)
:- op(500, fx, exp_nec).  % Expansive Necessity (Box_up)
:- op(500, fx, exp_poss). % Expansive Possibility (Diamond_up)
:- op(500, fx, comp_poss).% Compressive Possibility (Diamond_down)
:- op(500, fx, neg).
:- op(1050, xfy, =>).
:- op(550, xfy, rdiv). % Operator for rational numbers

% =================================================================
% Part 1: Knowledge Domains
% =================================================================

% --- 1.1 Geometry (Chapter 2) ---
incompatible_pair(square, r1). incompatible_pair(rectangle, r1). incompatible_pair(rhombus, r1). inc
incompatible_pair(square, r2). incompatible_pair(rhombus, r2). incompatible_pair(kite, r2).
incompatible_pair(square, r3). incompatible_pair(rectangle, r3). incompatible_pair(rhombus, r3). inc
incompatible_pair(square, r4). incompatible_pair(rhombus, r4). incompatible_pair(kite, r4).
incompatible_pair(square, r5). incompatible_pair(rectangle, r5). incompatible_pair(rhombus, r5). inc
incompatible_pair(square, r6). incompatible_pair(rectangle, r6).

is_shape(S) :- (incompatible_pair(S, _); S = quadrilateral), !.

entails_via_incompatibility(P, Q) :- P == Q, !.
entails_via_incompatibility(_, quadrilateral) :- !.
entails_via_incompatibility(P, Q) :- forall(incompatible_pair(Q, R), incompatible_pair(P, R)).

geometric_predicates([square, rectangle, rhombus, parallelogram, trapezoid, kite, quadrilateral, r1,

% --- 1.4 Fraction Domain (Jason.pl) ---
fraction_predicates([rdiv, iterate, partition]).

% --- 1.2 Arithmetic (O/N Domains) ---

:- dynamic current_domain/1.

%!      current_domain(?Domain:atom) is nondet.
%
%       Dynamic fact that holds the current arithmetic domain.
%       Possible values are `n` (natural numbers), `z` (integers),
%       or `q` (rational numbers).
%
%       @param Domain The current arithmetic domain.
current_domain(n).

%!      set_domain(+Domain:atom) is det.
%
%       Sets the current arithmetic domain.
%       Retracts the current domain and asserts the new one.
%       Valid domains are `n`, `z`, and `q`.
%
%       @param Domain The new arithmetic domain to set.
set_domain(D) :-
    % Added 'q' (Rationals) as a valid domain.
    ( member(D, [n, z, q]) -> retractall(current_domain(_)), assertz(current_domain(D)) ; true).


% Deprecated: obj_coll/1. Replaced by is_recollection/2.
```

```prolog
% The old obj_coll/1 predicate checked for static, timeless properties.
% The new ontology requires that a number's validity is proven by
% demonstrating a constructive history (an anaphoric recollection).
%
% obj_coll(N) :- current_domain(n), !, integer(N), N >= 0.
% obj_coll(N) :- current_domain(z), !, integer(N).
% obj_coll(X) :- current_domain(q), !,
%     ( integer(X)
%     ; (X = N rdiv D, integer(N), integer(D), D > 0)
%     ).

%!      is_recollection(?Term, ?History) is semidet.
%
%       The new core ontological predicate. It succeeds if `Term` is a
%       validly constructed number, where `History` is the execution
%       trace of the calculation that constructed it. This replaces the
%       static `obj_coll/1` check with a dynamic, process-based validation.
%
%       @param Term The numerical term to be validated (e.g., 5).
%       @param History The constructive trace that proves the term's existence.

% Base case: 0 is axiomatically a number.
is_recollection(0, [axiom(zero)]).

% Recursive case for positive integers: N is a recollection if N-1 is, and we
% can construct N by adding 1 using the hermeneutic calculator.
is_recollection(N, History) :-
    integer(N),
    N > 0,
    Prev is N - 1,
    is_recollection(Prev, _), % Foundational check on the predecessor
    hermeneutic_calculator:calculate(Prev, +, 1, _Strategy, N, History).

% Case for negative integers: A negative number is constructed by subtracting
% its absolute value from 0.
is_recollection(N, History) :-
    integer(N),
    N < 0,
    is_recollection(0, _), % Grounded in the axiom of zero
    Val is abs(N),
    hermeneutic_calculator:calculate(0, -, Val, _Strategy, N, History).

% Case for rational numbers: A rational N/D is a recollection if its
% numerator and denominator are themselves valid recollections.
% The history records this compositional validation.
is_recollection(N rdiv D, [history(rational, from(N, D))]) :-
    % Denominator must be a positive integer. We check its recollection status.
    is_recollection(D, _),
    integer(D), D > 0,
    % Numerator can be any recollected number.
    is_recollection(N, _).


% --- Helpers for Rational Arithmetic ---
gcd(A, 0, A) :- A \= 0, !.
gcd(A, B, G) :- B \= 0, R is A mod B, gcd(B, R, G).

%!      normalize(+Input, -Normalized) is det.
%
```

```prolog
%       Normalizes a number. Integers are unchanged. Rational numbers
%       (e.g., `6 rdiv 8`) are reduced to their simplest form (e.g., `3 rdiv 4`).
%       If the denominator is 1, it is converted to an integer.
%
%       @param Input The integer or rational number to normalize.
%       @param Normalized The resulting normalized number.
normalize(N, N) :- integer(N), !.
normalize(N rdiv D, R) :-
    (D =:= 1 -> R = N ;
        G is abs(gcd(N, D)),
        SN is N // G, % Integer division
        SD is D // G,
        (SD =:= 1 -> R = SN ; R = SN rdiv SD)
    ), !.

% Helper for dynamic arithmetic (FIX: Resolve syntax error)
perform_arith(+, A, B, C) :- C is A + B.
perform_arith(-, A, B, C) :- C is A - B.

% Helper for rational addition/subtraction (FIX: Resolve syntax error)
arith_op(A, B, Op, C) :-
    % Ensure Op is a valid arithmetic operator we handle here
    member(Op, [+, -]),
    normalize(A, NA), normalize(B, NB),
    (integer(NA), integer(NB) ->
        % Case 1: Integer Arithmetic
        % Use helper predicate to perform the operation
        perform_arith(Op, NA, NB, C_raw)
    ;
        % Case 2: Rational Arithmetic
        (integer(NA) -> N1=NA, D1=1 ; NA = N1 rdiv D1),
        (integer(NB) -> N2=NB, D2=1 ; NB = N2 rdiv D2),

        D_res is D1 * D2,
        N1_scaled is N1 * D2,
        N2_scaled is N2 * D1,

        perform_arith(Op, N1_scaled, N2_scaled, N_res),

        C_raw = N_res rdiv D_res
    ),
    normalize(C_raw, C).

% --- 1.3 Number Theory Domain (Euclid) ---

number_theory_predicates([prime, composite, divides, is_complete, analyze_euclid_number, member]).

% Combined list of excluded predicates for Arithmetic Evaluation
excluded_predicates(AllPreds) :-
    geometric_predicates(G),
    number_theory_predicates(NT),
    fraction_predicates(F),
    append(G, NT, Temp),
    append(Temp, F, DomainPreds),
    append([neg, conj, nec, comp_nec, exp_nec, exp_poss, comp_poss, is_recollection], DomainPreds, A

% --- Helpers for Number Theory (Grounded) ---

% Helper: Product of a list
```

```
product_of_list(L, P) :- (is_list(L) -> product_of_list_impl(L, P) ; fail).
product_of_list_impl([], 1).
product_of_list_impl([H|T], P) :- number(H), product_of_list_impl(T, P_tail), P is H * P_tail.

% Helper: Find a prime factor
find_prime_factor(N, F) :- number(N), N > 1, find_factor_from(N, 2, F).
find_factor_from(N, D, D) :- N mod D =:= 0, !.
find_factor_from(N, D, F) :-
    D * D =< N,
    (D =:= 2 -> D_next is 3 ; D_next is D + 2),
    find_factor_from(N, D_next, F).
find_factor_from(N, _, N). % N is prime

% Helper: Grounded check for primality
is_prime(N) :- number(N), N > 1, find_factor_from(N, 2, F), F =:= N.

% =================================================================
% Part 2: Core Logic Engine
% =================================================================

% Helper predicates
select(X, [X|T], T).
select(X, [H|T], [H|R]) :- select(X, T, R).

% Helper to match antecedents against premises (Allows unification)
match_antecedents([], _).
match_antecedents([A|As], Premises) :-
    member(A, Premises),
    match_antecedents(As, Premises).

% --- 2.1 Incoherence Definitions (SAFE AND COMPLETE) ---

%!      incoherent(+PropositionSet:list) is semidet.
%
%       Checks if a set of propositions is incoherent (contradictory).
%       A set is incoherent if:
%       1. It contains a direct contradiction (e.g., `P` and `neg(P)`).
%       2. It violates a material incompatibility (e.g., `n(square(a))` and `n(r1(a))`).
%       3. An empty conclusion `[]` can be proven from it, i.e., `proves(PropositionSet => [])`.
%
%       @param PropositionSet A list of propositions.
incoherent(X) :- is_incoherent(X), !.
incoherent(X) :- proves(X => []).

% is_incoherent/1: Non-recursive Incoherence Check

% --- 1. Specific Material Optimizations ---

% Geometric Incompatibility
is_incoherent(X) :-
    member(n(ShapePred), X), ShapePred =.. [Shape, V],
    member(n(RestrictionPred), X), RestrictionPred =.. [Restriction, V],
    ground(Shape), ground(Restriction),
    incompatible_pair(Shape, Restriction), !.

% Arithmetic Incompatibility (Generalized to handle fractions)
% This is incoherent if a norm demands an impossible recollection.
is_incoherent(X) :-
    member(n(minus(A,B,_)), X), % Check for the normative proposition
```

```
    current_domain(n),
    is_recollection(A, _), is_recollection(B, _), % Operands must be valid numbers
    normalize(A, NA), normalize(B, NB),
    NA < NB, !.

% M6-Case1: Euclid Case 1 Incoherence
is_incoherent(X) :-
    member(n(prime(EF)), X),
    member(n(is_complete(L)), X),
    product_of_list(L, DE),
    EF is DE + 1.

% --- 2. Base Incoherence (LNC) and Persistence ---

% Law of Non-Contradiction (LNC)
incoherent_base(X) :- member(P, X), member(neg(P), X).
incoherent_base(X) :- member(D_P, X), D_P =.. [D, P], member(D_NegP, X), D_NegP =.. [D, neg(P)], mem

% Persistence
is_incoherent(Y) :- incoherent_base(Y), !.


% --- 2.2 Sequent Calculus Prover (REORDERED) ---
% Order: Identity/Explosion -> Axioms -> Structural Rules -> Reduction Schemata.

%!      proves(+Sequent) is semidet.
%
%       Attempts to prove a given sequent using the rules of the calculus.
%       A sequent has the form `Premises => Conclusions`, where `Premises`
%       and `Conclusions` are lists of propositions. The predicate succeeds
%       if the conclusions can be derived from the premises.
%
%       The prover uses a recursive, history-tracked implementation (`proves_impl/2`)
%       to apply inference rules and avoid infinite loops.
%
%       @param Sequent The sequent to be proven.
proves(Sequent) :- proves_impl(Sequent, []).

% --- PRIORITY 1: Identity and Explosion ---

% Axiom of Identity (A |- A)
proves_impl((Premises => Conclusions), _) :-
    member(P, Premises), member(P, Conclusions), !.

% From base incoherence (Explosion)
proves_impl((Premises => _), _) :-
    is_incoherent(Premises), !.

% --- PRIORITY 2: Material Inferences and Grounding (Axioms) ---

% --- Arithmetic Grounding (Extended for Q) ---
proves_impl(_ => [o(eq(A,B))], _) :-
    is_recollection(A, _), is_recollection(B, _),
    normalize(A, NA), normalize(B, NB),
    NA == NB.

proves_impl(_ => [o(plus(A,B,C))], _) :-
    is_recollection(A, _), is_recollection(B, _),
    arith_op(A, B, +, C),
```

```prolog
    is_recollection(C, _).

proves_impl(_ => [o(minus(A,B,C))], _) :-
    current_domain(D), is_recollection(A, _), is_recollection(B, _),
    arith_op(A, B, -, C),
    % Subtraction constraints only apply to N. We must normalize C before comparison.
    normalize(C, NC),
    ((D=n, NC >= 0) ; member(D, [z, q])),
    is_recollection(C, _).

% --- Arithmetic Material Inferences ---
proves_impl([n(plus(A,B,C))] => [n(plus(B,A,C))], _).

% --- EML Material Inferences (Axioms) - UPDATED ---
% Commitment 2: Emergence of Awareness (Temporal Compression)
proves_impl([s(u)] => [s(comp_nec a)], _).
proves_impl([s(u_prime)] => [s(comp_nec a)], _).

% Commitment 3 (Revised): The Tension of Awareness (Choice Point)
proves_impl([s(a)] => [s(exp_poss lg)], _). % Possibility of Release
proves_impl([s(a)] => [s(comp_poss t)], _).  % Possibility of Fixation (Temptation)

% Commitment 4: Dynamics of the Choice
% 4a: Fixation (Deepened Contraction)
proves_impl([s(t)] => [s(comp_nec neg(u))], _).
% 4b: Release (Sublation)
proves_impl([s(lg)] => [s(exp_nec u_prime)], _).

% Hegel's Triad Oscillation:
proves_impl([s(t_b)] => [s(comp_nec t_n)], _).
proves_impl([s(t_n)] => [s(comp_nec t_b)], _).

% --- 3.5 Fraction Grounding (Jason.pl integration) ---

% Grounding: Iterating (Multiplication)
proves_impl(([] => [o(iterate(U, M, R))]), _) :-
    is_recollection(U, _), integer(M), M >= 0,
    % R = U * M
    normalize(U, NU),
    (integer(NU) -> N1=NU, D1=1 ; NU = N1 rdiv D1),
    N_res is N1 * M,
    % D_res = D1,
    normalize(N_res rdiv D1, R).

% Grounding: Partitioning (Division)
proves_impl(([] => [o(partition(W, N, U))]), _) :-
    is_recollection(W, _), integer(N), N > 0,
    % U = W / N
    normalize(W, NW),
    (integer(NW) -> N1=NW, D1=1 ; NW = N1 rdiv D1),
    % N_res = N1,
    D_res is D1 * N,
    normalize(N1 rdiv D_res, U).

% --- Number Theory Material Inferences ---

% M5-Revised: Euclid's Core Argument (For Forward Chaining)
proves_impl(( [n(prime(G)), n(divides(G, N)), n(is_complete(L))] => [n(neg(member(G, L)))] ), _) :-
    product_of_list(L, P),
```

```
    N is P + 1.

% M5-Direct: (For Direct proof, where L is bound by the conclusion)
proves_impl(( [n(prime(G)), n(divides(G, N))] => [n(neg(member(G, L)))] ), _) :-
    product_of_list(L, P),
    N is P + 1.

% M4-Revised: Definition of Completeness Violation (For Forward Chaining)
proves_impl(([n(prime(G)), n(neg(member(G, L))), n(is_complete(L))] => [n(neg(is_complete(L)))]), _)

% M4-Direct: (For Direct proof)
proves_impl(([n(prime(G)), n(neg(member(G, L)))] => [n(neg(is_complete(L)))]), _).

% Grounding Primality
proves_impl(([] => [n(prime(N))]), _) :- is_prime(N).
proves_impl(([] => [n(composite(N))]), _) :- number(N), N > 1, \+ is_prime(N).


% --- PRIORITY 3: Structural Rules (Domain Specific and General) ---
% (Structural rules remain the same)

% Geometric Entailment (Inferential Strength)
proves_impl((Premises => Conclusions), _) :-
    member(n(P_pred), Premises), P_pred =.. [P_shape, X], is_shape(P_shape),
    member(n(Q_pred), Conclusions), Q_pred =.. [Q_shape, X], is_shape(Q_shape),
    entails_via_incompatibility(P_shape, Q_shape), !.

% Structural Rule for EML Dynamics - UPDATED
proves_impl((Premises => Conclusions), History) :-
    select(s(P), Premises, RestPremises), \+ member(s(P), History),
    eml_axiom(s(P), s(M_Q)),
    % Case 1: Necessities drive state transition
    ( (M_Q = comp_nec Q ; M_Q = exp_nec Q) -> proves_impl(([s(Q)|RestPremises] => Conclusions), [s(P
    % Case 2: Possibilities are checked against conclusions (for direct proofs) - Updated
    ; ((M_Q = exp_poss _ ; M_Q = comp_poss _), (member(s(M_Q), Conclusions) ; member(M_Q, Conclusion
    ).

% --- Structural Rules for Euclid's Proof ---

% Structural Rule: Euclid's Construction
proves_impl((Premises => Conclusions), History) :-
    member(n(is_complete(L)), Premises),
    \+ member(euclid_construction(L), History),
    product_of_list(L, DE),
    EF is DE + 1,
    NewPremise = n(analyze_euclid_number(EF, L)),
    proves_impl(([NewPremise|Premises] => Conclusions), [euclid_construction(L)|History]).

% Case Analysis Rule (Handles analyze_euclid_number)
proves_impl((Premises => Conclusions), History) :-
    select(n(analyze_euclid_number(EF, L)), Premises, RestPremises),
    EF > 1,
    (member(n(is_complete(L)), Premises) ->
        % Case 1: Assume EF is prime
        proves_impl(([n(prime(EF))|RestPremises] => Conclusions), History),
        % Case 2: Assume EF is composite
        proves_impl(([n(composite(EF))|RestPremises] => Conclusions), History)
    ; fail
    ).
```

```prolog
% Structural Rule: Prime Factorization (Existential Instantiation) (Case 2)
proves_impl((Premises => Conclusions), History) :-
    select(n(composite(N)), Premises, RestPremises),
    \+ member(factorization(N), History),
    find_prime_factor(N, G),
    NewPremises = [n(prime(G)), n(divides(G, N))|RestPremises],
    proves_impl((NewPremises => Conclusions), [factorization(N)|History]).

% --- General Structural Rule: Forward Chaining (Modus Ponens / MMP) ---
proves_impl((Premises => Conclusions), History) :-
    Module = incompatibility_semantics,
    % 1. Find an applicable material inference rule (axiom) defined in Priority 2.
    clause(Module:proves_impl((A_clause => [C_clause]), _), B_clause),

    copy_term((A_clause, C_clause, B_clause), (Antecedents, Consequent, Body)),
    is_list(Antecedents), % Handle grounding axioms like [] => P

    % 2. Check if the antecedents are satisfied by the current premises.
    match_antecedents(Antecedents, Premises),
    % 3. Execute the body of the axiom.
    call(Module:Body),
    % 4. Ensure the consequent hasn't already been derived.
    \+ member(Consequent, Premises),
    % 5. Add the consequent to the premises and continue.
    proves_impl(([Consequent|Premises] => Conclusions), History).


% Arithmetic Evaluation (Legacy support for simple integer evaluation in sequents)
proves_impl(([Premise|RestPremises] => Conclusions), History) :-
    (Premise =.. [Index, Expr], member(Index, [s, o, n]) ; (Index = none, Expr = Premise)),
    (compound(Expr) -> (
        functor(Expr, F, _),
        excluded_predicates(Excluded),
        \+ member(F, Excluded)
    ) ; true),
    % Ensure the expression is not a rational structure before using 'is'
    \+ (compound(Expr), functor(Expr, rdiv, 2)),
    catch(Value is Expr, _, fail), !,
    (Index \= none -> NewPremise =.. [Index, Value] ; NewPremise = Value),
    proves_impl(([NewPremise|RestPremises] => Conclusions), History).


% --- PRIORITY 4: Reduction Schemata (Logical Connectives) ---

% Left Negation (LN)
proves_impl((P => C), H) :- select(neg(X), P, P1), proves_impl((P1 => [X|C]), H).
proves_impl((P => C), H) :- select(D_NegX, P, P1), D_NegX=..[D, neg(X)], member(D,[s,o,n]), D_X=..[D

% Right Negation (RN)
proves_impl((P => C), H) :- select(neg(X), C, C1), proves_impl(([X|P] => C1), H).
proves_impl((P => C), H) :- select(D_NegX, C, C1), D_NegX=..[D, neg(X)], member(D,[s,o,n]), D_X=..[D

% Conjunction (Generalized)
proves_impl((P => C), H) :- select(conj(X,Y), P, P1), proves_impl(([X,Y|P1] => C), H).
proves_impl((P => C), H) :- select(s(conj(X,Y)), P, P1), proves_impl(([s(X),s(Y)|P1] => C), H).

proves_impl((P => C), H) :- select(conj(X,Y), C, C1), proves_impl((P => [X|C1]), H), proves_impl((P
proves_impl((P => C), H) :- select(s(conj(X,Y)), C, C1), proves_impl((P => [s(X)|C1]), H), proves_im
```

```
% S5 Modal rules (Generalized)
proves_impl((P => C), H) :- select(nec(X), P, P1), !, ( proves_impl((P1 => C), H) ; \+ proves_impl((
proves_impl((P => C), H) :- select(nec(X), C, C1), !, ( proves_impl((P => C1), H) ; proves_impl(([]

% (Helpers for EML Dynamics)
eml_axiom(A, C) :-
    clause(incompatibility_semantics:proves_impl(([A] => [C]), _), true),
    is_eml_modality(C).

is_eml_modality(s(comp_nec _)).
is_eml_modality(s(exp_nec _)).
is_eml_modality(s(exp_poss _)).
is_eml_modality(s(comp_poss _)).

% =================================================================
% Part 4: Automata and Placeholders
% =================================================================

%!      highlander(+List:list, -Result) is semidet.
%
%       Succeeds if the `List` contains exactly one element, which is unified with `Result`.
%       "There can be only one."
%
%       @param List The input list.
%       @param Result The single element of the list.
highlander([Result], Result) :- !.
highlander([], _) :- !, fail.
highlander([_|Rest], Result) :- highlander(Rest, Result).

%!      bounded_region(+I:number, +L:number, +U:number, -R:term) is det.
%
%       Checks if a number `I` is within a given lower `L` and upper `U` bound.
%
%       @param I The number to check.
%       @param L The lower bound.
%       @param U The upper bound.
%       @param R `in_bounds(I)` if `L =< I =< U`, otherwise `out_of_bounds(I)`.
bounded_region(I, L, U, R) :- ( number(I), I >= L, I =< U -> R = in_bounds(I) ; R = out_of_bounds(I)

%!      equality_iterator(?C:integer, +T:integer, -R:integer) is nondet.
%
%       Iterates from a counter `C` up to a target `T`.
%       Unifies `R` with `T` when `C` reaches `T`.
%
%       @param C The current value of the counter.
%       @param T The target value.
%       @param R The result, unified with T on success.
equality_iterator(T, T, T) :- !.
equality_iterator(C, T, R) :- C < T, C1 is C + 1, equality_iterator(C1, T, R).

% Placeholder definitions for exported functors
%! s(P) is det.
% Wrapper for subjective propositions.
s(_).
%! o(P) is det.
% Wrapper for objective propositions.
o(_).
%! n(P) is det.
```

```
% Wrapper for normative propositions.
n(_).
%! neg(P) is det.
% Wrapper for negation.
neg(_).
%! comp_nec(P) is det.
% Compressive necessity modality.
comp_nec(_).
%! exp_nec(P) is det.
% Expansive necessity modality.
exp_nec(_).
%! exp_poss(P) is det.
% Expansive possibility modality.
exp_poss(_).
%! comp_poss(P) is det.
% Compressive possibility modality.
comp_poss(_).
%! square(X) is det.
% Geometric shape placeholder.
square(_).
%! rectangle(X) is det.
% Geometric shape placeholder.
rectangle(_).
%! rhombus(X) is det.
% Geometric shape placeholder.
rhombus(_).
%! parallelogram(X) is det.
% Geometric shape placeholder.
parallelogram(_).
%! trapezoid(X) is det.
% Geometric shape placeholder.
trapezoid(_).
%! kite(X) is det.
% Geometric shape placeholder.
kite(_).
%! quadrilateral(X) is det.
% Geometric shape placeholder.
quadrilateral(_).
%! r1(X) is det.
% Geometric restriction placeholder.
r1(_).
%! r2(X) is det.
% Geometric restriction placeholder.
r2(_).
%! r3(X) is det.
% Geometric restriction placeholder.
r3(_).
%! r4(X) is det.
% Geometric restriction placeholder.
r4(_).
%! r5(X) is det.
% Geometric restriction placeholder.
r5(_).
%! r6(X) is det.
% Geometric restriction placeholder.
r6(_).
%! prime(N) is det.
% Number theory placeholder for prime numbers.
prime(_).
```

```
%! composite(N) is det.
% Number theory placeholder for composite numbers.
composite(_).
%! divides(A, B) is det.
% Number theory placeholder for divisibility.
divides(_, _).
%! is_complete(L) is det.
% Number theory placeholder for a complete list of primes.
is_complete(_).
%! analyze_euclid_number(N, L) is det.
% Placeholder for Euclid's proof step.
analyze_euclid_number(_, _).
%! rdiv(N, D) is det.
% Placeholder for rational number representation (Numerator rdiv Denominator).
rdiv(_, _).
%! iterate(U, M, R) is det.
% Placeholder for iteration/multiplication of fractions.
iterate(_, _, _).
%! partition(W, N, U) is det.
% Placeholder for partitioning/division of fractions.
partition(_, _, _).
```

9 index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Synthesis Explorer: Brandom, CGI, Piaget</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <header>
        <h1>Synthesis Explorer</h1>
        <p>Incompatibility Semantics, Cognitively Guided Instruction, and Constructivism</p>
    </header>

    <div class="container">
        <div class="tabs">
            <button class="tab-button active" onclick="openTab(event, 'CGI')">Strategy Analyzer (CGI
            <button class="tab-button" onclick="openTab(event, 'Explorer')">Concept Explorer (Brando
        </div>

        <div id="CGI" class="tab-content active">
            <h2>Strategy Analyzer</h2>
            <p>Analyze a student's problem-solving strategy to understand their cognitive structure
            <div class="input-group">
                <label for="problemContext">Problem Context:</label>
                <select id="problemContext">
                    <option value="Math-JRU">Math: Join (Result Unknown) e.g., 5 + 3 = ?</option>
                    <option value="Math-JCU">Math: Join (Change Unknown) e.g., 5 + ? = 8</option>
                    <option value="Science-Float">Science: Sink or Float Prediction</option>
                </select>
            </div>
            <div class="input-group">
                <label for="strategyInput">Observed Strategy/Reasoning:</label>
                <textarea id="strategyInput" rows="4" placeholder="Describe how the student solved t
            </div>
```

```html
            <button onclick="analyzeCGI()">Analyze Strategy</button>
            <div id="cgiResult" class="results">
                <i>Analysis results will appear here.</i>
            </div>
        </div>

        <div id="Explorer" class="tab-content">
            <h2>Concept Explorer</h2>
            <p>Enter a statement to explore its semantic content based on what it excludes (incompat
            <div class="input-group">
                <label for="conceptInput">Statement:</label>
                <input type="text" id="conceptInput" placeholder="e.g., The object is red">
            </div>
            <button onclick="analyzeIncompatibility()">Analyze</button>
            <div id="incompatibilityResult" class="results">
                <i>Analysis results will appear here.</i>
            </div>
        </div>

    </div>

    <script src="script.js"></script>
</body>
</html>
```

10 interactive_ui.pl

```prolog
/** <module> Interactive Command-Line UI for the More Machine Learner
 *
 * This module provides a text-based, interactive user interface for the
 * "More Machine Learner" system. It allows a user to:
 *
 * - Trigger the learning of new addition strategies from examples.
 * - Trigger a critique of existing rules using challenging subtraction problems.
 * - View the strategies that have been learned during the session.
 * - Load and save learned knowledge from a file (`learned_knowledge.pl`).
 *
 * The main entry point is `start/0`, which initializes the system and
 * displays the main menu.
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(interactive_ui, [start/0]).

:- use_module(more_machine_learner).

% --- Main Entry Point ---

%!      start is det.
%
%       The main entry point for the interactive user interface.
%
%       This predicate displays a welcome message, asks the user if they want
%       to load previously saved knowledge, and then enters the main menu loop
%       where the user can select different actions.
start :-
    welcome_message,
    ask_to_load_knowledge,
```

```prolog
    main_menu.

% --- Interactive UI Predicates ---

welcome_message :-
    nl,
    writeln('===================================================='),
    writeln('          Welcome to the More Machine Learner       '),
    writeln('===================================================='),
    writeln('All I can do is count, but I can learn from what you show me.'),
    nl.

ask_to_load_knowledge :-
    write('Do you want to load previously learned strategies? (y/n) > '),
    read_line_to_string(user_input, Response),
    (   (Response = "y" ; Response = "Y")
    ->  (   exists_file('learned_knowledge.pl')
        ->  writeln('Loading previously learned knowledge...'),
            consult('learned_knowledge.pl')
        ;   writeln('No saved knowledge file found.')
        )
    ;   writeln('Starting with a clean slate.')
    ).

main_menu :-
    nl,
    writeln('--- Main Menu ---'),
    writeln('1. Learn a new addition strategy (e.g., from 8+5=13)'),
    writeln('2. Critique a normative rule (e.g., from 3-5=-2)'),
    writeln('3. Show currently learned strategies'),
    writeln('4. Save learned strategies'),
    writeln('5. Exit'),
    write('> '),
    read_line_to_string(user_input, Choice),
    handle_menu_choice(Choice).

handle_menu_choice("1") :- !, run_learning_interaction, main_menu.
handle_menu_choice("2") :- !, run_critique_interaction, main_menu.
handle_menu_choice("3") :- !, show_learned_strategies, main_menu.
handle_menu_choice("4") :- !, save_knowledge, main_menu.
handle_menu_choice("5") :- !, writeln('Goodbye!'), nl.
handle_menu_choice(_) :- writeln('Invalid choice, please try again.'), main_menu.

run_learning_interaction :-
    nl,
    writeln('--- Learning a New Strategy ---'),
    writeln('Please provide a basic addition problem and its result.'),
    write('Example: 8+5=13'), nl,
    write('Problem > '),
    read_line_to_string(user_input, ProblemString),
    (   parse_problem(ProblemString, +(A,B), Result)
    ->  bootstrap_from_observation(+(A,B), Result)
    ;   writeln('Invalid problem format. Please use the format "A+B=C".')
    ).

run_critique_interaction :-
    nl,
    writeln('--- Critiquing a Norm ---'),
    writeln('Please provide a challenging subtraction problem.'),
```

```
    write('Example: 3-5=-2'), nl,
    write('Problem > '),
    read_line_to_string(user_input, ProblemString),
    (   parse_problem(ProblemString, -(A,B), Result)
    ->  critique_and_bootstrap(minus(A, B, Result))
    ;   writeln('Invalid problem format. Please use the format "A-B=C".')
    ).

show_learned_strategies :-
    nl,
    writeln('--- Learned Strategies ---'),
    (   current_predicate(learned_strategy/1)
    ->  listing(learned_strategy/1)
    ;   writeln('No strategies have been learned in this session.')
    ),
    nl.

% --- Parsing Helper ---
parse_problem(String, Term, Result) :-
    normalize_space(string(CleanString), String),
    atomic_list_concat(Parts, '=', CleanString),
    (   Parts = [Problem, ResultStr]
    ->  normalize_space(string(TrimmedResult), ResultStr),
        number_string(Result, TrimmedResult),
        (   atomic_list_concat([A_str, B_str], '+', Problem)
        ->  normalize_space(string(TrimmedA), A_str),
            normalize_space(string(TrimmedB), B_str),
            number_string(A, TrimmedA),
            number_string(B, TrimmedB),
            Term = +(A,B)
        ;   atomic_list_concat([A_str, B_str], '-', Problem)
        ->  normalize_space(string(TrimmedA), A_str),
            normalize_space(string(TrimmedB), B_str),
            number_string(A, TrimmedA),
            number_string(B, TrimmedB),
            Term = -(A,B)
        ;   fail
        )
    ;   fail
    ).
```

11 jason.pl

```
/** <module> Jason's Partitive Fractional Schemes
 *
 * This module implements a computational model of Jason's partitive
 * fractional schemes, as described in cognitive science literature on
 * mathematical development. It models how a student might conceptualize
 * and operate on fractions by partitioning, disembedding, and iterating units.
 *
 * The core data structure is a `unit(Value, History)` term, which tracks
 * both a rational numerical value and its operational history.
 *
 * The module defines two main strategic state machines:
 * 1.  **Partitive Fractional Scheme (PFS)**: Models the process of finding
 *     a simple fraction (e.g., 3/7) of a whole.
 * 2.  **Fractional Composition Scheme (FCS)**: Models the more complex process
 *     of finding a fraction of a fraction (e.g., 3/4 of 1/4), which involves
 *     a "metamorphic accommodation" where the result of one operation becomes
```

```
 *      the input for the next.
 *
 * The primary entry point for demonstration is `run_tests/0`.
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(jason, [run_tests/0, debug_run_fcs/0]).
:- (   catch(use_module(library(rat)), E, (format('[jason] Optional library "rat" not available: ~w~

% =============================================================================
% I. Cognitive Material Representation (ContinuousUnit)
% =============================================================================
%
% We represent a ContinuousUnit as a compound term: unit(Value, History).
% - Value: A rational number (e.g., 1, 3 rdiv 7).
% - History: A string representing the operational history.

% =============================================================================
% II. Iterative Core: Explicitly Nested Number Sequence (ENS) Operations
% =============================================================================

% ens_partition(+UnitIn, +N, -PartitionedWhole)
% Divides a continuous unit into N equal parts.
ens_partition(unit(Value, History), N, PartitionedWhole) :-
    N > 0,
    NewValue is Value / N,
    format(string(NewHistory), '1/~w part of (~w)', [N, History]),
    length(PartitionedWhole, N),
    maplist(=(unit(NewValue, NewHistory)), PartitionedWhole).

% ens_disembed(+PartitionedWhole, -UnitFraction)
% Isolates a single unit part from the partitioned whole.
ens_disembed([UnitFraction | _], UnitFraction) :- !.
ens_disembed([], _) :- throw(error(cannot_disembed_from_empty_list, _)).

% ens_iterate(+UnitIn, +M, -ResultUnit)
% Repeats a unit M times.
ens_iterate(unit(Value, History), M, unit(NewValue, NewHistory)) :-
    NewValue is Value * M,
    format(string(NewHistory), '~w iterations of [~w]', [M, History]).

% =============================================================================
% III. Strategic Shell: The Partitive Fractional Scheme (PFS)
% =============================================================================

%!      run_pfs(+Whole:unit, +Numerator:integer, +Denominator:integer, -Result:unit, -Trace:list) is
%
%       Executes the Partitive Fractional Scheme to calculate `Num/Den` of `Whole`.
%
%       This state machine models the cognitive process of:
%       1. Partitioning the `Whole` into `Denominator` equal parts.
%       2. Disembedding one of those parts (the unit fraction).
%       3. Iterating the unit fraction `Numerator` times.
%
%       @param Whole The initial `unit/2` term to be operated on.
%       @param Numerator The numerator of the fraction.
%       @param Denominator The denominator of the fraction.
%       @param Result The final `unit/2` term representing the result.
```

```prolog
%       @param Trace A list of strings describing the cognitive steps taken.
run_pfs(Whole, Num, Den, Result, Trace) :-
    % Initialize V (variables) in a dict
    V0 = v{whole: Whole, n: Den, m: Num},
    ( Whole = unit(WholeVal, _) -> true ; WholeVal = Whole ),
    format(string(Log0), 'PFS Initialized: Find ~w/~w of ~w', [Num, Den, WholeVal]),

    % Start the state machine loop with an accumulator for logs
    pfs_loop(q_start, V0, Result, [Log0], Trace).

% pfs_loop/5 uses Acc as accumulator and Trace as final output
pfs_loop(q_accept, V, Result, Acc, TraceOut) :-
    ( get_dict(result, V, Result) -> true ; Result = V ),
    reverse(Acc, RevAcc),
    append(RevAcc, ["PFS Complete."], TraceOut).
pfs_loop(CurrentState, V_in, Result, Acc, TraceOut) :-
    pfs_transition(CurrentState, V_in, NextState, V_out, Log),
    pfs_loop(NextState, V_out, Result, [Log|Acc], TraceOut).

% pfs_transition(+State, +V_in, -NextState, -V_out, -Log)
% Defines the state transitions (delta function)
pfs_transition(q_start, V, q_partition, V, "Transition to partition state") :- !.

pfs_transition(q_partition, V_in, q_disembed, V_out, Log) :-
    format(string(Log), '[State: q_partition] Action: Partitioning Whole into ~w parts.', [V_in.n]),
    ens_partition(V_in.whole, V_in.n, Partitioned),
    V_out = V_in.put(partitioned_whole, Partitioned),
    !.

pfs_transition(q_disembed, V_in, q_iterate, V_out, Log) :-
    ens_disembed(V_in.partitioned_whole, UnitFraction),
    ( UnitFraction = unit(UVal, _) -> true ; UVal = UnitFraction ),
    format(string(Log), '[State: q_disembed] Action: Disembedded Unit Fraction (~w).', [UVal]),
    V_out = V_in.put(unit_fraction, UnitFraction),
    !.

pfs_transition(q_iterate, V_in, q_accept, V_out, Log) :-
    format(string(Log), '[State: q_iterate] Action: Iterating Unit Fraction ~w times.', [V_in.m]),
    ens_iterate(V_in.unit_fraction, V_in.m, Result),
    V_out = V_in.put(result, Result),
    !.

% ============================================================================
% IV. Strategic Shell: The Fractional Composition Scheme (FCS)
% ============================================================================

%!      run_fcs(+Whole:unit, +OuterFrac:pair, +InnerFrac:pair, -Result:unit, -Trace:list) is det.
%
%       Executes the Fractional Composition Scheme to calculate a fraction of a fraction.
%       It solves `(A/B) of (C/D)` of `Whole`.
%
%       This state machine models a more advanced cognitive process involving
%       "metamorphic accommodation," where the result of one fractional operation
%       becomes the new "whole" for the next fractional operation. It achieves
%       this by calling `run_pfs/5` as a subroutine.
%
%       @param Whole The initial `unit/2` term.
%       @param OuterFrac A pair `A-B` for the outer fraction.
%       @param InnerFrac A pair `C-D` for the inner fraction.
```

```
%       @param Result The final `unit/2` term.
%       @param Trace A nested list describing the cognitive steps, including the
%       trace of the inner `run_pfs/5` calls.
run_fcs(Whole, A-B, C-D, Result, Trace) :-
    % Compose two PFS computations: inner then outer.
    format(string(Log0), 'FCS Initialized: Find ~w/~w of ~w/~w of whole', [A,B,C,D]),
    (   catch(run_pfs(Whole, C, D, IntermediateResult, InnerTrace), E, (format('Error computing inne
    ->  true
    ;   fail
    ),
    format(string(AccLog), '-> Intermediate Result: ~w', [IntermediateResult]),
    (   catch(run_pfs(IntermediateResult, A, B, FinalResult, OuterTrace), E2, (format('Error computi
    ->  true
    ;   fail
    ),
    Result = FinalResult,
    Trace = [log(q_start, Log0, []), log(q_inner_PFS, AccLog, InnerTrace), log(q_accommodate, '[acco

% =============================================================================
% V. Demonstration and Testing
% =============================================================================

%!      run_tests is det.
%
%       The main demonstration predicate for this module.
%
%       It runs two tests:
%       1. A test of the basic Partitive Fractional Scheme (PFS).
%       2. A test of the more complex Fractional Composition Scheme (FCS),
%          which demonstrates recursive partitioning.
%
%       It prints detailed execution traces for both tests to the console.
run_tests :-
    writeln('=== JASON AUTOMATON MODEL TESTING ==='),

    % Define the initial Whole
    TheWhole = unit(1, "Reference Unit"),

    % --- Test 1: Partitive Fractional Scheme (PFS) ---
    writeln('\n' + '============================================================'),
    writeln('TEST 1: Construct 3/7 of the Whole (PFS)'),
    writeln('============================================================'),
    run_pfs(TheWhole, 3, 7, ResultPFS, TracePFS),
    writeln('\nExecution Trace (Cognitive Choreography):'),
    print_pfs_trace(TracePFS),
    format('~nRESULT (PFS): ~w~n', [ResultPFS]),

    % --- Test 2: Fractional Composition Scheme (FCS) ---
    writeln('\n' + '============================================================'),
    writeln('TEST 2: Construct 3/4 of 1/4 of the Whole (FCS)'),
    writeln('Modeling Metamorphic Accommodation (Recursive Partitioning)'),
    writeln('============================================================'),
    run_fcs(TheWhole, 3-4, 1-4, ResultFCS, TraceFCS),
    writeln('\nExecution Trace (Cognitive Choreography):'),
    print_fcs_trace(TraceFCS, ""),
    format('~nRESULT (FCS): ~w~n', [ResultFCS]).

% Helper to print the flat trace from PFS
print_pfs_trace(Trace) :-
```

```
    forall(member(Line, Trace), writeln(Line)).

% Helper to print the potentially nested trace from FCS
print_fcs_trace([], _).
print_fcs_trace([log(State, Action, NestedTrace)|Rest], Indent) :-
    format('~wState: ~w, Action: ~w~n', [Indent, State, Action]),
    ( NestedTrace \= [] ->
        format('~w  [Begin Nested PFS Execution]~n', [Indent]),
        atom_concat(Indent, '    ', NewIndent),
        % Since PFS trace is flat list of strings
        forall(member(Line, NestedTrace), format('~w~w~n', [NewIndent, Line])),
        format('~w  [End Nested PFS Execution]~n', [Indent])
    ; true
    ),
    print_fcs_trace(Rest, Indent).

%! debug_run_fcs is det.
%  Debug helper: run a representative FCS calculation and print canonical result and trace.
debug_run_fcs :-
    TheWhole = unit(1, "Reference Unit"),
    V0 = v{whole: TheWhole, a:3, b:4, c:1, d:4},
    format('Debug: V0=~w~n', [V0]),
    ( fcs_transition(q_start, V0, NS1, V1, Log1, NT1) -> format('q_start -> ~w ; Log=~w NT=~w~n', [N
    ( fcs_transition(q_inner_PFS, V0, NS2, V2, Log2, NT2) -> (format('q_inner_PFS -> ~w ; Log=~w NT=
    ( fcs_transition(q_accommodate, V0, NS3, V3, Log3, NT3) -> format('q_accommodate -> ~w ; Log=~w
    ( fcs_transition(q_outer_PFS, V0, NS4, V4, Log4, NT4) -> (format('q_outer_PFS -> ~w ; Log=~w NT=
```

12 learned_knowledge.pl

```
/** <module> Learned Knowledge Base (Auto-Generated)
 *
 *  DO NOT EDIT THIS FILE MANUALLY.
 *
 *  This file serves as the persistent memory for the `more_machine_learner`.
 *  It stores the clauses for the dynamic predicate `run_learned_strategy/5`
 *  that the system discovers and validates through its generative-reflective
 *  exploration process.
 *
 *  The contents of this file are automatically generated by the
 *  `save_knowledge/0` predicate in `more_machine_learner.pl` and are
 *  loaded automatically when the system starts. Any manual edits will be
 *  overwritten.
 *
 * @author More Machine Learner (Auto-Generated)
 * @license MIT
 */

% Automatically generated knowledge base.
:- op(550, xfy, rdiv).
run_learned_strategy(A, B, C, rmb(10), D) :-
    integer(A),
    integer(B),
    A>0,
    A<10,
    E is 10-A,
    B>=E,
    F is B-E,
    C is 10+F,
    D=trace{a_start:A, b_start:B, steps:[step(A, 10), step(10, C)], strategy:rmb(10)}.
```

```
run_learned_strategy(A, B, C, doubles, D) :-
    integer(A),
    A==B,
    C is A*2,
    D=trace{a_start:A, b_start:B, steps:[rote(C)], strategy:doubles}.
run_learned_strategy(A, B, C, cob, D) :-
    integer(A),
    integer(B),
    (   A>=B
    ->  E=A,
        F=B,
        G=no_swap
    ;   E=B,
        F=A,
        G=swapped(B, A)
    ),
    (   G=swapped(_, _)
    ->  (   proves(([n(plus(A, B, H))]=>[n(plus(B, A, H))]))
        ->  true
        ;   fail
        )
    ;   true
    ),
    solve_foundationally(E, F, C, I),
    D=trace{a_start:A, b_start:B, steps:[G, inner_trace(I)], strategy:cob}.
```

13 learner_1.pl

```
/** <module> Simple Strategy Resolution Model
 *
 *  This module provides a simple, self-contained model for resolving the
 *  outputs from multiple, potentially conflicting, information sources or
 *  "strategies". It is a conceptual demonstration and is not integrated with
 *  the main ORR cycle or the other learner modules.
 *
 *  The model consists of two parts:
 *  1.  A database of facts (`strategy_output/4`) that simulates the results
 *      produced by different named strategies for various problems.
 *  2.  A `compute/3` predicate that gathers all possible results for a given
 *      problem and uses a `resolve/2` helper to determine the final outcome
 *      based on a simple semantic:
 *      - If all strategies agree, that is the result.
 *      - If strategies disagree, the result is `incompatible`.
 *      - If no strategy can solve the problem, the result is `unknown`.
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(learner_1, [compute/3]).

:- use_module(library(lists)).

% --- A. DATABASE OF STRATEGY OUTPUTS ---
% This section simulates the results from different strategies.
% Format: strategy_output(StrategyName, Operation, InputsList, Result).

% Case 1: Agreement
% Both strategy_a and strategy_b correctly compute 2 + 3.
strategy_output(strategy_a, add, [2, 3], 5).
```

```
strategy_output(strategy_b, add, [2, 3], 5).

% Case 2: Incompatibility (Disagreement)
% strategy_a correctly computes 5 - 1, but strategy_c gets it wrong.
strategy_output(strategy_a, subtract, [5, 1], 4).
strategy_output(strategy_c, subtract, [5, 1], 6). % Incorrect result

% Case 3: Single Available Strategy
% Only strategy_b knows how to multiply.
strategy_output(strategy_b, multiply, [10, 2], 20).

% --- B. RULES FOR COMPUTATION ---
% This section implements the logic to compute a final result.

% resolve(+ListOfResults, -FinalResult)
% This helper predicate applies the semantics to a list of gathered results.

% Rule 1: If the list of results is empty, the answer is 'unknown'.
resolve([], unknown).

% Rule 2: If the list of results contains different values, it's 'incompatible'.
% We convert the list to a set. If the set has more than one member, there was a disagreement.
resolve(ResultsList, incompatible) :-
    list_to_set(ResultsList, Set),
    length(Set, L),
    L > 1.

% Rule 3: If the list of results contains one or more identical values, that is the answer.
% After converting to a set, there will be exactly one element.
resolve(ResultsList, Result) :-
    list_to_set(ResultsList, Set),
    length(Set, 1),
    [Result] = Set.

%!      compute(+Op:atom, +Inputs:list, -Result) is det.
%
%       Computes the result for a given operation and inputs by polling all
%       available strategies and resolving their outputs.
%
%       It uses `findall/3` to collect all possible results from the
%       `strategy_output/4` database for the given `Op` and `Inputs`. It then
%       passes this list of results to `resolve/2` to determine the final,
%       semantically coherent result.
%
%       @param Op The operation to perform (e.g., `add`, `subtract`).
%       @param Inputs A list of input numbers for the operation.
%       @param Result The final resolved result, which can be a number,
%       the atom `incompatible`, or the atom `unknown`.
compute(Op, Inputs, Result) :-
    % Step 1: Find all results from all available strategies for the given problem.
    findall(R, strategy_output(_, Op, Inputs, R), ResultsList),

    % Step 2: Resolve the collected list of results using our semantics.
    resolve(ResultsList, Result).
```

14 main.pl

```
/** <module> Main Entry Point for Command-Line Execution
 *
```

```
 * This module provides a simple, non-interactive entry point for running the
 * cognitive modeling system from the command line. It is primarily used for
 * testing and demonstration purposes.
 *
 * When executed, it invokes the ORR (Observe, Reorganize, Reflect) cycle
 * with a predefined goal and prints the final result to the console.
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- use_module(execution_handler).

%!      main is det.
%
%       The main predicate for command-line execution.
%
%       It runs a predefined query, `add(5, 5, X)`, using the `run_computation/2`
%       predicate from the `execution_handler`. This triggers the full ORR
%       cycle. After the cycle completes, it prints the final result for `X`
%       and halts the Prolog system. The number 5 is represented using
%       Peano arithmetic (`s(s(s(s(s(0)))))`).
main :-
    % Use a reasonable inference step limit so the ORR cycle can trigger
    % reorganization if resource exhaustion occurs.
    Limit = 30,
    Goal = add(s(s(s(s(s(0))))), s(s(s(s(s(0))))), X),
    execution_handler:run_computation(Goal, Limit),
    format('Final Result (may be unbound if not solved): ~w~n', [X]),
    halt.

% This directive makes it so that running the script from the command line
% will automatically call the main/0 predicate.
:- initialization(main, main).
```

15 meta_interpreter.pl

```
/** <module> Embodied Tracing Meta-Interpreter
 *
 * This module provides the core "Observe" capability of the ORR cycle.
 * It contains a stateful meta-interpreter, `solve/4`, which executes goals
 * defined in the `object_level` module.
 *
 * This version is "embodied": it maintains a `ModalContext` (e.g., neutral,
 * compressive, expansive) that alters its reasoning behavior. For example,
 * in a `compressive` context, the cost of inferences increases, simulating
 * cognitive tension and narrowing the search. This context is switched when
 * the interpreter encounters modal operators defined in `incompatibility_semantics`.
 *
 * It produces a detailed `Trace` of the execution, which is the primary
 * data source for the `reflective_monitor`.
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(meta_interpreter, [solve/4]).
:- use_module(object_level). % Ensure we can access the object-level code
:- use_module(hermeneutic_calculator). % For strategic choice
:- use_module(incompatibility_semantics, [s/1, comp_nec/1, comp_poss/1, exp_nec/1, exp_poss/1]). % F
```

```prolog
% Note: is_list/1 is a built-in, no need to import from library(lists).

% --- Embodied Cognition Helpers ---

%!      is_modal_operator(?Goal, ?ModalContext) is semidet.
%
%       Identifies an embodied modal operator and maps it to a context.
is_modal_operator(comp_nec(_), compressive).
is_modal_operator(comp_poss(_), compressive).
is_modal_operator(exp_nec(_), expansive).
is_modal_operator(exp_poss(_), expansive).

%!      get_inference_cost(+ModalContext, -Cost) is det.
%
%       Determines the inference cost based on the current modal context.
%       - `compressive`: Cost is 2 (cognitive narrowing).
%       - `neutral`, `expansive`: Cost is 1.
get_inference_cost(compressive, 2).
get_inference_cost(expansive, 1).
get_inference_cost(neutral, 1).


% --- Arithmetic Goal Handling ---

%!      is_arithmetic_goal(?Goal, ?Op) is semidet.
%
%       Identifies arithmetic goals and maps them to standard operators.
%       This allows the meta-interpreter to intercept these goals and
%       handle them with the Hermeneutic Calculator instead of the
%       inefficient object-level definitions.
is_arithmetic_goal(add(_,_,_), +).
is_arithmetic_goal(multiply(_,_,_), *).
% Add other operations like subtract/3, divide/3 here if needed.


%!      peano_to_int(?Peano, ?Int) is det.
%
%       Converts a Peano number (s(s(0))) to an integer.
peano_to_int(0, 0).
peano_to_int(s(P), I) :-
    peano_to_int(P, I_prev),
    I is I_prev + 1.

%!      int_to_peano(?Int, ?Peano) is det.
%
%       Converts an integer to a Peano number.
int_to_peano(0, 0).
int_to_peano(I, s(P)) :-
    I > 0,
    I_prev is I - 1,
    int_to_peano(I_prev, P).


%!      solve(+Goal, +InferencesIn, -InferencesOut, -Trace) is nondet.
%
%       Public wrapper for the stateful meta-interpreter.
%       Initializes the `ModalContext` to `neutral` and calls the
%       internal `solve/6` predicate.
solve(Goal, I_In, I_Out, Trace) :-
```

```prolog
    solve(Goal, neutral, _, I_In, I_Out, Trace).


%!      solve(+Goal, +CtxIn, -CtxOut, +I_In, -I_Out, -Trace) is nondet.
%
%       The core stateful, embodied meta-interpreter.
%
%       @param Goal The goal to be solved.
%       @param CtxIn The current `ModalContext`.
%       @param CtxOut The `ModalContext` after the goal is solved.
%       @param I_In The initial number of available inference steps.
%       @param I_Out The remaining number of inference steps.
%       @param Trace A list representing the execution trace.
%       @error perturbation(resource_exhaustion) if inference counter drops to zero.

% Base case: `true` always succeeds. Context is unchanged.
solve(true, Ctx, Ctx, I, I, []) :- !.

% Modal Operator: Detect a modal operator, switch context for the sub-proof,
% and restore it upon completion.
solve(s(ModalGoal), CtxIn, CtxIn, I_In, I_Out, [modal_trace(ModalGoal, Ctx, SubTrace)]) :-
    is_modal_operator(ModalGoal, Ctx),
    !,
    ModalGoal =.. [_, InnerGoal],
    % The context is switched for the InnerGoal, but restored to CtxIn afterward.
    solve(InnerGoal, Ctx, _, I_In, I_Out, SubTrace).

% Conjunction: Solve `A` then `B`. The context flows from `A` to `B`.
solve((A, B), CtxIn, CtxOut, I_In, I_Out, [trace(A, A_Trace), trace(B, B_Trace)]) :-
    !,
    solve(A, CtxIn, CtxMid, I_In, I_Mid, A_Trace),
    solve(B, CtxMid, CtxOut, I_Mid, I_Out, B_Trace).

% System predicates: Use context-dependent cost. Context is unchanged.
solve(Goal, Ctx, Ctx, I_In, I_Out, [call(Goal)]) :-
    predicate_property(Goal, built_in),
    !,
    get_inference_cost(Ctx, Cost),
    check_viability(I_In, Cost),
    I_Out is I_In - Cost,
    call(Goal).

% Arithmetic predicates: Use context-dependent cost. Context is unchanged.
solve(Goal, Ctx, Ctx, I_In, I_Out, [arithmetic_trace(Strategy, Result, History)]) :-
    is_arithmetic_goal(Goal, Op),
    !,
    get_inference_cost(Ctx, Cost),
    check_viability(I_In, Cost),
    I_Out is I_In - Cost,
    Goal =.. [_, Peano1, Peano2, PeanoResult],
    peano_to_int(Peano1, N1),
    peano_to_int(Peano2, N2),
    list_strategies(Op, Strategies),
    ( is_list(Strategies), Strategies = [Strategy|_] -> true ; throw(error(no_strategy_found(Op), _)
    calculate(N1, Op, N2, Strategy, Result, History),
    int_to_peano(Result, PeanoResult).

% Object-level predicates: Use context-dependent cost. Context flows through sub-proof.
solve(Goal, CtxIn, CtxOut, I_In, I_Out, [clause(object_level:(Goal:-Body)), trace(Body, BodyTrace)])
```

```prolog
    get_inference_cost(CtxIn, Cost),
    check_viability(I_In, Cost),
    I_Mid is I_In - Cost,
    clause(object_level:Goal, Body),
    solve(Body, CtxIn, CtxOut, I_Mid, I_Out, BodyTrace).

% Failure case: If a goal is not a built-in and has no matching clauses,
% record the failure. Context is unchanged.
solve(Goal, Ctx, Ctx, I, I, [fail(Goal)]) :-
    \+ predicate_property(Goal, built_in),
    \+ (Goal = s(_), functor(Goal, s, 1)), % Don't fail on modal operators here
    \+ clause(object_level:Goal, _), !.


% --- Viability Check ---

% check_viability(+Inferences, +Cost)
%
% Succeeds if the inference counter is sufficient for the next step's cost.
check_viability(I, Cost) :- I >= Cost, !.
check_viability(_, _) :-
    % Constraint violated: PERTURBATION DETECTED
    throw(perturbation(resource_exhaustion)).
```

16 more_machine_learner.pl

```prolog
/** <module> More Machine Learner (Protein Folding Analogy)
 *
 * This module implements a machine learning system inspired by protein folding,
 * where a system seeks a lower-energy, more efficient state. It learns new,
 * more efficient arithmetic strategies by observing the execution traces of
 * less efficient ones.
 *
 * The core components are:
 * 1.  **A Foundational Solver**: The most basic, inefficient way to solve a
 *     problem (e.g., counting on by ones). This is the "unfolded" state.
 * 2.  **A Strategy Hierarchy**: A dynamic knowledge base of `run_learned_strategy/5`
 *     clauses. The system always tries the most "folded" (efficient) strategies first.
 * 3.  **A Generative-Reflective Loop (`explore/1`)**:
 *     - **Generative Phase**: Solves a problem using the current best strategy.
 *     - **Reflective Phase**: Analyzes the execution trace of the solution,
 *       looking for patterns that suggest a more efficient strategy (a "fold").
 * 4.  **Pattern Detection & Construction**: Specific predicates that detect
 *     patterns (e.g., commutativity, making a 10) and construct new, more
 *     efficient strategy clauses. These new clauses are then asserted into
 *     the knowledge base.
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(more_machine_learner,
          [ critique_and_bootstrap/1,
            run_learned_strategy/5,
            solve/4,
            save_knowledge/0,
            reflect_and_learn/1
          ]).

% Use the semantics engine for validation
```

```prolog
:- use_module(incompatibility_semantics, [proves/1, set_domain/1, current_domain/1, is_recollection/
:- use_module(library(random)).
:- use_module(library(lists)).

% Ensure operators are visible
:- op(1050, xfy, =>).
:- op(500, fx, neg).
:- op(550, xfy, rdiv).

%!      run_learned_strategy(?A, ?B, ?Result, ?StrategyName, ?Trace) is nondet.
%
%       A dynamic, multifile predicate that stores the collection of learned
%       strategies. Each clause of this predicate represents a single, efficient
%       strategy that the system has discovered and validated.
%
%       The `solve/4` predicate queries this predicate first, implementing a
%       hierarchy where learned, efficient strategies are preferred over
%       foundational, inefficient ones.
%
%       @param A The first input number.
%       @param B The second input number.
%       @param Result The result of the calculation.
%       @param StrategyName An atom identifying the learned strategy (e.g., `cob`, `rmb(10)`).
%       @param Trace A structured term representing the efficient execution path.
:- dynamic run_learned_strategy/5.

% ===================================================================
% Part 0: Initialization and Persistence
% ===================================================================

knowledge_file('learned_knowledge.pl').

% Load persistent knowledge when this module is loaded.
load_knowledge :-
    knowledge_file(File),
    (   exists_file(File)
    ->  consult(File),
        findall(_, clause(run_learned_strategy(_,_,_,_,_), _), Clauses),
        length(Clauses, Count),
        format('~N[Learner Init] Successfully loaded ~w learned strategies.~n', [Count])
    ;   format('~N[Learner Init] Knowledge file not found. Starting fresh.~n')
    ).

% Ensure initialization runs after the predicate is defined
:- initialization(load_knowledge, now).

%!      save_knowledge is det.
%
%       Saves all currently learned strategies (clauses of the dynamic
%       `run_learned_strategy/5` predicate) to the file specified by
%       `knowledge_file/1`. This allows for persistence of learning across sessions.
save_knowledge :-
    knowledge_file(File),
    setup_call_cleanup(
        open(File, write, Stream),
        (
            writeln(Stream, '% Automatically generated knowledge base.'),
            writeln(Stream, ':- op(550, xfy, rdiv).'),
            forall(clause(run_learned_strategy(A, B, R, S, T), Body),
```

```
                    portray_clause(Stream, (run_learned_strategy(A, B, R, S, T) :- Body)))
        ),
        close(Stream)
    ).

% =================================================================
% Part 1: The Unified Solver (Strategy Hierarchy)
% =================================================================

%!      solve(+A, +B, -Result, -Trace) is semidet.
%
%       Solves `A + B` using a strategy hierarchy.
%
%       It first attempts to use a highly efficient, learned strategy by
%       querying `run_learned_strategy/5`. If no applicable learned strategy
%       is found, it falls back to the foundational, inefficient counting
%       strategy (`solve_foundationally/4`).
%
%       @param A The first addend.
%       @param B The second addend.
%       @param Result The numerical result.
%       @param Trace The execution trace produced by the winning strategy.
solve(A, B, Result, Trace) :-
    (   run_learned_strategy(A, B, Result, _StrategyName, Trace)
    ->  true
    ;
        solve_foundationally(A, B, Result, Trace)
    ).

% =================================================================
% Part 2: Reflection and Learning
% =================================================================

%!      reflect_and_learn(+Result:dict) is semidet.
%
%       The core reflective learning trigger. It analyzes a computation's
%       result, which includes the goal and execution trace, to find
%       opportunities for creating more efficient strategies.
%
%       @param Result A dict containing at least `goal` and `trace`.
reflect_and_learn(Result) :-
    Goal = Result.goal,
    Trace = Result.trace,
    % We only learn from addition, and only if we have a trace.
    (   nonvar(Trace), Goal = add(A, B, _)
    ->  (   writeln('    (Reflecting on addition trace...)'),
            (   detect_cob_pattern(Trace, _),
                construct_and_validate_cob(A, B)
            ;   detect_rmb_pattern(Trace, RMB_Data),
                construct_and_validate_rmb(A, B, RMB_Data)
            ;   detect_doubles_pattern(Trace, _),
                construct_and_validate_doubles(A, B)
            ;   true % Succeed even if no new strategy is found
            )
        )
    ;   true % Succeed if not an addition goal or no trace
    ).

% =================================================================
```

```
% Part 3: Foundational Abilities & Trace Analysis
% =================================================================

% --- 3.1 Foundational Ability: Counting ---

successor(X, Y) :- proves([] => [o(plus(X, 1, Y))]).

% solve_foundationally(+A, +B, -Result, -Trace)
%
% The most basic, "unfolded" strategy. It solves addition by counting on
% from A, B times. This is deliberately inefficient to provide rich traces
% for the reflective process to analyze.
solve_foundationally(A, B, Result, Trace) :-
    is_recollection(A, _), is_recollection(B, _),
    integer(A), integer(B), B >= 0,
    count_loop(A, B, Result, Steps),
    Trace = trace{a_start:A, b_start:B, strategy:counting, steps:Steps}.

count_loop(CurrentA, 0, CurrentA, []) :- !.
count_loop(CurrentA, CurrentB, Result, [step(CurrentA, NextA)|Steps]) :-
    CurrentB > 0,
    NextB is CurrentB - 1,
    successor(CurrentA, NextA),
    count_loop(NextA, NextB, Result, Steps).

% --- 3.2 Trace Analysis Helpers ---

count_trace_steps(Trace, Count) :-
    (   member(Trace.strategy, [counting, doubles, rmb(_)])
    ->  length(Trace.steps, Count)
    ;   Trace.strategy = cob
    ->
        ( member(inner_trace(InnerTrace), Trace.steps)
          -> count_trace_steps(InnerTrace, Count)
          ; Count = 0
        )
    ;   Count = 1
    ).

get_calculation_trace(T, T) :- member(T.strategy, [counting, rmb(_), doubles]).
get_calculation_trace(T, CT) :-
    T.strategy = cob,
    member(inner_trace(InnerT), T.steps),
    get_calculation_trace(InnerT, CT).

% =================================================================
% Part 4: Pattern Detection & Construction
% =================================================================

% Detects if an inefficient counting strategy was used where commutativity (A+B = B+A) would have be
detect_cob_pattern(Trace, cob_data) :-
    Trace.strategy = counting,
    A = Trace.a_start, B = Trace.b_start,
    integer(A), integer(B),
    A < B.

% Constructs and validates a new "Counting On Bigger" (COB) strategy clause.
construct_and_validate_cob(A, B) :-
    StrategyName = cob,
```

```prolog
    StrategyHead = run_learned_strategy(A_in, B_in, Result, StrategyName, Trace),
    StrategyBody = (
        integer(A_in), integer(B_in),
        (A_in >= B_in -> Start = A_in, Count = B_in, Swap = no_swap ; Start = B_in, Count = A_in, Sw
        (   Swap = swapped(_, _) ->
            (proves([n(plus(A_in, B_in, R_temp))] => [n(plus(B_in, A_in, R_temp))]) -> true ; fail)
            ; true
        ),
        solve_foundationally(Start, Count, Result, InnerTrace),
        Trace = trace{a_start:A_in, b_start:B_in, strategy:StrategyName, steps:[Swap, inner_trace(In
    ),
    validate_and_assert(A, B, StrategyHead, StrategyBody).


% Detects if the counting trace shows a pattern of "making a ten".
detect_rmb_pattern(TraceWrapper, rmb_data{k:K, base:Base}) :-
    get_calculation_trace(TraceWrapper, Trace),
    Trace.strategy = counting,
    Base = 10,
    A = Trace.a_start, B = Trace.b_start,
    integer(A), integer(B),
    A > 0, A < Base, K is Base - A, B >= K,
    nth1(K, Trace.steps, Step),
    Step = step(_, Base).

% Constructs and validates a new "Rearranging to Make Bases" (RMB) strategy.
construct_and_validate_rmb(A, B, RMB_Data) :-
    Base = RMB_Data.base,
    StrategyName = rmb(Base),
    StrategyHead = run_learned_strategy(A_in, B_in, Result, StrategyName, Trace),
    StrategyBody = (
        integer(A_in), integer(B_in),
        A_in > 0, A_in < Base, K_runtime is Base - A_in, B_in >= K_runtime,
        B_new_runtime is B_in - K_runtime,
        Result is Base + B_new_runtime,
        Trace = trace{a_start:A_in, b_start:B_in, strategy:StrategyName, steps:[step(A_in, Base), st
    ),
    validate_and_assert(A, B, StrategyHead, StrategyBody).

% Detects if a problem was a "doubles" fact that was solved less efficiently.
detect_doubles_pattern(TraceWrapper, doubles_data) :-
    get_calculation_trace(TraceWrapper, Trace),
    member(Trace.strategy, [counting, rmb(_)]),
    A = Trace.a_start, B = Trace.b_start,
    A == B, integer(A).

% Constructs and validates a new "Doubles" strategy (rote knowledge).
construct_and_validate_doubles(A, B) :-
    StrategyName = doubles,
    StrategyHead = run_learned_strategy(A_in, B_in, Result, StrategyName, Trace),
    StrategyBody = (
        integer(A_in), A_in == B_in,
        Result is A_in * 2,
        Trace = trace{a_start:A_in, b_start:B_in, strategy:StrategyName, steps:[rote(Result)]}
    ),
    validate_and_assert(A, B, StrategyHead, StrategyBody).


% --- Validation Helper ---
```

```prolog
% Ensures a newly constructed strategy is sound before asserting it.
validate_and_assert(A, B, StrategyHead, StrategyBody) :-
    copy_term((StrategyHead, StrategyBody), (ValidationHead, ValidationBody)),
    arg(1, ValidationHead, A),
    arg(2, ValidationHead, B),
    arg(3, ValidationHead, CalculatedResult),
    arg(4, ValidationHead, StrategyName),

    (   call(ValidationBody),
        proves([] => [o(plus(A, B, CalculatedResult))])
    ->
        (   clause(run_learned_strategy(_, _, _, StrategyName, _), _)
        ->  format('  (Strategy ~w already known)~n', [StrategyName])
        ;   assertz((StrategyHead :- StrategyBody)),
            format('  -> New Strategy Asserted: ~w~n', [StrategyName])
        )
    ;   writeln('ERROR: Strategy validation failed. Not asserted.')
    ).

% ==================================================================
% Part 5: Normative Critique (Placeholder)
% ==================================================================

%!      critique_and_bootstrap(+Goal:term) is det.
%
%       Placeholder for a future capability where the system can analyze
%       a given normative rule (e.g., a subtraction problem that challenges
%       its current knowledge) and potentially learn from it.
%
%       @param Goal The goal representing the normative rule to critique.
critique_and_bootstrap(_) :- writeln('Normative Critique Placeholder.').
```

17 neuro/incompatibility_semantics.pl

```prolog
% Filename: incompatibility_semantics.pl (Neuro-Symbolic Integration)
:- module(incompatibility_semantics,
          [ proves/1, obj_coll/1, incoherent/1, set_domain/1, current_domain/1
          , product_of_list/2 % Exported for the learner module
          % Updated exports
          , s/1, o/1, n/1, comp_nec/1, exp_nec/1, exp_poss/1, comp_poss/1, neg/1
          , highlander/2, bounded_region/4, equality_iterator/3
          % Geometry
          , square/1, rectangle/1, rhombus/1, parallelogram/1, trapezoid/1, kite/1, quadrilateral/1
          , r1/1, r2/1, r3/1, r4/1, r5/1, r6/1
          % Number Theory (Euclid)
          , prime/1, composite/1, divides/2, is_complete/1
          % Fractions (Jason.pl)
          , rdiv/2, iterate/3, partition/3, normalize/2
          ]).
% Declare predicates that are defined across different sections.
:- discontiguous proves_impl/2.
:- discontiguous is_incoherent/1. % Non-recursive check

% ==================================================================
% Part 0: Setup and Configuration
% ==================================================================

% Define operators
:- op(500, fx, comp_nec).
```

```prolog
:- op(500, fx, exp_nec).
:- op(500, fx, exp_poss).
:- op(500, fx, comp_poss).
:- op(500, fx, neg).
:- op(1050, xfy, =>).
:- op(550, xfy, rdiv).

% ================================================================
% Part 1: Knowledge Domains
% ================================================================

% --- 1.1 Geometry ---
% (Geometry definitions remain the same as the original file)
incompatible_pair(square, r1). incompatible_pair(rectangle, r1). incompatible_pair(rhombus, r1). inc
incompatible_pair(square, r2). incompatible_pair(rhombus, r2). incompatible_pair(kite, r2).
incompatible_pair(square, r3). incompatible_pair(rectangle, r3). incompatible_pair(rhombus, r3). inc
incompatible_pair(square, r4). incompatible_pair(rhombus, r4). incompatible_pair(kite, r4).
incompatible_pair(square, r5). incompatible_pair(rectangle, r5). incompatible_pair(rhombus, r5). inc
incompatible_pair(square, r6). incompatible_pair(rectangle, r6).

is_shape(S) :- (incompatible_pair(S, _); S = quadrilateral), !.

entails_via_incompatibility(P, Q) :- P == Q, !.
entails_via_incompatibility(_, quadrilateral) :- !.
entails_via_incompatibility(P, Q) :- forall(incompatible_pair(Q, R), incompatible_pair(P, R)).

geometric_predicates([square, rectangle, rhombus, parallelogram, trapezoid, kite, quadrilateral, r1,

% --- 1.4 Fraction Domain ---
fraction_predicates([rdiv, iterate, partition]).

% --- 1.2 Arithmetic (O/N Domains) ---
% (Arithmetic definitions remain the same as the original file)

:- dynamic current_domain/1.
current_domain(n).

set_domain(D) :-
    ( member(D, [n, z, q]) -> retractall(current_domain(_)), assertz(current_domain(D)) ; true).

obj_coll(N) :- current_domain(n), !, integer(N), N >= 0.
obj_coll(N) :- current_domain(z), !, integer(N).
obj_coll(X) :- current_domain(q), !,
    ( integer(X)
    ; (X = N rdiv D, integer(N), integer(D), D > 0)
    ).

% --- Helpers for Rational Arithmetic ---
gcd(A, 0, A) :- A \= 0, !.
gcd(A, B, G) :- B \= 0, R is A mod B, gcd(B, R, G).

normalize(N, N) :- integer(N), !.
normalize(N rdiv D, R) :-
    (D =:= 1 -> R = N ;
        G is abs(gcd(N, D)),
        SN is N // G,
        SD is D // G,
        (SD =:= 1 -> R = SN ; R = SN rdiv SD)
    ), !.
```

```prolog
perform_arith(+, A, B, C) :- C is A + B.
perform_arith(-, A, B, C) :- C is A - B.

arith_op(A, B, Op, C) :-
    member(Op, [+, -]),
    normalize(A, NA), normalize(B, NB),
    (integer(NA), integer(NB) ->
        perform_arith(Op, NA, NB, C_raw)
    ;
        (integer(NA) -> N1=NA, D1=1 ; NA = N1 rdiv D1),
        (integer(NB) -> N2=NB, D2=1 ; NB = N2 rdiv D2),

        D_res is D1 * D2,
        N1_scaled is N1 * D2,
        N2_scaled is N2 * D1,

        perform_arith(Op, N1_scaled, N2_scaled, N_res),

        C_raw = N_res rdiv D_res
    ),
    normalize(C_raw, C).

% --- 1.3 Number Theory Domain (Euclid) ---

% Added 'euclid_number' concept, introduced by the neuro-symbolic bridge.
number_theory_predicates([prime, composite, divides, is_complete, member, euclid_number]).

excluded_predicates(AllPreds) :-
    geometric_predicates(G),
    number_theory_predicates(NT),
    fraction_predicates(F),
    append(G, NT, Temp),
    append(Temp, F, DomainPreds),
    append([neg, conj, nec, comp_nec, exp_nec, exp_poss, comp_poss, obj_coll], DomainPreds, AllPreds

% --- Helpers for Number Theory (Grounded) ---

product_of_list(L, P) :- (is_list(L) -> product_of_list_impl(L, P) ; fail).
product_of_list_impl([], 1).
product_of_list_impl([H|T], P) :- number(H), product_of_list_impl(T, P_tail), P is H * P_tail.

find_prime_factor(N, F) :- number(N), N > 1, find_factor_from(N, 2, F).
find_factor_from(N, D, D) :- N mod D =:= 0, !.
find_factor_from(N, D, F) :-
    D * D =< N,
    (D =:= 2 -> D_next is 3 ; D_next is D + 2),
    find_factor_from(N, D_next, F).
find_factor_from(N, _, N).

is_prime(N) :- number(N), N > 1, find_factor_from(N, 2, F), F =:= N.

% =================================================================
% Part 2: Core Logic Engine
% =================================================================

% Helper predicates
select(X, [X|T], T).
select(X, [H|T], [H|R]) :- select(X, T, R).
```

```prolog
match_antecedents([], _).
match_antecedents([A|As], Premises) :-
    member(A, Premises),
    match_antecedents(As, Premises).

% --- 2.1 Incoherence Definitions ---

incoherent(X) :- is_incoherent(X), !.
incoherent(X) :- proves(X => []).

% --- 1. Specific Material Optimizations ---

% Geometric Incompatibility
is_incoherent(X) :-
    member(n(ShapePred), X), ShapePred =.. [Shape, V],
    member(n(RestrictionPred), X), RestrictionPred =.. [Restriction, V],
    ground(Shape), ground(Restriction),
    incompatible_pair(Shape, Restriction), !.

% Arithmetic Incompatibility
is_incoherent(X) :-
    member(n(obj_coll(minus(A,B,_))), X),
    current_domain(n),
    normalize(A, NA), normalize(B, NB),
    NA < NB, !.

% M6-Case1: Euclid Case 1 Incoherence (Optimization)
is_incoherent(X) :-
    member(n(prime(EF)), X),
    member(n(is_complete(L)), X),
    % Check if the concept was introduced by the Muse, or calculate P+1 if needed.
    (member(n(euclid_number(EF, L)), X) ; (product_of_list(L, DE), EF is DE + 1)).


% --- 2. Base Incoherence (LNC) and Persistence ---
incoherent_base(X) :- member(P, X), member(neg(P), X).
incoherent_base(X) :- member(D_P, X), D_P =.. [D, P], member(D_NegP, X), D_NegP =.. [D, neg(P)], mem

is_incoherent(Y) :- incoherent_base(Y), !.


% --- 2.2 Sequent Calculus Prover (RESTRUCTURED) ---

proves(Sequent) :- proves_impl(Sequent, []).

% --- PRIORITY 1: Identity and Explosion ---
proves_impl((Premises => Conclusions), _) :-
    member(P, Premises), member(P, Conclusions), !.

proves_impl((Premises => _), _) :-
    is_incoherent(Premises), !.

% --- PRIORITY 2: Material Inferences and Grounding (Axioms) ---

% --- Arithmetic Grounding ---
proves_impl(_ => [o(eq(A,B))], _) :-
    obj_coll(A), obj_coll(B),
    normalize(A, NA), normalize(B, NB),
```

```
    NA == NB.

proves_impl(_ => [o(plus(A,B,C))], _) :-
    obj_coll(A), obj_coll(B),
    arith_op(A, B, +, C),
    obj_coll(C).

proves_impl(_ => [o(minus(A,B,C))], _) :-
    current_domain(D), obj_coll(A), obj_coll(B),
    arith_op(A, B, -, C),
    normalize(C, NC),
    ((D=n, NC >= 0) ; member(D, [z, q])),
    obj_coll(C).

% --- Arithmetic Material Inferences ---
proves_impl([n(plus(A,B,C))] => [n(plus(B,A,C))], _).

% --- EML Material Inferences (Axioms) ---
proves_impl([s(u)] => [s(comp_nec a)], _).
proves_impl([s(u_prime)] => [s(comp_nec a)], _).
proves_impl([s(a)] => [s(exp_poss lg)], _).
proves_impl([s(a)] => [s(comp_poss t)], _).
proves_impl([s(t)] => [s(comp_nec neg(u))], _).
proves_impl([s(lg)] => [s(exp_nec u_prime)], _).
proves_impl([s(t_b)] => [s(comp_nec t_n)], _).
proves_impl([s(t_n)] => [s(comp_nec t_b)], _).

% --- Fraction Grounding ---
proves_impl(([] => [o(iterate(U, M, R))]), _) :-
    obj_coll(U), integer(M), M >= 0,
    normalize(U, NU),
    (integer(NU) -> N1=NU, D1=1 ; NU = N1 rdiv D1),
    N_res is N1 * M,
    normalize(N_res rdiv D1, R).

proves_impl(([] => [o(partition(W, N, U))]), _) :-
    obj_coll(W), integer(N), N > 0,
    normalize(W, NW),
    (integer(NW) -> N1=NW, D1=1 ; NW = N1 rdiv D1),
    D_res is D1 * N,
    normalize(N1 rdiv D_res, U).

% --- Number Theory Material Inferences (Axioms/Definitions) ---

% M5 (Revised): If a prime G divides the Euclid number N derived from L, then G is not in L.
% This now relies on the concept introduced by the Muse.
proves_impl(( [n(prime(G)), n(divides(G, N)), n(euclid_number(N, L))] => [n(neg(member(G, L)))] ), _

% M4: If there is a prime G not in L, then L is not complete.
proves_impl(([n(prime(G)), n(neg(member(G, L)))] => [n(neg(is_complete(L)))]), _).

% Grounding Primality
proves_impl(([] => [n(prime(N))]), _) :- is_prime(N).
proves_impl(([] => [n(composite(N))]), _) :- number(N), N > 1, \+ is_prime(N).


% --- PRIORITY 3: Structural Rules (Domain Specific and General) ---

% Geometric Entailment
```

```
proves_impl((Premises => Conclusions), _) :-
    member(n(P_pred), Premises), P_pred =.. [P_shape, X], is_shape(P_shape),
    member(n(Q_pred), Conclusions), Q_pred =.. [Q_shape, X], is_shape(Q_shape),
    entails_via_incompatibility(P_shape, Q_shape), !.

% Structural Rule for EML Dynamics
proves_impl((Premises => Conclusions), History) :-
    select(s(P), Premises, RestPremises), \+ member(s(P), History),
    eml_axiom(s(P), s(M_Q)),
    ( (M_Q = comp_nec Q ; M_Q = exp_nec Q) -> proves_impl(([s(Q)|RestPremises] => Conclusions), [s(P
    ; ((M_Q = exp_poss _ ; M_Q = comp_poss _), (member(s(M_Q), Conclusions) ; member(M_Q, Conclusion
    ).

% Structural Rule: Prime Factorization (Existential Instantiation)
% This is a general principle of number theory, so we keep it in the core prover.
proves_impl((Premises => Conclusions), History) :-
    select(n(composite(N)), Premises, RestPremises),
    \+ member(factorization(N), History),
    find_prime_factor(N, G),
    NewPremises = [n(prime(G)), n(divides(G, N))|RestPremises],
    proves_impl((NewPremises => Conclusions), [factorization(N)|History]).

% --- General Structural Rule: Forward Chaining (Modus Ponens / MMP) ---
proves_impl((Premises => Conclusions), History) :-
    Module = incompatibility_semantics,
    clause(Module:proves_impl((A_clause => [C_clause]), _), B_clause),

    copy_term((A_clause, C_clause, B_clause), (Antecedents, Consequent, Body)),
    is_list(Antecedents),

    match_antecedents(Antecedents, Premises),
    call(Module:Body),
    \+ member(Consequent, Premises),
    proves_impl(([Consequent|Premises] => Conclusions), History).


% Arithmetic Evaluation
% (Arithmetic Evaluation remains the same as the original file)
proves_impl(([Premise|RestPremises] => Conclusions), History) :-
    (Premise =.. [Index, Expr], member(Index, [s, o, n]) ; (Index = none, Expr = Premise)),
    (compound(Expr) -> (
        functor(Expr, F, _),
        excluded_predicates(Excluded),
        \+ member(F, Excluded)
    ) ; true),
    \+ (compound(Expr), functor(Expr, rdiv, 2)),
    catch(Value is Expr, _, fail), !,
    (Index \= none -> NewPremise =.. [Index, Value] ; NewPremise = Value),
    proves_impl(([NewPremise|RestPremises] => Conclusions), History).


% --- PRIORITY 4: Reduction Schemata (Logical Connectives) ---
% (Logical connective rules remain the same as the original file)

% Left Negation (LN)
proves_impl((P => C), H) :- select(neg(X), P, P1), proves_impl((P1 => [X|C]), H).
proves_impl((P => C), H) :- select(D_NegX, P, P1), D_NegX=..[D, neg(X)], member(D,[s,o,n]), D_X=..[D

% Right Negation (RN)
```

```prolog
proves_impl((P => C), H) :- select(neg(X), C, C1), proves_impl(([X|P] => C1), H).
proves_impl((P => C), H) :- select(D_NegX, C, C1), D_NegX=..[D, neg(X)], member(D,[s,o,n]), D_X=..[D

% Conjunction (Generalized)
proves_impl((P => C), H) :- select(conj(X,Y), P, P1), proves_impl(([X,Y|P1] => C), H).
proves_impl((P => C), H) :- select(s(conj(X,Y)), P, P1), proves_impl(([s(X),s(Y)|P1] => C), H).

proves_impl((P => C), H) :- select(conj(X,Y), C, C1), proves_impl((P => [X|C1]), H), proves_impl((P
proves_impl((P => C), H) :- select(s(conj(X,Y)), C, C1), proves_impl((P => [s(X)|C1]), H), proves_im

% S5 Modal rules (Generalized)
proves_impl((P => C), H) :- select(nec(X), P, P1), !, ( proves_impl((P1 => C), H) ; \+ proves_impl((
proves_impl((P => C), H) :- select(nec(X), C, C1), !, ( proves_impl((P => C1), H) ; proves_impl(([]

% --- PRIORITY 5: Neuro-Symbolic Integration Point (The "Muse" Hook) ---
% If all standard logical reductions (Priority 1-4) fail, consult the learned strategies.

proves_impl((Premises => Conclusions), History) :-
    % Check if the bridge module is loaded and the predicate exists
    current_predicate(neuro_symbolic_bridge:suggest_strategy/3),
    % Call the bridge to suggest a strategy (The "neural" intuition)
    neuro_symbolic_bridge:suggest_strategy(Premises, Conclusions, Strategy),
    % Apply the suggested strategy (The "symbolic" execution)
    apply_strategy(Strategy, Premises, Conclusions, History).

% --- Strategy Application Helper ---

% Strategy: Introduce Lemma/Construction
apply_strategy(introduce(NewPremise), Premises, Conclusions, History) :-
    \+ member(NewPremise, Premises),
    proves_impl(([NewPremise|Premises] => Conclusions), History).

% Strategy: Case Split
apply_strategy(case_split(Case1, Case2), Premises, Conclusions, History) :-
    proves_impl(([Case1|Premises] => Conclusions), History),
    proves_impl(([Case2|Premises] => Conclusions), History).


% (Helpers for EML Dynamics)
eml_axiom(A, C) :-
    clause(incompatibility_semantics:proves_impl(([A] => [C]), _), true),
    is_eml_modality(C).

is_eml_modality(s(comp_nec _)).
is_eml_modality(s(exp_nec _)).
is_eml_modality(s(exp_poss _)).
is_eml_modality(s(comp_poss _)).

% =================================================================
% Part 4: Automata and Placeholders
% =================================================================
% (Placeholders remain the same as the original file)

highlander([Result], Result) :- !.
highlander([], _) :- !, fail.
highlander([_|Rest], Result) :- highlander(Rest, Result).

bounded_region(I, L, U, R) :- ( number(I), I >= L, I =< U -> R = in_bounds(I) ; R = out_of_bounds(I)
```

```
equality_iterator(T, T, T) :- !.
equality_iterator(C, T, R) :- C < T, C1 is C + 1, equality_iterator(C1, T, R).

% Placeholder definitions for exported functors
s(_). o(_). n(_). neg(_). comp_nec(_). exp_nec(_). exp_poss(_). comp_poss(_).
square(_). rectangle(_). rhombus(_). parallelogram(_). trapezoid(_). kite(_). quadrilateral(_).
r1(_). r2(_). r3(_). r4(_). r5(_). r6(_).
prime(_). composite(_). divides(_, _). is_complete(_).
rdiv(_, _). iterate(_, _, _). partition(_, _, _).
% Placeholder for the concept introduced by the bridge
euclid_number(_, _).
```

18 neuro/learned_knowledge_v2.pl

```
% Automatically generated knowledge base V2.
:- op(550, xfy, rdiv).
learned_proof_strategy(goal{context:[n(is_complete(A))], vars:[A, B]}, introduce(n(euclid_number(B,
    incompatibility_semantics:product_of_list(A, C),
    B is C+1,
    B>1.
learned_proof_strategy(goal{context:[n(euclid_number(A, B))], vars:[A, B]}, case_split(n(prime(A)),
```

19 neuro/neuro_symbolic_bridge.pl

```
% Filename: neuro_symbolic_bridge.pl (The Neuro-Symbolic Bridge V4)
:- module(neuro_symbolic_bridge,
          [ explore_calculation/1,
            solve/4,
            suggest_strategy/3, % Export for the prover hook
            learn_euclid_strategy/0 % Export for triggering simulated learning
          ]).

% Use the semantics engine
% Import product_of_list/2, needed for defining the Euclid construction strategy.
:- use_module(incompatibility_semantics, [proves/1, set_domain/1, current_domain/1, obj_coll/1, norm
:- use_module(library(random)).
:- use_module(library(lists)).

% Ensure operators are visible
:- op(1050, xfy, =>).
:- op(500, fx, neg).
:- op(550, xfy, rdiv).

% Dynamic predicates for learned strategies.
:- dynamic run_learned_strategy/5. % Calculation strategies
:- dynamic learned_proof_strategy/2. % Proof strategies (The "Intuition" Database)

% =============================================================
% Part 0: Initialization and Persistence
% =============================================================

knowledge_file('learned_knowledge_v2.pl').
%:- initialization(load_knowledge, now).

load_knowledge :-
    knowledge_file(File),
    (   exists_file(File)
    ->  consult(File),
```

```
        format('~N[Bridge Init] Loaded persistent knowledge.~n')
    ;   format('~N[Bridge Init] Knowledge file not found. Starting fresh.~n')
    ).

% Ensure initialization runs after the predicate is defined
:- initialization(load_knowledge, now).

save_knowledge :-
    knowledge_file(File),
    setup_call_cleanup(
        open(File, write, Stream),
        (
            writeln(Stream, '% Automatically generated knowledge base V2.'),
            writeln(Stream, ':- op(550, xfy, rdiv).'),
            % Save Calculation Strategies
            forall(clause(run_learned_strategy(A, B, R, S, T), Body),
                   portray_clause(Stream, (run_learned_strategy(A, B, R, S, T) :- Body))),
            % Save Proof Strategies
            forall(clause(learned_proof_strategy(GoalPattern, Strategy), Body),
                   portray_clause(Stream, (learned_proof_strategy(GoalPattern, Strategy) :- Body)))
        ),
        close(Stream)
    ).

% ===================================================================
% Part 1-4: Calculation Learning (Retained from more_machine_learner.pl)
% ===================================================================
% This section retains the functionality for optimizing arithmetic operations.

explore_calculation(addition) :-
    writeln('====================================================='),
    writeln('--- Autonomous Exploration Initiated: Addition ---'),
    current_domain(D),
    (member(D, [n, z, q]) -> explore_addition_loop(50) ; writeln('Requires domain (n, z, or q).')).

explore_addition_loop(0) :- save_knowledge, writeln('================================================
explore_addition_loop(I) :-
    generate_addition_problem(A, B),
    format('\n[Cycle ~w] Exploring Problem: ~w + ~w~n', [I, A, B]),
    (discover_strategy(A, B, _) ; true),
    NextI is I - 1,
    explore_addition_loop(NextI).

generate_addition_problem(A, B) :-
    random_between(3, 12, A),
    (   random(R), R < 0.3 -> B = A ; random_between(3, 15, B)).

% --- Solver Hierarchy ---
solve(A, B, Result, Trace) :-
    (   run_learned_strategy(A, B, Result, _StrategyName, Trace) -> true
    ;   solve_foundationally(A, B, Result, Trace)).

% --- Strategy Discovery (Calculation) ---
discover_strategy(A, B, StrategyName) :-
    solve(A, B, Result, Trace),
    count_trace_steps(Trace, TraceLength),
    format('  Solution found via [~w]: ~w. Steps: ~w~n', [Trace.strategy, Result, TraceLength]),
    (   detect_cob_pattern(Trace, _), StrategyName = cob, construct_and_validate_cob(A, B)
    ;   detect_rmb_pattern(Trace, RMB_Data), StrategyName = rmb, construct_and_validate_rmb(A, B, RM
```

```
    ;   detect_doubles_pattern(Trace, _), StrategyName = doubles, construct_and_validate_doubles(A,

% --- Foundational Ability: Counting ---
successor(X, Y) :- proves([] => [o(plus(X, 1, Y))]).

solve_foundationally(A, B, Result, Trace) :-
    obj_coll(A), obj_coll(B), integer(A), integer(B), B >= 0,
    count_loop(A, B, Result, Steps),
    Trace = trace{a_start:A, b_start:B, strategy:counting, steps:Steps}.

count_loop(CurrentA, 0, CurrentA, []) :- !.
count_loop(CurrentA, CurrentB, Result, [step(CurrentA, NextA)|Steps]) :-
    CurrentB > 0, NextB is CurrentB - 1, successor(CurrentA, NextA),
    count_loop(NextA, NextB, Result, Steps).

% (Trace Analysis Helpers)
count_trace_steps(Trace, Count) :-
    (   is_dict(Trace) ->
        (   member(Trace.strategy, [counting, doubles, rmb(_)]) -> length(Trace.steps, Count)
        ;   Trace.strategy = cob -> ( member(inner_trace(InnerTrace), Trace.steps) -> count_trace_st
        ;   Count = 1)
    ; Count = 0).

get_calculation_trace(T, T) :- is_dict(T), member(T.strategy, [counting, rmb(_), doubles]).
get_calculation_trace(T, CT) :- is_dict(T), T.strategy = cob, member(inner_trace(InnerT), T.steps),

% (Pattern Detection & Construction: COB, RMB, Doubles)
% PATTERN 1: Counting On Bigger (COB)
detect_cob_pattern(Trace, cob_data) :- is_dict(Trace), Trace.strategy = counting, A = Trace.a_start,

construct_and_validate_cob(A, B) :-
    StrategyName = cob,
    StrategyHead = run_learned_strategy(A_in, B_in, Result, StrategyName, Trace),
    StrategyBody = (
        integer(A_in), integer(B_in),
        (A_in >= B_in -> Start = A_in, Count = B_in, Swap = no_swap ; Start = B_in, Count = A_in, Sw
        (   Swap = swapped(_, _) -> (proves([n(plus(A_in, B_in, R_temp))] => [n(plus(B_in, A_in, R_t
        solve_foundationally(Start, Count, Result, InnerTrace),
        Trace = trace{a_start:A_in, b_start:B_in, strategy:StrategyName, steps:[Swap, inner_trace(In
    ),
    validate_and_assert(A, B, StrategyHead, StrategyBody).

% PATTERN 2: Rearranging to Make Bases (RMB)
detect_rmb_pattern(TraceWrapper, rmb_data{k:K, base:Base}) :-
    get_calculation_trace(TraceWrapper, Trace), Trace.strategy = counting, Base = 10,
    A = Trace.a_start, B = Trace.b_start, integer(A), integer(B),
    A > 0, A < Base, K is Base - A, B >= K, nth1(K, Trace.steps, Step), Step = step(_, Base).

construct_and_validate_rmb(A, B, RMB_Data) :-
    Base = RMB_Data.base, StrategyName = rmb(Base),
    StrategyHead = run_learned_strategy(A_in, B_in, Result, StrategyName, Trace),
    StrategyBody = (
        integer(A_in), integer(B_in), A_in > 0, A_in < Base, K_runtime is Base - A_in, B_in >= K_run
        B_new_runtime is B_in - K_runtime, Result is Base + B_new_runtime,
        Trace = trace{a_start:A_in, b_start:B_in, strategy:StrategyName, steps:[step(A_in, Base), st
    ),
    validate_and_assert(A, B, StrategyHead, StrategyBody).

% PATTERN 3: Doubles
```

```
detect_doubles_pattern(TraceWrapper, doubles_data) :-
    get_calculation_trace(TraceWrapper, Trace), member(Trace.strategy, [counting, rmb(_)]),
    A = Trace.a_start, B = Trace.b_start, A == B, integer(A).

construct_and_validate_doubles(A, B) :-
    StrategyName = doubles,
    StrategyHead = run_learned_strategy(A_in, B_in, Result, StrategyName, Trace),
    StrategyBody = (
        integer(A_in), A_in == B_in, Result is A_in * 2,
        Trace = trace{a_start:A_in, b_start:B_in, strategy:StrategyName, steps:[rote(Result)]}
    ),
    validate_and_assert(A, B, StrategyHead, StrategyBody).

% Validation Helper
validate_and_assert(A, B, StrategyHead, StrategyBody) :-
    copy_term((StrategyHead, StrategyBody), (ValidationHead, ValidationBody)),
    arg(1, ValidationHead, A), arg(2, ValidationHead, B), arg(3, ValidationHead, CalculatedResult),
    (   call(ValidationBody), proves([] => [o(plus(A, B, CalculatedResult))])
    ->  (   clause(run_learned_strategy(_, _, _, StrategyName, _), _) -> format('  (Strategy ~w alre
        ;   assertz((StrategyHead :- StrategyBody)), format('  -> New Strategy Asserted: ~w~n', [Str
    ;   writeln('ERROR: Strategy validation failed. Not asserted.')).


% =================================================================
% Part 5: Neuro-Symbolic Proof Strategy Integration (The "Muse")
% =================================================================

% suggest_strategy(+Premises, +Conclusions, -Strategy)
% This is the hook called by the prover when it is stuck (PRIORITY 5).
suggest_strategy(Premises, Conclusions, Strategy) :-
    % 1. Identify the Goal Pattern (Optional, useful for goal-directed strategies)
    (   Conclusions = [] -> Goal = incoherent(Premises)
    ;   member(C, Conclusions), Goal = proves(Premises => [C])
    ),

    % 2. Consult Learned Strategies (The "Intuition Database")
    % Use findall and then select to allow backtracking through different suggestions if the first f
    findall(S, consult_learned_proof_strategies(Premises, Goal, S), Strategies),
    member(Strategy, Strategies).

% consult_learned_proof_strategies(+Premises, +Goal, -Strategy)
consult_learned_proof_strategies(Premises, _Goal, Strategy) :-
    % Iterate through learned strategies. The associated Body is executed here by clause/2 and call/
    clause(learned_proof_strategy(GoalPattern, StrategyTemplate), Body),

    % Check if the current premises match the required context for the strategy.
    % This binds variables in GoalPattern (like L) to the actual values in the proof state.
    match_context(GoalPattern.context, Premises),

    % Execute the body (e.g., to calculate constructions like N=P+1).
    % This binds variables used in the calculation (like N).
    call(Body),

    % Instantiate the strategy template with the bound variables.
    instantiate_strategy(StrategyTemplate, GoalPattern.vars, Strategy).

% Helper to check context and bind variables
match_context([], _).
match_context([P|Ps], Premises) :-
```

```prolog
    % Use member/2 for unification, binding variables in P (like L in n(is_complete(L)))
    member(P, Premises),
    match_context(Ps, Premises).

% Helper to instantiate the strategy
instantiate_strategy(Template, Vars, Strategy) :-
    % Ensures variables bound during match_context and the body execution are propagated.
    copy_term((Template, Vars), (Strategy, _)).

% =================================================================
% Part 6: The Learning/Reflection Process (The "Critique")
% =================================================================

% This section simulates the "neural" process of analyzing a domain and discovering a strategy.

learn_euclid_strategy :-
    writeln('\n--- Neuro-Symbolic Reflection Initiated: Euclid Domain (The "Muse") ---'),
    % 1. Analyze the Domain (Simulated Intuition)
    % The "Muse" recognizes that to disprove completeness, one needs a construction and subsequent a

    % 2. Formulate the Strategy

    % Strategy 1: Euclid Construction
    % "When assuming is_complete(L), construct the Euclid number N."
    Pattern1 = goal{
        context: [n(is_complete(L))],
        vars: [L, N] % Variables involved (L and N are unbound here)
    },
    % Action: Introduce the constructed number concept
    StrategyTemplate1 = introduce(n(euclid_number(N, L))),
    % Preconditions/Calculations: How to instantiate N based on L.
    Body1 = (
        % We must qualify the call as product_of_list resides in the other module.
        incompatibility_semantics:product_of_list(L, P),
        N is P + 1,
        N > 1 % Prerequisite for prime analysis
    ),
    assert_proof_strategy(Pattern1, StrategyTemplate1, Body1, 'euclid_construction'),

    % Strategy 2: Case Analysis
    % "When analyzing a constructed Euclid number N, consider if it is prime or composite."
    Pattern2 = goal{
        context: [n(euclid_number(N, L))],
        vars: [N, L]
    },
    StrategyTemplate2 = case_split(n(prime(N)), n(composite(N))),
    Body2 = true, % Conditions (N>1) are checked in the construction phase

    assert_proof_strategy(Pattern2, StrategyTemplate2, Body2, 'euclid_case_analysis'),

    save_knowledge,
    writeln('--- Reflection Complete. Knowledge base updated. ---').

% Helper to assert a new proof strategy if not already known
assert_proof_strategy(GoalPattern, StrategyTemplate, Body, Name) :-
    % We assert the strategy with its body, so the body is executed when the strategy is consulted.
    (   clause(learned_proof_strategy(GP, ST), B),
        % Check if a strategy with the same structure already exists (variant check)
        variant((GP, ST, B), (GoalPattern, StrategyTemplate, Body))
```

```
    -> format('  (Proof strategy ~w already known)~n', [Name])
    ;   % Assert the clause: (learned_proof_strategy(GoalPattern, StrategyTemplate) :- Body).
        assertz((learned_proof_strategy(GoalPattern, StrategyTemplate) :- Body)),
        format('  -> New Proof Strategy Asserted: ~w~n', [Name])
    ).
```

20 neuro/test_synthesis.pl

```
% Filename: test_synthesis.pl (Updated for Neuro-Symbolic Testing)
% Load the core module
:- use_module(incompatibility_semantics, [
    proves/1, incoherent/1, set_domain/1, obj_coll/1, normalize/2
]).
% Load the bridge module to access the learning triggers.
% We must ensure the bridge is loaded so the Priority 5 hook in the prover can find it.
:- use_module(neuro_symbolic_bridge, [learn_euclid_strategy/0]).

:- use_module(library(plunit)).

% Ensure operators are visible
:- op(500, fx, neg).
:- op(500, fx, comp_nec).
:- op(500, fx, exp_nec).
:- op(500, fx, exp_poss).
:- op(500, fx, comp_poss).
:- op(1050, xfy, =>).
:- op(550, xfy, rdiv).

% Helper to clear knowledge for isolated tests
clear_knowledge :-
    retractall(neuro_symbolic_bridge:learned_proof_strategy(_, _)),
    retractall(neuro_symbolic_bridge:run_learned_strategy(_, _, _, _, _)).

:- begin_tests(unified_synthesis).

% --- Tests for Part 1: Core Logic and Domains ---
test(identity_subjective) :- assertion(proves([s(p)] => [s(p)])).
test(incoherence_subjective) :- assertion(incoherent([s(p), s(neg(p))])).

test(negation_handling_subjective_lem) :-
    assertion(proves([] => [s(p), s(neg(p))])).

% --- Tests for Part 2: Arithmetic Coexistence and Fixes ---

test(arithmetic_commutativity_normative) :-
    assertion(proves([n(plus(2,3,5))] => [n(plus(3,2,5))])).

test(arithmetic_subtraction_limit_n, [setup(set_domain(n))]) :-
    assertion(incoherent([n(obj_coll(minus(3,5,_)))])).

test(arithmetic_subtraction_limit_z, [setup(set_domain(z))]) :-
    assertion(\+(incoherent([n(obj_coll(minus(3,5,_)))]))).

% --- Tests for Part 3: Embodied Modal Logic (EML) ---
test(eml_dynamic_u_to_a) :- assertion(proves([s(u)] => [s(a)])).
test(eml_dynamic_full_cycle) :- assertion(proves([s(lg)] => [s(a)])).
test(eml_tension_conjunction) :-
    assertion(proves([s(a)] => [s(conj(exp_poss lg, comp_poss t))])).
```

```prolog
% --- Tests for Quadrilateral Hierarchy ---

test(quad_incompatibility_square_r1) :-
    assertion(incoherent([n(square(x)), n(r1(x))])).

test(quad_entailment_square_rectangle) :-
    assertion(proves([n(square(x))] => [n(rectangle(x))])).


% --- Tests for Number Theory (Euclid's Proof) ---

% Test Grounding Helpers and Material Inferences (These rely only on Axioms, not Strategies)
test(euclid_grounding_prime) :-
    assertion(proves([] => [n(prime(7))])).

% Note: M5 definition now uses the 'euclid_number' concept.
test(euclid_material_inference_m5) :-
    % L=[2,3], N=7.
    assertion(proves([n(prime(7)), n(divides(7, 7)), n(euclid_number(7, [2,3]))] => [n(neg(member(7,

test(euclid_material_inference_m4) :-
    assertion(proves([n(prime(5)), n(neg(member(5, [2, 3])))] => [n(neg(is_complete([2, 3])))] )).

% Test Forward Chaining (Using the prover's built-in forward chaining - Priority 3)
test(euclid_forward_chaining) :-
    % L=[2,3], N=7.
    Premises = [n(prime(7)), n(divides(7, 7)), n(euclid_number(7, [2,3])), n(is_complete([2, 3]))],
    Conclusion = [n(neg(is_complete([2, 3])))],
    assertion(proves(Premises => Conclusion)).

% Test The Final Theorem (Euclid's Theorem)
% !!! NEURO-SYMBOLIC TEST !!!
% These tests rely on the strategies learned via the Neuro-Symbolic Bridge (Priority 5).

test(euclid_theorem_infinitude_of_primes, [
    % The setup simulates the "neural" reflection phase.
    % We clear knowledge first to ensure learning happens fresh for the test.
    setup((clear_knowledge, learn_euclid_strategy))
]) :-
    L = [2, 5, 11],
    % The prover is stuck (Priority 1-4 fail).
    % It calls the Muse (Priority 5).
    % The Muse suggests 'euclid_construction' -> introduces n(euclid_number(111, L)).
    % The Muse suggests 'euclid_case_analysis' -> splits into Prime(111) or Composite(111).
    % Both cases lead to incoherence.
    assertion(incoherent([n(is_complete(L))])).

test(euclid_theorem_empty_list, [
     setup((clear_knowledge, learn_euclid_strategy))
]) :-
    % Construction: N = Product([]) + 1 = 1 + 1 = 2.
    % Case Split: Prime(2) or Composite(2).
    % Case 1: Prime(2). Leads to incoherence.
    assertion(incoherent([n(is_complete([]))])).

% --- Tests for Fractions (Jason.pl integration) ---

test(fraction_normalization) :-
    assertion(normalize(4 rdiv 8, 1 rdiv 2)).
```

```
test(fraction_addition_grounding, [setup(set_domain(q))]) :-
    % 1/2 + 1/3 = 5/6
    assertion(proves([] => [o(plus(1 rdiv 2, 1 rdiv 3, 5 rdiv 6))])).

test(fraction_subtraction_limit_n, [setup(set_domain(n))]) :-
    % 1/3 - 1/2 = -1/6. Incoherent in N.
    assertion(incoherent([n(obj_coll(minus(1 rdiv 3, 1 rdiv 2, _)))])).

:- end_tests(unified_synthesis).
```

21 object_level.pl

```
/** <module> Object-Level Knowledge Base
 *
 * This module represents the "object level" of the cognitive architecture.
 * It contains the initial, and potentially flawed, knowledge base that the
 * system reasons with. The predicates defined in this module are the ones
 * that are observed by the meta-interpreter and modified by the
 * reorganization engine.
 *
 * The key predicate `add/3` is declared as `dynamic` because it is the
 * target of learning and reorganization. Its initial implementation is
 * deliberately inefficient to create opportunities for the system to detect
 * disequilibrium and self-improve.
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(object_level, [add/3, multiply/3]).

:- dynamic add/3.
:- dynamic multiply/3.

% enumerate/1
% Helper to force enumeration of a Peano number. Its primary purpose
% in this context is to consume inference steps in the meta-interpreter,
% making the initial `add/3` implementation inefficient and prone to
% resource exhaustion, which acts as a trigger for reorganization.
enumerate(0).
enumerate(s(N)) :- enumerate(N).

% recursive_add/3
% This is the standard, efficient, recursive definition of addition for
% Peano numbers. It serves as the "correct" implementation that the
% reorganization engine will synthesize and assert when the initial,
% inefficient `add/3` rule is retracted.
recursive_add(0, B, B).
recursive_add(s(A), B, s(Sum)) :-
    recursive_add(A, B, Sum).

%!      add(?A, ?B, ?Sum) is nondet.
%
%       The initial, inefficient definition of addition.
%       This predicate is designed to simulate a "counting-all" strategy. It
%       works by first completely grounding the two inputs `A` and `B` by
%       recursively calling `enumerate/1`. This process is computationally
%       expensive and is intended to fail (by resource exhaustion) for larger
%       numbers, thus triggering the ORR learning cycle.
```

```
%
%       This predicate is declared `dynamic` and will be replaced by a more
%       efficient version by the `reorganization_engine`.
%
%       @param A A Peano number representing the first addend.
%       @param B A Peano number representing the second addend.
%       @param Sum The Peano number representing the sum of A and B.
add(A, B, Sum) :-
    enumerate(A),
    enumerate(B),
    recursive_add(A, B, Sum).

%!      multiply(?A, ?B, ?Product) is nondet.
%
%       The initial, inefficient definition of multiplication.
%       This predicate is designed to simulate multiplication via repeated
%       addition. It is computationally expensive and intended to trigger
%       reorganization for larger numbers.
%
%       This predicate is declared `dynamic` and will be replaced by a more
%       efficient version by the `reorganization_engine`.
multiply(A, B, Product) :-
    enumerate(A),
    enumerate(B),
    recursive_multiply(A, B, Product).

% recursive_multiply/3
% This is the standard, efficient, recursive definition of multiplication.
recursive_multiply(0, _, 0).
recursive_multiply(s(A), B, Product) :-
    recursive_multiply(A, B, PartialProduct),
    add(PartialProduct, B, Product).
```

22 reflective_monitor.pl

```
/** <module> Reflective Monitor for Disequilibrium Detection
 *
 * This module implements the "Reflect" stage of the ORR cycle. Its primary
 * role is to analyze the execution trace produced by the meta-interpreter
 * (`meta_interpreter.pl`) and detect signs of "disequilibrium."
 *
 * Disequilibrium can manifest in two main ways:
 * 1.   **Goal Failure**: The system was unable to find a proof for the goal.
 * 2.   **Logical Incoherence**: The proof that was found relies on a set of
 *      commitments (clauses) that are logically inconsistent with each other,
 *      as determined by `incompatibility_semantics.pl`.
 *
 * This module also maintains a "conceptual stress map," which tracks how
 * often certain predicates are involved in failures. This map can be used by
 * the reorganization engine to guide its search for a solution.
 *
 * The stress map is stored as dynamic facts of the form:
 * `stress(PredicateSignature, Count)`.
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(reflective_monitor, [
    reflect/2,
```

```
    get_stress_map/1,
    reset_stress_map/0
]).

:- use_module(incompatibility_semantics).

:- dynamic stress/2.

%!      reflect(+Trace:list, -DisequilibriumTrigger:term) is semidet.
%
%       Analyzes an execution trace from the meta-interpreter to detect
%       disequilibrium. It succeeds if a trigger for disequilibrium is found,
%       binding `DisequilibriumTrigger` to a term describing the issue. It
%       fails if the trace represents a state of equilibrium (i.e., the goal
%       succeeded and its premises are coherent).
%
%       The process involves:
%       1. Parsing the trace to separate successful commitments from failures.
%       2. Updating a conceptual stress map based on any failures.
%       3. Checking for disequilibrium triggers, prioritizing goal failure over
%          incoherence.
%
%       @param Trace The execution trace generated by `solve/4`.
%       @param DisequilibriumTrigger A term describing the reason for
%       disequilibrium, e.g., `goal_failure([...])` or `incoherence([...])`.
reflect(Trace, Trigger) :-
    % 1. Parse the trace to extract commitments and failures.
    parse_trace(Trace, Commitments, Failures),

    % 2. Update the conceptual stress map based on failures.
    update_stress_map(Failures),

    % 3. Check for disequilibrium triggers.
    (
        % Trigger 1: Goal Failure
        Failures \= [],
        Trigger = goal_failure(Failures), !
    ;
        % Trigger 2: Logical Incoherence
        incoherent(Commitments),
        Trigger = incoherence(Commitments), !
    ).


% parse_trace(+Trace, -Commitments, -Failures)
%
% Recursively walks the trace structure generated by the meta-interpreter
% and extracts the list of commitments (clauses used) and failures.
parse_trace(Trace, Commitments, Failures) :-
    parse_trace_recursive(Trace, Commitments_Nested, Failures_Nested),
    flatten(Commitments_Nested, Commitments),
    flatten(Failures_Nested, Failures).

parse_trace_recursive([], [], []).
parse_trace_recursive([Event|Events], [Commitments|Other_Cs], [Failures|Other_Fs]) :-
    parse_event(Event, Commitments, Failures),
    parse_trace_recursive(Events, Other_Cs, Other_Fs).

% How to handle each type of trace event.
```

```
parse_event(trace(_, SubTrace), C, F) :- parse_trace_recursive(SubTrace, C, F).
parse_event(clause(Clause), [Clause], []).
parse_event(fail(Goal), [], [fail(Goal)]).
parse_event(call(_), [], []). % Built-in calls are not commitments in this context.


% update_stress_map(+Failures)
%
% For each failed goal, identify the clause signature and increment its stress level.
update_stress_map([]).
update_stress_map([fail(Goal)|Failures]) :-
    functor(Goal, Name, Arity),
    increment_stress(Name/Arity),
    update_stress_map(Failures).

increment_stress(Signature) :-
    (   retract(stress(Signature, Count))
    ->  NewCount is Count + 1
    ;   NewCount = 1
    ),
    assertz(stress(Signature, NewCount)).

% --- Public helpers for managing the stress map ---

%!      get_stress_map(-Map:list) is det.
%
%       Returns the current conceptual stress map as a list of
%       `stress(Signature, Count)` terms.
%
%       @param Map A list containing all current stress facts.
get_stress_map(Map) :-
    findall(stress(Signature, Count), stress(Signature, Count), Map).

%!      reset_stress_map is det.
%
%       Clears the entire conceptual stress map by retracting all `stress/2` facts.
reset_stress_map :-
    retractall(stress(_, _)).
```

23 reorganization_engine.pl

```
/** <module> Reorganization Engine for Cognitive Accommodation
 *
 * This module implements the "Reorganize" stage of the ORR cycle. It is
 * responsible for `accommodate/1`, the process of modifying the system's
 * own knowledge base (`object_level.pl`) in response to a state of
 * disequilibrium detected by the `reflective_monitor.pl`.
 *
 * The engine currently handles failures by:
 * 1.  Identifying the predicate causing the most "conceptual stress" (i.e.,
 *     the one involved in the most failures).
 * 2.  Applying a predefined transformation strategy to that predicate.
 *
 * The only transformation implemented is `specialize_add_rule`, which
 * replaces a failing `add/3` implementation with a more robust, recursive
 * one based on the Peano axioms.
 *
 * @author Tilo Wiedera
 * @license MIT
```

```prolog
 */
:- module(reorganization_engine, [accommodate/1]).

:- use_module(object_level).
:- use_module(reflective_monitor).
:- use_module(reorganization_log).
:- use_module(more_machine_learner).
:- use_module(incompatibility_semantics).
:- use_module(strategies). % Load all defined strategies

% 'learned_knowledge.pl' is consulted into the learner's module at runtime
% (see more_machine_learner:load_knowledge/0). It is not a separate module, so
% attempting to reexport from it causes a domain error. Remove the faulty
% reexport directive.
% :- reexport(learned_knowledge, [learned_rule/1]).

%!      reorganize_system(+Goal:term, +Trace:list) is semidet.
%
%       The main entry point for the reorganization process, triggered when
%       a perturbation (e.g., resource exhaustion) occurs. This predicate
%       orchestrates the analysis, synthesis, validation, and integration of
%       a new, more efficient strategy.
%
%       @param Goal The goal that failed.
%       @param Trace The execution trace leading to the failure.
reorganize_system(Goal, _Trace) :-
    % Deconstruct the goal to get the arguments
    Goal =.. [Pred, A, B, _Result],
    ( (Pred = add ; Pred = multiply) ->
        % Convert Peano numbers to integers for the learner
        peano_to_int(A, IntA),
        peano_to_int(B, IntB),

        writeln('Invoking machine learner to discover new strategies...'),
        % The learner will analyze, validate, and assert the new rule internally
        (   more_machine_learner:discover_strategy(IntA, IntB, StrategyName) ->
            format('Learner discovered and asserted strategy: ~w~n', [StrategyName]),
            more_machine_learner:save_knowledge,
            writeln('New knowledge has been persisted.')
        ;   writeln('Learner did not find a new strategy for this case.'),
            fail
        )
    ;
        format('Reorganization for predicate ~w is not supported.~n', [Pred]),
        fail
    ).

%!      peano_to_int(+Peano, -Int) is det.
%
%       Converts a Peano number (e.g., `s(s(0))`) to an integer.
peano_to_int(0, 0).
peano_to_int(s(N), Int) :-
    peano_to_int(N, SubInt),
    Int is SubInt + 1.

%!      integrate_new_rule(+Rule:term) is det.
%
%       Integrates a validated new rule into the system's knowledge base.
%       It retracts the old, inefficient rule and asserts the new one in
```

```prolog
%       the `object_level` module.
integrate_new_rule((Head :- Body)) :-
    functor(Head, Name, Arity),
    retractall(object_level:Name/Arity),
    assertz(object_level:(Head :- Body)),
    log_event(reorganized(from(Name/Arity), to(Head :- Body))).

%!      save_learned_rule(+Rule:term) is det.
%
%       Persists a newly learned rule to the `learned_knowledge.pl` file
%       so that it can be reused across sessions.
save_learned_rule(Rule) :-
    open('learned_knowledge.pl', append, Stream),
    format(Stream, 'learned_rule(~q).~n', [Rule]),
    close(Stream).

%!      accommodate(+Trigger:term) is semidet.
%
%       Attempts to accommodate a state of disequilibrium by modifying the
%       knowledge base. This is the main entry point for the reorganization engine.
%
%       It dispatches to different handlers based on the type of `Trigger`:
%       - `goal_failure` or `perturbation`: Calls `handle_failure/1` to attempt
%         a knowledge repair based on conceptual stress.
%       - `incoherence`: Currently a placeholder; fails as this type of
%         reorganization is not yet implemented.
%
%       Succeeds if a transformation is successfully applied. Fails otherwise.
%
%       @param Trigger The term describing the disequilibrium, provided by the
%       reflective monitor.
accommodate(Trigger) :-
    (   (Trigger = goal_failure(_); Trigger = perturbation(_)) ->
        handle_failure(Trigger)
    ;   Trigger = incoherence(Commitments) ->
        handle_incoherence(Commitments)
    ;   format('Unknown trigger type: ~w. Cannot accommodate.~n', [Trigger]),
        fail
    ).

% handle_failure(+Trigger)
%
% Handles disequilibrium caused by goal failure. It identifies the most
% stressed predicate from the conceptual stress map and attempts to apply a
% transformation to repair it.
handle_failure(_Trigger) :-
    get_most_stressed_predicate(Signature),
    format('Highest conceptual stress found for predicate: ~w~n', [Signature]),
    log_event(reorganization_start(Signature)),
    apply_transformation(Signature).

% handle_incoherence(+Commitments)
%
% Placeholder for handling disequilibrium caused by logical contradictions.
% This is a future work area and currently always fails.
handle_incoherence(Commitments) :-
    format('Handling incoherence for commitments: ~w~n', [Commitments]),
    format('Incoherence-driven reorganization is not yet implemented.~n'),
    fail.
```

```prolog
% get_most_stressed_predicate(-Signature)
%
% Finds the predicate with the highest stress count in the stress map
% maintained by the reflective monitor.
get_most_stressed_predicate(Signature) :-
    get_stress_map(StressMap),
    StressMap \= [],
    find_max_stress(StressMap, stress(_, 0), stress(Signature, _)), !.
get_most_stressed_predicate(_) :-
    format('Could not identify a stressed predicate. Reorganization failed.~n'),
    fail.

% find_max_stress(+StressMap, +CurrentMax, -Max)
%
% Helper predicate to find the maximum entry in the stress map list.
find_max_stress([], Max, Max).
find_max_stress([stress(S, C)|Rest], stress(_, MaxC), Max) :-
    C > MaxC, !, find_max_stress(Rest, stress(S, C), Max).
find_max_stress([_|Rest], Max, Result) :- find_max_stress(Rest, Max, Result).

% apply_transformation(+Signature)
%
% Dispatches to a specific transformation strategy based on the predicate
% signature. Currently, only a transformation for `add/3` exists.
apply_transformation(add/3) :-
    !, specialize_add_rule.
apply_transformation(Signature) :-
    format('No specific reorganization strategy available for ~w.~n', [Signature]),
    fail.

% --- Transformation Strategies ---

% specialize_add_rule/0
%
% A specific transformation strategy that replaces the existing `add/3` rules
% with a correct, recursive implementation based on Peano arithmetic. This
% represents a form of learning or knowledge repair.
specialize_add_rule :-
    format('Applying "Specialization" strategy to add/3.~n'),
    % Retract all existing rules for add/3 and log each one.
    forall(
        clause(object_level:add(A, B, C), Body),
        (   retract(object_level:add(A, B, C) :- Body),
            log_event(retracted((add(A, B, C) :- Body)))
        )
    ),
    % Synthesize and assert the new, correct rule and log it.
    NewHead = add(A, B, Sum),
    NewBody = recursive_add(A, B, Sum),
    assertz(object_level:(NewHead :- NewBody)),
    log_event(asserted((NewHead :- NewBody))),
    format('Asserted new specialized add/3 clause.~n'),
    % Synthesize and assert helper predicates if they don't exist.
    (   \+ predicate_property(object_level:recursive_add(_,_,_), defined) ->
        assert_and_log((object_level:recursive_add(0, X, X))),
        assert_and_log((object_level:recursive_add(s(N), Y, s(Z)) :- object_level:recursive_add(N, Y
        format('Asserted helper predicate recursive_add/3.~n')
    ;   true
```

```
    ),
    log_event(reorganization_success).

% assert_and_log(+Clause)
%
% Helper to assert a clause and log the assertion event.
assert_and_log(Clause) :-
    assertz(Clause),
    log_event(asserted(Clause)).
```

24 reorganization_log.pl

```
/** <module> Reorganization and Cognitive Process Logger
 *
 * This module provides a logging facility for the ORR (Observe, Reorganize,
 * Reflect) cycle. It captures key events during the cognitive process,
 * such as the start of a cycle, detection of disequilibrium, and the
 * success or failure of reorganization attempts.
 *
 * The log can be retrieved as a raw list of events or generated as a
 * human-readable narrative report using a Definite Clause Grammar (DCG).
 *
 * Log entries are stored as dynamic facts of the form:
 * `log_entry(Timestamp, Event)`.
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(reorganization_log, [
    log_event/1,
    get_log/1,
    clear_log/0,
    generate_report/1
]).

:- dynamic log_entry/2.

%!      log_event(+Event:term) is det.
%
%       Records a structured event in the log with a current timestamp.
%
%       @param Event The structured term representing the event to be logged
%       (e.g., `disequilibrium(trigger_term)`).
log_event(Event) :-
    get_time(Timestamp),
    assertz(log_entry(Timestamp, Event)).

%!      get_log(-Log:list) is det.
%
%       Retrieves the entire log as a list of `log_entry/2` facts.
%
%       @param Log A list of all `log_entry(Timestamp, Event)` terms currently
%       in the database.
get_log(Log) :-
    findall(log_entry(T, E), log_entry(T, E), Log).

%!      clear_log is det.
%
%       Clears all entries from the reorganization log by retracting all
```

```
%       `log_entry/2` facts. This is typically done before starting a new
%       `run_query/1`.
clear_log :-
    retractall(log_entry(_, _)).

%!      generate_report(-Report:string) is det.
%
%       Translates the current log into a single, human-readable narrative string.
%       It uses a DCG to convert the structured log events into descriptive sentences.
%
%       @param Report The generated narrative report as a string.
generate_report(Report) :-
    get_log(Log),
    phrase(narrative(Log), Tokens),
    atomics_to_string(Tokens, Report).

% --- DCG for Narrative Generation ---

% narrative//1 processes the list of log entries.
narrative([]) --> [].
narrative([log_entry(_, Event)|Rest]) -->
    event_narrative(Event),
    narrative(Rest).

% event_narrative//1 translates a single event term into a string component.
event_narrative(orr_cycle_start(Goal)) -->
    ["- System started observing goal: ", Goal, ".\n"].

event_narrative(disequilibrium(Trigger)) -->
    ["- Reflection detected disequilibrium. Trigger: ", Trigger, ".\n"].

event_narrative(reorganization_start(Signature)) -->
    ["- Reorganization started, targeting predicate: ", Signature, ".\n"].

event_narrative(retracted(Clause)) -->
    ["  - The old clause was retracted: ", Clause, ".\n"].

event_narrative(asserted(Clause)) -->
    ["  - A new clause was asserted: ", Clause, ".\n"].

event_narrative(reorganization_success) -->
    ["- Reorganization was successful. System is retrying the goal to seek a new equilibrium.\n"].

event_narrative(reorganization_failure) -->
    ["- Reorganization failed. The system could not find a way to accommodate the issue.\n"].

event_narrative(equilibrium) -->
    ["- Equilibrium reached. The goal succeeded and was found to be coherent.\n"].

event_narrative(Unknown) -->
    ["- An unknown event was logged: ", Unknown, ".\n"].
```

25 RMB.pl

```
/** <module> Reflective Pushdown Automaton for RMB Strategy
 *
 * This module provides a detailed, low-level simulation of the 'Rearranging
 * to Make Bases' (RMB) addition strategy, implemented as a Pushdown
 * Automaton (PDA).
```

```
 *
 * **Note:** This appears to be an older or more experimental implementation
 * compared to `sar_add_rmb.pl`. It includes a unique "reflective" state
 * (`q6`) that demonstrates emergent behavior under specific conditions, which
 * is not present in the simplified `sar` models.
 *
 * The automaton processes an input string like `[4, '+', 8]` and uses a
 * stack to manipulate the numbers. The core logic involves deciding whether
 * a standard RMB rearrangement is possible or if a special reflective loop
 * should be entered.
 *
 * The main entry point is `run/4`.
 *
 * @author Theodore M. Savich (Concept), Revised Implementation (AI Assist)
 * @license Unknown
 */
:- module(refrmb_corrected, [run/4]).
:- use_module(library(lists)).

% --- Dynamic Predicates for State ---
:- dynamic stored_A/1.
:- dynamic stored_B/1.
:- dynamic transition/5.
:- dynamic stack_item/1.
:- dynamic reflection_enabled/1.
:- dynamic decision_made/1.

% --- Configuration ---
base(10).

% --- Define valid digits ---
digit(D) :- member(D, [0,1,2,3,4,5,6,7,8,9]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%           Main Entry Point            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%!      run(+Start:integer, +Input:list, -Result:atom, +ReflectFlag:atom) is det.
%
%       Runs the Reflective Pushdown Automaton simulation.
%
%       This is the main entry point for the module. It initializes the
%       automaton's state by clearing any dynamic facts from previous runs,
%       setting up the initial stack, and defining the static transitions.
%       It then starts the simulation process by calling `step/4`.
%
%       @param Start The initial state of the automaton (e.g., `1`).
%       @param Input The input string to be processed, as a list of atoms
%       and numbers (e.g., `[4, '+', 8]`).
%       @param Result The final result of the run, either `accept` or `error`.
%       @param ReflectFlag A flag (`y` or `n`) to enable or disable the
%       special reflective behavior of the automaton.
run(Start, Input, Result, ReflectFlag) :-
    % --- Cleanup from any previous run ---
    retractall(stored_A(_)),
    retractall(stored_B(_)),
    retractall(reflection_enabled(_)),
    retractall(stack_item(_)),
    retractall(transition(_,_,_,_,_)),
```

```
    retractall(decision_made(_)),

    % --- Setup for new run ---
    assertz(reflection_enabled(ReflectFlag)),
    set_global_stack([]),
    setup_base_transitions,
    write('Starting run with reflection='), write(ReflectFlag), nl, nl,

    % --- Start processing ---
    step(Start, Input, [], Result).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%         Main Processing Step         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% step(+State, +Input, +Stack, -Result)
%
% The main recursive predicate that drives the PDA. In each step, it
% determines the next action based on the current state, input, and stack,
% then calls itself with the updated parameters. It handles terminal states,
% special decision points, and the reflective loop.
step(State, Input, Stack, Result) :-
    print_config(State, Input, Stack),

    % --- Handle Terminal States ---
    ( State == 4 ->
        Result = accept,
        write('*** ACCEPT reached. ***'), nl
    ; State == 5 ->
        Result = error,
        write('*** ERROR reached. ***'), nl

    % --- Handle State 3: Decision Phase ---
    ; State == 3, \+ decision_made(_) ->
        !,
        make_decision_at_q3(Stack, Decision),
        assertz(decision_made(Decision)),
        setup_q3_transition(Decision),
        step(State, Input, Stack, Result)

    % --- Handle State 6: Reflection Loop ---
    ; State == 6 ->
        handle_reflection_state(State, Input, Stack, Result)

    % --- Default Transition Handling ---
    ; select_transition(State, Input, Stack, NextState, NextInput, NextStack, Action) ->
        print_transition(State, Input, Action, NextState),
        step(NextState, NextInput, NextStack, Result)

    % --- No Transition Found ---
    ; write('*** ERROR: No transition found from state '), print_state(State),
      write(' with input '), write(Input), write(' and stack '), write(Stack), nl,
      Result = error
    ).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%        State-Specific Logic          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```prolog
% --- State 3: Decision Making & Transition Setup ---
% make_decision_at_q3(+Stack, -Decision)
%
% Determines the next step from state q3. It decodes the numbers A and B
% from the stack, calculates the required transfer amount K, and then decides
% whether to (1) rearrange, (2) enter the reflective state, or (3) error out.
make_decision_at_q3(Stack, Decision) :-
    decode_stack_final(Stack, A, B, K, Possible),
    ( Possible == error ->
        write('Decision@q3: Stack format error.'), nl,
        Decision = error
    ; reflection_enabled(RF), RF == y, base(Base), A =:= (Base - 6), B >= 6 ->
        write('Decision@q3: Conditions met for Reflection (k=6).'), nl,
        Decision = reflect
    ; B >= K ->
        write('Decision@q3: Conditions met for Rearrangement (Accept).'), nl,
        Decision = accept
    ;
        write('Decision@q3: B < K, cannot rearrange standardly. Error.'), nl,
        Decision = error
    ).

% setup_q3_transition(+Decision)
%
% Dynamically asserts the transition rule leading out of state q3 based on
% the decision made by make_decision_at_q3/2.
setup_q3_transition(accept) :-
    assertz(transition(3, epsilon, 7, rearrange_action, no)),
    print_dynamic_transition(3, epsilon, 7, rearrange_action).
setup_q3_transition(error) :-
    assertz(transition(3, epsilon, 5, noop, no)),
    print_dynamic_transition(3, epsilon, 5, noop).
setup_q3_transition(reflect) :-
    assertz(transition(3, epsilon, 6, setup_reflect_stack, no)),
    print_dynamic_transition(3, epsilon, 6, setup_reflect_stack).

% --- State 6: Reflection Loop Handling ---
% handle_reflection_state(+State, +Input, +Stack, -Result)
%
% Manages the logic for the special reflective state q6. It checks the top of
% the stack. If it's 0, the loop halts and transitions to the accept state.
% Otherwise, it applies a "reflect_add_6_step" action to the stack and loops
% back to q6.
handle_reflection_state(State, Input, Stack, Result) :-
    Stack = [CurrentBmodBase | _RestStack],
    ( CurrentBmodBase == 0 ->
        write('State q6: Halt condition met (Stack top == 0). Transitioning to Accept (q4).'), nl,
        NextState = 4,
        NextInput = Input,
        NextStack = Stack,
        print_pseudo_transition(State, 'halt_check', NextState),
        step(NextState, NextInput, NextStack, Result)
    ;
        write('State q6: Continuing reflection loop...'), nl,
        Action = reflect_add_6_step,
        apply_action(Action, Stack, NextStack),
        NextState = 6,
        NextInput = Input,
        print_pseudo_transition(State, Action, NextState),
```

```prolog
        step(NextState, NextInput, NextStack, Result)
    ).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%         Transition Selection         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Select transition based on input symbol
% Modified to apply action and return the resulting NextStack
select_transition(State, [Sym|RestInput], Stack, NextState, RestInput, NextStack, Action) :-
    transition(State, Sym, NextState, Action, _),
    !,
    apply_action(Action, Stack, NextStack).

% Select epsilon transition if no symbol match
% Modified to apply action and return the resulting NextStack
select_transition(State, Input, Stack, NextState, Input, NextStack, Action) :-
    transition(State, epsilon, NextState, Action, _),
    !,
    apply_action(Action, Stack, NextStack).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%           Action Handlers            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Dispatcher for actions - Actions NOW update global stack if they modify it

apply_action(noop, Stack, Stack).

apply_action(push(X), Stack, NewStack) :-
    (digit(X) ; X == '#'), !,
    NewStack = [X|Stack],
    set_global_stack(NewStack).

apply_action(pop, [_|Stack], NewStack) :- !,
    NewStack = Stack,
    set_global_stack(NewStack).
apply_action(pop, [], []) :- !,
    write('Warning: Pop attempted on empty stack.'), nl.

apply_action(rearrange_action, InitialStack, FinalStack) :-
    write('Action: Performing RMB rearrangement...'), nl,
    rearrange_stack(InitialStack, FinalStack),
    !.

apply_action(setup_reflect_stack, Stack, NewStack) :-
    write('Action: Setting up stack for reflection state q6...'), nl,
    split_at_hash(Stack, APart, BPart),
    digits_to_num(BPart, B),
    base(Base),
    BmodBase is B mod Base,
    append(['#'], APart, RestOfStack),
    NewStack = [BmodBase | RestOfStack],
    write(' -> New stack top for B (mod Base): '), write(BmodBase), nl,
    set_global_stack(NewStack),!.

apply_action(reflect_add_6_step, Stack, NewStack) :-
```

```
    Stack = [CurrentB | Rest], !,
    base(Base),
    K_reflect is 6,
    NewB is (CurrentB + K_reflect) mod Base,
    write('Action: Reflection step: '),
    write(CurrentB), write(' + '), write(K_reflect), write(' mod '), write(Base), write(' = '), writ
    NewStack = [NewB | Rest],
    set_global_stack(NewStack).

apply_action(Action, Stack, Stack) :-
    write('Warning: Unknown action encountered: '), write(Action), nl.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       RMB Rearrangement Logic        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Modified to use InitialStack argument instead of current_stack
rearrange_stack(InitialStack, FinalStack) :-
    decode_stack_final(InitialStack, A, B, K, Possible),
    ( Possible == ok, B >= K ->
        base(Base),
        Anew is A + K,
        Bnew is B - K,
        write(' -> Rearranging: A='), write(A), write(', B='), write(B),
        write(', K='), write(K), nl,
        write('    -> New A=(A+K)='), write(Anew),
        write(', New B=(B-K)='), write(Bnew), nl,
        num_to_digits(Anew, AnewDigits),
        num_to_digits(Bnew, BnewDigits),
        reverse(BnewDigits, RevB),
        reverse(AnewDigits, RevA),
        append(RevB, ['#'|RevA], NewStackReversed),
        reverse(NewStackReversed, FinalStack),
        set_global_stack(FinalStack),
        write(' -> Rearrangement complete. New stack: '), write(FinalStack), nl
    ;
      write('Error: Rearrange action called inappropriately or decode failed.'), nl,
      FinalStack = InitialStack
    ).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%           Stack & Arithmetic         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Decode stack into A, B, K. Returns 'ok' or 'error' in Possible.
% Operates purely on the input Stack argument.
decode_stack_final(Stack, A, B, K, Possible) :-
    ( member('#', Stack) ->
        split_at_hash(Stack, APart, BPart),
        ( digits_to_num(APart, A), digits_to_num(BPart, B) ->
            retractall(stored_A(_)), retractall(stored_B(_)),
            assertz(stored_A(A)), assertz(stored_B(B)),
            base(Base),
            ( A =< Base -> K is Base - A, Possible = ok
            ; write('Error: Decoded A > Base.'), nl, Possible = error
            )
        ; write('Error: Failed to convert digits to numbers.'), nl, Possible = error, A = -1, B = -1
        )
    ;
```

```prolog
        write('Error: Stack missing "#" separator.'), nl,
        Possible = error, A = -1, B = -1, K = -1
    ).

% Split stack list at '#' marker
split_at_hash(Stack, APart, BPart) :-
    reverse(Stack, RevStack),
    append(RevA, ['#'|RevB], RevStack), !,
    reverse(RevA, APart),
    reverse(RevB, BPart).

% Convert list of digits to number
digits_to_num(Digs, N) :-
    foldl(add_digit, Digs, 0, N).
add_digit(D, Acc, Val) :- Val is Acc*10 + D.

% Convert number to list of digits
num_to_digits(0, [0]) :- !.
num_to_digits(N, Digs) :- N > 0, num_to_digits_acc(N, [], Digs).
num_to_digits_acc(0, Acc, Acc) :- !.
num_to_digits_acc(N, Acc, Digs) :-
    N > 0,
    D is N mod 10,
    N1 is N // 10,
    num_to_digits_acc(N1, [D|Acc], Digs).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%          Global Stack Access          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Update the global stack representation (used by actions that modify stack)
set_global_stack(NewStack) :-
    retractall(stack_item(_)),
    forall(member(E, NewStack), assertz(stack_item(E))).

% Retrieve the current global stack (ONLY for external query/debug if needed)
% Note: Main logic should rely on stack passed through step/4 arguments.
current_stack_global(Stack) :-
    findall(X, stack_item(X), S),
    reverse(S, Stack).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%      Static Transition Setup          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

setup_base_transitions :-
    % q1: reading A until '+'
    forall(digit(D), assertz(transition(1, D, 1, push(D), no))),
    assertz(transition(1, '+', 2, push('#'), no)),
    % q2: reading B digits until end of input
    forall(digit(D), assertz(transition(2, D, 2, push(D), no))),
    assertz(transition(2, epsilon, 3, noop, no)),
    % q7: after successful rearranging, go to q4 (accept)
    assertz(transition(7, epsilon, 4, noop, no)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%        Printing & Debug Helpers       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```prolog
% Modified to print the Stack argument passed to it.
print_config(State, Input, Stack) :-
    write('--------------------------------------'), nl,
    write('State: '), print_state(State),
    write(' | Input: '), write(Input),
    write(' | Stack: '), write(Stack), nl.

print_state(S) :- write('q'), write(S).

% Print standard transitions found via transition/5
print_transition(SFrom, Input, Action, STo) :-
    ( Input == [] -> InputSym = 'epsilon'
    ; Input = [InputSym|_]
    ),
    write('Transition: '), print_state(SFrom),
    write(' --['), write(InputSym), write(':'), write(Action), write(']--> '),
    print_state(STo), nl.

% Print dynamically added transitions from q3
print_dynamic_transition(SFrom, Sym, STo, Action) :-
     write('Dynamically Added Transition: '), print_state(SFrom),
     write(' --['), write(Sym), write(':'), write(Action), write(']--> '),
     print_state(STo), nl.

% Print pseudo-transitions decided within state 6 logic
print_pseudo_transition(SFrom, ActionOrCheck, STo) :-
     write('State q6 Logic: '), print_state(SFrom),
     write(' --['), write('epsilon'), write(':'), write(ActionOrCheck), write(']--> '),
     print_state(STo), nl.
```

26 sar_add_chunking.pl

```prolog
/** <module> Student Addition Strategy: Chunking by Bases and Ones
 *
 * This module implements the 'Chunking by Bases and Ones' strategy for
 * multi-digit addition, modeled as a finite state machine. This strategy
 * involves decomposing one of the numbers (B) into its base-10 components
 * (e.g., tens and ones), adding them sequentially to the other number (A),
 * and using strategic 'chunks' to reach friendly base-10 numbers.
 *
 * The process is as follows:
 * 1. Decompose B into a 'base chunk' (the tens part) and an 'ones chunk'.
 * 2. Add the entire base chunk to A at once.
 * 3. Strategically add parts of the ones chunk to get the sum to the next multiple of 10.
 * 4. Repeat until all parts of B have been added.
 *
 * The state is represented by the term:
 * `state(Name, Sum, BasesRem, OnesRem, K, InternalSum, TargetBase)`
 *
 * The history of execution is captured as a list of steps:
 * `step(StateName, CurrentSum, BasesRemaining, OnesRemaining, K, Interpretation)`
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(sar_add_chunking,
          [ run_chunking/4
          ]).
```

```prolog
:- use_module(library(lists)).

%!      run_chunking(+A:integer, +B:integer, -FinalSum:integer, -History:list) is det.
%
%       Executes the 'Chunking by Bases and Ones' addition strategy for A + B.
%
%       This predicate initializes the state machine and runs it until it
%       reaches the accept state. It traces the execution, providing a
%       step-by-step history of how the sum was computed.
%
%       @param A The first addend.
%       @param B The second addend, which will be decomposed and added in chunks.
%       @param FinalSum The resulting sum of A and B.
%       @param History A list of `step/6` terms that describe the state
%       machine's execution path and the interpretation of each step.

run_chunking(A, B, FinalSum, History) :-
    Base = 10,
    % Initial state (q_init): Decompose B and set the initial sum.
    Sum is A,
    BasesRemaining is (B // Base) * Base,
    OnesRemaining is B mod Base,

    format(string(InitialInterpretation), 'Initialize Sum to ~w. Decompose B: ~w + ~w.', [A, BasesRe
    InitialHistoryEntry = step(q_start, A, 0, 0, 0, InitialInterpretation),

    InitialState = state(q_init, Sum, BasesRemaining, OnesRemaining, 0, 0, 0),

    % Run the state machine.
    run(InitialState, Base, [InitialHistoryEntry], ReversedHistory),
    reverse(ReversedHistory, History),

    % Extract the final sum from the last history entry.
    (last(History, step(_, FinalSum, _, _, _, _)) -> true ; FinalSum = A).

% run/4 is the main loop of the state machine. It stops at the q_accept state.
run(state(q_accept, Sum, BR, OR, K, _IS, _TB), _Base, Acc, FinalHistory) :-
    HistoryEntry = step(q_accept, Sum, BR, OR, K, 'Execution finished.'),
    FinalHistory = [HistoryEntry | Acc].

run(CurrentState, Base, Acc, FinalHistory) :-
    transition(CurrentState, Base, NextState, Interpretation),
    CurrentState = state(Name, Sum, BR, OR, K, _, _),
    HistoryEntry = step(Name, Sum, BR, OR, K, Interpretation),
    run(NextState, Base, [HistoryEntry | Acc], FinalHistory).

% transition/4 defines the state transitions of the finite state machine.

% From q_init, always proceed to add the base chunk.
transition(state(q_init, Sum, BR, OR, K, IS, TB), _Base, state(q_add_base_chunk, Sum, BR, OR, K, IS,
           'Proceed to add base chunk.').

% From q_add_base_chunk:
% If there are bases remaining, add them all at once.
transition(state(q_add_base_chunk, Sum, BR, OR, _K, _IS, _TB), _Base, state(q_init_ones_chunk, NewSu
    BR > 0,
    NewSum is Sum + BR,
    format(string(Interpretation), 'Add Base Chunk (+~w). Sum = ~w.', [BR, NewSum]).
% If there are no bases, move on.
```

```
transition(state(q_add_base_chunk, Sum, 0, OR, _K, _IS, _TB), _Base, state(q_init_ones_chunk, Sum, 0
           'No bases to add.').

% From q_init_ones_chunk:
% If there are ones to add, start the strategic chunking process.
transition(state(q_init_ones_chunk, Sum, BR, OR, K, _IS, _TB), _Base, state(q_init_K, Sum, BR, OR, K
    OR > 0,
    format(string(Interpretation), 'Begin strategic chunking of remaining ones (~w).', [OR]),
    (Sum > 0, Sum mod 10 =\= 0 -> TargetBase is ((Sum // 10) + 1) * 10 ; TargetBase is Sum).
% If no ones are left, the process is finished.
transition(state(q_init_ones_chunk, Sum, _, 0, _, _, _), _Base, state(q_accept, Sum, 0, 0, 0, 0, 0),
           'All ones added. Accepting.').

% From q_init_K, calculate the value K needed to reach the next base.
transition(state(q_init_K, Sum, BR, OR, _, IS, TB), _Base, state(q_loop_K, Sum, BR, OR, 0, IS, TB),
    format(string(Interpretation), 'Calculating K: Counting from ~w to ~w.', [Sum, TB]).

% From q_loop_K, count up from the current sum to the target base to find K.
transition(state(q_loop_K, Sum, BR, OR, K, IS, TB), _Base, state(q_loop_K, Sum, BR, OR, NewK, NewIS,
    IS < TB,
    NewIS is IS + 1,
    NewK is K + 1,
    format(string(Interpretation), 'Counting Up: ~w, K=~w', [NewIS, NewK]).
% Once the target base is reached, the value of K is known.
transition(state(q_loop_K, Sum, BR, OR, K, IS, TB), _Base, state(q_add_ones_chunk, Sum, BR, OR, K, I
    IS >= TB,
    format(string(Interpretation), 'K needed to reach base is ~w.', [K]).

% From q_add_ones_chunk:
% If we have enough ones remaining to add the strategic chunk K, do so.
transition(state(q_add_ones_chunk, Sum, BR, OR, K, _IS, _TB), _Base, state(q_init_ones_chunk, NewSum
    OR >= K, K > 0,
    NewSum is Sum + K,
    NewOR is OR - K,
    format(string(Interpretation), 'Add Strategic Chunk (+~w) to make base. Sum = ~w.', [K, NewSum])
% Otherwise, add all remaining ones. This happens if K is too large or 0.
transition(state(q_add_ones_chunk, Sum, BR, OR, K, _IS, _TB), _Base, state(q_init_ones_chunk, NewSum
    (OR < K ; K =< 0), OR > 0,
    NewSum is Sum + OR,
    format(string(Interpretation), 'Add Remaining Chunk (+~w). Sum = ~w.', [OR, NewSum]).
```

27 sar_add_cobo.pl

```
/** <module> Student Addition Strategy: Counting On by Bases and Ones (COBO)
 *
 * This module implements the 'Counting On by Bases and then Ones' (COBO)
 * strategy for multi-digit addition, modeled as a finite state machine.
 * This strategy involves decomposing one number (B) into its base-10
 * components and then incrementally counting on from the other number (A).
 *
 * The process is as follows:
 * 1. Decompose B into a number of 'bases' (tens) and 'ones'.
 * 2. Starting with A, count on by ten for each base.
 * 3. After all bases are added, count on by one for each one.
 *
 * The state of the automaton is represented by the term:
 * `state(StateName, Sum, BaseCounter, OneCounter)`
 *
 * The history of execution is captured as a list of steps:
```

```
 * `step(StateName, CurrentSum, BaseCounter, OneCounter, Interpretation)`
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(sar_add_cobo,
          [ run_cobo/4
          ]).

:- use_module(library(lists)).

%!      run_cobo(+A:integer, +B:integer, -FinalSum:integer, -History:list) is det.
%
%       Executes the 'Counting On by Bases and Ones' (COBO) addition strategy for A + B.
%
%       This predicate initializes the state machine and runs it until it
%       reaches the accept state. It traces the execution, providing a
%       step-by-step history of how the sum was computed by first counting
%       on by tens, and then by ones.
%
%       @param A The first addend, the number to start counting from.
%       @param B The second addend, which is decomposed into bases and ones.
%       @param FinalSum The resulting sum of A and B.
%       @param History A list of `step/5` terms that describe the state
%       machine's execution path and the interpretation of each step.

run_cobo(A, B, FinalSum, History) :-
    Base = 10,
    % Initial state: Decompose B into base and one counters.
    BaseCounter is B // Base,
    OneCounter is B mod Base,

    InitialState = state(q_initialize, A, BaseCounter, OneCounter),

    % Record the start and the interpretation of the initialization.
    format(string(InitialInterpretation), 'Initialize Sum to ~w. Decompose ~w into ~w Bases, ~w Ones
    InitialHistoryEntry = step(q_start, A, BaseCounter, OneCounter, InitialInterpretation),

    % Run the state machine.
    run(InitialState, Base, [InitialHistoryEntry], ReversedHistory),

    % Reverse the history for correct chronological order.
    reverse(ReversedHistory, History),

    % Extract the final sum from the last history entry.
    (last(History, step(_, FinalSum, _, _, _)) -> true ; FinalSum = A).


% run/4 is the main recursive loop of the state machine.
% It drives the state transitions until the accept state is reached.

% Base case: Stop when the machine reaches the 'q_accept' state.
run(state(q_accept, Sum, BC, OC), _Base, AccHistory, FinalHistory) :-
    Interpretation = 'All ones added. Accept.',
    HistoryEntry = step(q_accept, Sum, BC, OC, Interpretation),
    FinalHistory = [HistoryEntry | AccHistory].

% Recursive step: Perform one transition and continue.
run(CurrentState, Base, AccHistory, FinalHistory) :-
```

```
    transition(CurrentState, Base, NextState, Interpretation),
    CurrentState = state(Name, Sum, BC, OC),
    HistoryEntry = step(Name, Sum, BC, OC, Interpretation),
    run(NextState, Base, [HistoryEntry | AccHistory], FinalHistory).

% transition/4 defines the logic for moving from one state to the next.

% From q_initialize, always transition to q_add_bases to start counting.
transition(state(q_initialize, Sum, BaseCounter, OneCounter), _Base, state(q_add_bases, Sum, BaseCou
    Interpretation = 'Begin counting on by bases.'.

% Loop in q_add_bases, counting on by one base (10) at a time.
transition(state(q_add_bases, Sum, BaseCounter, OneCounter), Base, state(q_add_bases, NewSum, NewBas
    BaseCounter > 0,
    NewSum is Sum + Base,
    NewBaseCounter is BaseCounter - 1,
    format(string(Interpretation), 'Count on by base: ~w -> ~w.', [Sum, NewSum]).

% When all bases are added, transition from q_add_bases to q_add_ones.
transition(state(q_add_bases, Sum, 0, OneCounter), _Base, state(q_add_ones, Sum, 0, OneCounter), Int
    Interpretation = 'All bases added. Transition to adding ones.'.

% Loop in q_add_ones, counting on by one at a time.
transition(state(q_add_ones, Sum, BaseCounter, OneCounter), _Base, state(q_add_ones, NewSum, BaseCou
    OneCounter > 0,
    NewSum is Sum + 1,
    NewOneCounter is OneCounter - 1,
    format(string(Interpretation), 'Count on by one: ~w -> ~w.', [Sum, NewSum]).

% When all ones are added, transition from q_add_ones to the final accept state.
transition(state(q_add_ones, Sum, BaseCounter, 0), _Base, state(q_accept, Sum, BaseCounter, 0), Inte
    Interpretation = 'All ones added. Final sum reached.'.
```

28 sar_add_rmb.pl

```
/** <module> Student Addition Strategy: Rearranging to Make Bases (RMB)
 *
 * This module implements the 'Rearranging to Make Bases' (RMB) strategy for
 * addition, modeled as a finite state machine. This is a sophisticated
 * strategy where a student rearranges quantities between the two addends
 * to create a "friendly" number (a multiple of 10), simplifying the final calculation.
 *
 * The process is as follows:
 * 1. Identify the larger number (A) and the smaller number (B).
 * 2. Calculate how much A needs to reach the next multiple of 10. This amount is K.
 * 3. "Take" K from B and "give" it to A. This is a decomposition and recombination step.
 * 4. The new problem becomes (A + K) + (B - K).
 * 5. The strategy fails if B is smaller than K.
 *
 * The state is represented by the term:
 * `state(Name, A, B, K, A_temp, B_temp, TargetBase, B_initial)`
 *
 * The history of execution is captured as a list of steps:
 * `step(Name, A, B, K, A_temp, B_temp, Interpretation)`
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(sar_add_rmb,
```

```prolog
          [ run_rmb/4
          ]).

:- use_module(library(lists)).

%!      run_rmb(+A_in:integer, +B_in:integer, -FinalResult:integer, -History:list) is det.
%
%       Executes the 'Rearranging to Make Bases' (RMB) addition strategy for A + B.
%
%       This predicate initializes and runs a state machine that models the RMB
%       strategy. It first determines the amount `K` needed for the larger number
%       to reach a multiple of 10, then transfers `K` from the smaller number.
%       It traces the execution, providing a step-by-step history.
%
%       @param A_in The first addend.
%       @param B_in The second addend.
%       @param FinalResult The resulting sum of A and B. If the strategy
%       fails (because the smaller addend is less than K), this will be the
%       atom `'error'`.
%       @param History A list of `step/7` terms that describe the state
%       machine's execution path and the interpretation of each step.

run_rmb(A_in, B_in, FinalResult, History) :-
    Base = 10,
    % Ensure A is the larger number and B is the smaller.
    A is max(A_in, B_in),
    B is min(A_in, B_in),

    % Initial state (q_calc_K): Determine K needed to get A to a multiple of 10.
    (A mod Base =:= 0, A =\= 0 -> TargetBase is A ; TargetBase is ((A // Base) + 1) * Base),
    InitialState = state(q_calc_K, A, B, 0, A, 0, TargetBase, B), % B_initial stored for error msg

    InitialInterpretation = 'Start. Determine larger number and target base.',
    InitialHistoryEntry = step(q_start, A, B, 0, 0, 0, InitialInterpretation),

    run(InitialState, Base, [InitialHistoryEntry], ReversedHistory),
    reverse(ReversedHistory, History),

    % Check the final state to determine the result.
    (last(History, step(q_accept, FinalA, FinalB, _, _, _, _)) ->
        FinalResult is FinalA + FinalB
    ;
        FinalResult = 'error'
    ).

% run/4 is the main recursive loop of the state machine.

% Base case: Stop when the machine reaches the 'q_accept' state.
run(state(q_accept, A, B, K, AT, BT, _, _), _, Acc, FinalHistory) :-
    Result is A + B,
    format(string(Interpretation), 'Combine rearranged numbers: ~w + ~w = ~w.', [A, B, Result]),
    HistoryEntry = step(q_accept, A, B, K, AT, BT, Interpretation),
    FinalHistory = [HistoryEntry | Acc].

% Recursive step: Perform one transition and continue.
run(CurrentState, Base, Acc, FinalHistory) :-
    transition(CurrentState, Base, NextState, Interpretation),
    CurrentState = state(Name, A, B, K, AT, BT, _, _),
    HistoryEntry = step(Name, A, B, K, AT, BT, Interpretation),
```

```
    run(NextState, Base, [HistoryEntry | Acc], FinalHistory).

% transition/4 defines the logic for moving from one state to the next.

% In q_calc_K, count up from A to the target base to determine K.
transition(state(q_calc_K, A, B, K, AT, BT, TB, B_init), _, state(q_calc_K, A, B, NewK, NewAT, BT, T
    AT < TB,
    NewAT is AT + 1,
    NewK is K + 1,
    format(string(Interpretation), 'Count up: ~w. Distance (K): ~w.', [NewAT, NewK]).
% Once K is found, transition to q_decompose_B to transfer K from B.
transition(state(q_calc_K, A, B, K, AT, _BT, TB, B_init), _, state(q_decompose_B, A, B, K, AT, B, TB
    AT >= TB,
    format(string(Interpretation), 'K needed is ~w. Start counting down K from B.', [K]).

% In q_decompose_B, "transfer" K from B to A by decrementing both K and a temp copy of B.
transition(state(q_decompose_B, A, B, K, AT, BT, TB, B_init), _, state(q_decompose_B, A, B, NewK, AT
    K > 0, BT > 0,
    NewK is K - 1,
    NewBT is BT - 1,
    format(string(Interpretation), 'Transferred 1. B remainder: ~w. K remaining: ~w.', [NewBT, NewK]
% Once K is fully transferred (K=0), recombine the numbers.
transition(state(q_decompose_B, _, _, 0, AT, BT, _, _), _, state(q_recombine, AT, BT, 0, AT, BT, 0,
    format(string(Interpretation), 'Decomposition Complete. New state: A=~w, B=~w.', [AT, BT]).
% If B runs out before K is transferred, the strategy fails.
transition(state(q_decompose_B, _, _, K, _, 0, _, B_init), _, state(q_error, 0,0,0,0,0,0,0), Interpr
    K > 0,
    format(string(Interpretation), 'Strategy Failed. B (~w) is too small to provide K (~w).', [B_ini

% From q_recombine, proceed to the final accept state.
transition(state(q_recombine, A, B, K, AT, BT, _, _), _, state(q_accept, A, B, K, AT, BT, 0, 0), 'Pr
```

29 sar_add_rounding.pl

```
/** <module> Student Addition Strategy: Rounding and Adjusting
 *
 * This module implements the 'Rounding and Adjusting' strategy for addition,
 * modeled as a multi-phase finite state machine. The strategy involves
 * simplifying an addition problem by rounding one number up to a multiple of 10,
 * performing the addition, and then adjusting the result.
 *
 * The process is as follows:
 * 1.   **Phase 1: Rounding**: Select one number (`Target`) to round up, typically
 *      the one closer to the next multiple of 10. Calculate the amount `K`
 *      needed for rounding.
 * 2.   **Phase 2: Addition**: Add the *rounded* number to the other number. This
 *      is performed using a 'Counting On by Bases and Ones' (COBO) sub-strategy.
 * 3.   **Phase 3: Adjustment**: Adjust the sum from Phase 2 by subtracting `K`
 *      to get the final, correct answer.
 *
 * The state is represented by the complex term:
 * `state(Name, K, A_rounded, TempSum, Result, Target, Other, TargetBase, BaseCounter, OneCounter)`
 *
 * The history of execution is captured as a list of steps:
 * `step(Name, K, RoundedTarget, TempSum, CurrentResult, Interpretation)`
 *
 * @author Tilo Wiedera
 * @license MIT
 */
```

```prolog
:- module(sar_add_rounding,
          [ run_rounding/4
          ]).

:- use_module(library(lists)).

% determine_target/5 is a helper to decide which number to round.
% It selects the number that is closer to the next multiple of the base.
determine_target(A_in, B_in, Base, Target, Other) :-
    A_rem is A_in mod Base,
    B_rem is B_in mod Base,
    (A_rem >= B_rem ->
        (Target = A_in, Other = B_in)
    ;
        (Target = B_in, Other = A_in)
    ).

%!      run_rounding(+A_in:integer, +B_in:integer, -FinalResult:integer, -History:list) is det.
%
%       Executes the 'Rounding and Adjusting' addition strategy for A + B.
%
%       This predicate initializes and runs a state machine that models the
%       three phases of the strategy: rounding, adding, and adjusting.
%       It traces the entire execution, providing a step-by-step history
%       of the cognitive process.
%
%       @param A_in The first addend.
%       @param B_in The second addend.
%       @param FinalResult The resulting sum of A and B.
%       @param History A list of `step/6` terms that describe the state
%       machine's execution path and the interpretation of each step.

run_rounding(A_in, B_in, FinalResult, History) :-
    Base = 10,
    determine_target(A_in, B_in, Base, Target, Other),

    % Initial state (q_init_K): Determine K and the target base for rounding.
    (Target =< 0 -> TB = 0 ; (Target mod Base =:= 0 -> TB = Target ; TB is ((Target // Base) + 1) *
    InitialState = state(q_init_K, 0, Target, 0, 0, Target, Other, TB, 0, 0),

    format(string(InitialInterpretation), 'Inputs: ~w, ~w. Target for rounding: ~w', [A_in, B_in, Ta
    InitialHistoryEntry = step(q_start, 0, 0, 0, 0, InitialInterpretation),

    run(InitialState, Base, [InitialHistoryEntry], ReversedHistory),
    reverse(ReversedHistory, History),

    % Extract the final result from the last history entry.
    (last(History, step(q_accept, _, _, _, R, _)) -> FinalResult = R ; FinalResult = 'error').

% run/4 is the main recursive loop of the state machine.
run(state(q_accept, K, AR, TS, Result, _, _, _, _, _), _, Acc, FinalHistory) :-
    HistoryEntry = step(q_accept, K, AR, TS, Result, 'Execution finished.'),
    FinalHistory = [HistoryEntry | Acc].

run(CurrentState, Base, Acc, FinalHistory) :-
    transition(CurrentState, Base, NextState, Interpretation),
    CurrentState = state(Name, K, AR, TS, Result, _, _, _, _, _),
    HistoryEntry = step(Name, K, AR, TS, Result, Interpretation),
    run(NextState, Base, [HistoryEntry | Acc], FinalHistory).
```

```
% transition/4 defines the logic for moving from one state to the next.

% Phase 1: Rounding
transition(state(q_init_K, K, AR, TS, R, T, O, TB, BC, OC), _, state(q_loop_K, K, AR, TS, R, T, O, T
    format(string(Interp), 'Initializing K calculation. Counting from ~w to ~w.', [T, TB]).

transition(state(q_loop_K, K, AR, TS, R, T, O, TB, BC, OC), _, state(q_loop_K, NewK, NewAR, TS, R, T
    AR < TB,
    NewK is K + 1, NewAR is AR + 1,
    format(string(Interp), 'Counting Up: ~w, K=~w', [NewAR, NewK]).
transition(state(q_loop_K, K, AR, TS, R, T, O, TB, BC, OC), _, state(q_init_Add, K, AR, TS, R, T, O,
    AR >= TB,
    format(string(Interp), 'K needed is ~w. Target rounded to ~w.', [K, AR]).

% Phase 2: Addition (using COBO sub-strategy)
transition(state(q_init_Add, K, AR, _TS, R, T, O, TB, _BC, _OC), Base, state(q_loop_AddBases, K, AR,
    OBC is O // Base, OOC is O mod Base,
    format(string(Interp), 'Initializing COBO: ~w + ~w. (Bases: ~w, Ones: ~w)', [AR, O, OBC, OOC]).

transition(state(q_loop_AddBases, K, AR, TS, R, T, O, TB, BC, OC), Base, state(q_loop_AddBases, K, A
    BC > 0,
    NewTS is TS + Base, NewBC is BC - 1,
    format(string(Interp), 'COBO (Base): ~w', [NewTS]).
transition(state(q_loop_AddBases, K, AR, TS, R, T, O, TB, 0, OC), _, state(q_loop_AddOnes, K, AR, TS
             'COBO Bases complete.').

transition(state(q_loop_AddOnes, K, AR, TS, R, T, O, TB, BC, OC), _, state(q_loop_AddOnes, K, AR, Ne
    OC > 0,
    NewTS is TS + 1, NewOC is OC - 1,
    format(string(Interp), 'COBO (One): ~w', [NewTS]).
transition(state(q_loop_AddOnes, K, AR, TS, R, T, O, TB, BC, 0), _, state(q_init_Adjust, K, AR, TS,
    format(string(Interp), '~w + ~w = ~w.', [AR, O, TS]).

% Phase 3: Adjustment
transition(state(q_init_Adjust, K, AR, TS, _, T, O, TB, BC, OC), _, state(q_loop_Adjust, K, AR, TS,
    format(string(Interp), 'Initializing Adjustment: Count back K=~w.', [K]).

transition(state(q_loop_Adjust, K, AR, TS, R, T, O, TB, BC, OC), _, state(q_loop_Adjust, NewK, AR, T
    K > 0,
    NewK is K - 1, NewR is R - 1,
    format(string(Interp), 'Counting Back: ~w', [NewR]).
transition(state(q_loop_Adjust, 0, AR, TS, R, T, _, _, _, _), _, state(q_accept, 0, AR, TS, R, T, _,
    Adj is AR - T,
    format(string(Interp), 'Subtracted Adjustment (~w). Final Result: ~w.', [Adj, R]).
```

30 sar_sub_cbbo_take_away.pl

```
/** <module> Student Subtraction Strategy: Counting Back By Bases and Ones (Take Away)
 *
 * This module implements the 'Counting Back by Bases and then Ones' (CBBO)
 * strategy for subtraction, often conceptualized as "taking away". It is
 * modeled as a finite state machine.
 *
 * The process is as follows:
 * 1. The subtrahend (S) is decomposed into its base-10 components (bases/tens and ones).
 * 2. Starting from the minuend (M), the strategy first "takes away" or
 *    counts back by the number of bases (tens).
 * 3. After all bases are subtracted, it counts back by the number of ones.
```

```
 * 4. The final value is the result of the subtraction.
 * 5. The strategy fails if the subtrahend is larger than the minuend.
 *
 * The state of the automaton is represented by the term:
 * `state(Name, CurrentValue, BaseCounter, OneCounter)`
 *
 * The history of execution is captured as a list of steps:
 * `step(Name, CurrentValue, BaseCounter, OneCounter, Interpretation)`
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(sar_sub_cbbo_take_away,
          [ run_cbbo_ta/4
          ]).

:- use_module(library(lists)).

%!      run_cbbo_ta(+M:integer, +S:integer, -FinalResult:integer, -History:list) is det.
%
%       Executes the 'Counting Back by Bases and Ones' (Take Away) subtraction
%       strategy for M - S.
%
%       This predicate initializes and runs a state machine that models the
%       CBBO strategy. It first checks if the subtraction is possible (M >= S).
%       If so, it decomposes S and simulates the process of counting back from M,
%       first by tens and then by ones. It traces the entire execution,
%       providing a step-by-step history.
%
%       @param M The Minuend, the number to subtract from.
%       @param S The Subtrahend, the number to subtract.
%       @param FinalResult The resulting difference (M - S). If S > M, this
%       will be the atom `'error'`.
%       @param History A list of `step/5` terms that describe the state
%       machine's execution path and the interpretation of each step.

run_cbbo_ta(M, S, FinalResult, History) :-
    Base = 10,
    (S > M ->
        History = [step(q_error, 0, 0, 0, 'Error: Subtrahend > Minuend.')],
        FinalResult = 'error'
    ;
        BC is S // Base,
        OC is S mod Base,
        InitialState = state(q_init, M, BC, OC),
        format(string(InitialInterpretation), 'Initialize at M (~w). Decompose S (~w): ~w bases, ~w
        InitialHistoryEntry = step(q_start, M, 0, 0, InitialInterpretation),

        run(InitialState, Base, [InitialHistoryEntry], ReversedHistory),
        reverse(ReversedHistory, History),

        (last(History, step(q_accept, CV, _, _, _)) ->
            FinalResult = CV
        ;
            FinalResult = 'error'
        )
    ).

% run/4 is the main recursive loop of the state machine.
```

```
run(state(q_accept, CV, BC, OC), _, Acc, FinalHistory) :-
    format(string(Interpretation), 'Subtraction finished. Result (Final Position) = ~w.', [CV]),
    HistoryEntry = step(q_accept, CV, BC, OC, Interpretation),
    FinalHistory = [HistoryEntry | Acc].

run(CurrentState, Base, Acc, FinalHistory) :-
    transition(CurrentState, Base, NextState, Interpretation),
    CurrentState = state(Name, CV, BC, OC),
    HistoryEntry = step(Name, CV, BC, OC, Interpretation),
    run(NextState, Base, [HistoryEntry | Acc], FinalHistory).

% transition/4 defines the logic for moving from one state to the next.

% From q_init, proceed to subtract the bases (tens).
transition(state(q_init, CV, BC, OC), _, state(q_sub_bases, CV, BC, OC),
           'Proceed to subtract bases.').

% Loop in q_sub_bases, counting back by one base (10) at a time.
transition(state(q_sub_bases, CV, BC, OC), Base, state(q_sub_bases, NewCV, NewBC, OC), Interp) :-
    BC > 0,
    NewCV is CV - Base,
    NewBC is BC - 1,
    format(string(Interp), 'Count back by base (-~w). New Value=~w.', [Base, NewCV]).
% When all bases are subtracted, transition to q_sub_ones.
transition(state(q_sub_bases, CV, 0, OC), _, state(q_sub_ones, CV, 0, OC),
           'Bases finished. Switching to ones.').

% Loop in q_sub_ones, counting back by one at a time.
transition(state(q_sub_ones, CV, BC, OC), _, state(q_sub_ones, NewCV, BC, NewOC), Interp) :-
    OC > 0,
    NewCV is CV - 1,
    NewOC is OC - 1,
    format(string(Interp), 'Count back by one (-1). New Value=~w.', [NewCV]).
% When all ones are subtracted, transition to the final accept state.
transition(state(q_sub_ones, CV, BC, 0), _, state(q_accept, CV, BC, 0),
           'Subtraction finished.').
```

31 sar_sub_chunking_a.pl

```
/** <module> Student Subtraction Strategy: Chunking Backwards by Place Value
 *
 * This module implements a "chunking" strategy for subtraction, modeled as a
 * finite state machine. The strategy involves subtracting the subtrahend (S)
 * from the minuend (M) in parts, based on place value (hundreds, tens, ones).
 *
 * The process is as follows:
 * 1. Identify the largest place-value chunk of the remaining subtrahend (S).
 *    For example, if S is 234, the first chunk is 200.
 * 2. Subtract this chunk from the current value (which starts at M).
 * 3. Repeat the process with the remainder of S. For S=234, the next chunk
 *    would be 30, then 4.
 * 4. The process ends when the entire subtrahend has been subtracted.
 * 5. The strategy fails if the subtrahend is larger than the minuend.
 *
 * The state of the automaton is represented by the term:
 * `state(Name, CurrentValue, S_Remaining, Chunk)`
 *
 * The history of execution is captured as a list of steps:
 * `step(Name, CurrentValue, S_Remaining, Chunk, Interpretation)`
```

```prolog
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(sar_sub_chunking_a,
          [ run_chunking_a/4
          ]).

:- use_module(library(lists)).
:- use_module(library(clpfd)). % For log/2

%!      run_chunking_a(+M:integer, +S:integer, -FinalResult:integer, -History:list) is det.
%
%       Executes the 'Chunking Backwards by Place Value' subtraction strategy for M - S.
%
%       This predicate initializes and runs a state machine that models the
%       chunking strategy. It first checks if the subtraction is possible (M >= S).
%       If so, it repeatedly identifies the largest place-value component of the
%       remaining subtrahend and subtracts it from the minuend. It traces
%       the entire execution, providing a step-by-step history.
%
%       @param M The Minuend, the number to subtract from.
%       @param S The Subtrahend, the number to subtract in chunks.
%       @param FinalResult The resulting difference (M - S). If S > M, this
%       will be the atom `'error'`.
%       @param History A list of `step/5` terms that describe the state
%       machine's execution path and the interpretation of each step.

run_chunking_a(M, S, FinalResult, History) :-
    Base = 10,
    (S > M ->
        History = [step(q_error, 0, 0, 0, 'Error: Subtrahend > Minuend.')],
        FinalResult = 'error'
    ;
        InitialState = state(q_init, M, S, 0),
        InitialHistoryEntry = step(q_start, 0, 0, 0, 'Start: Initialize.'),

        run(InitialState, Base, [InitialHistoryEntry], ReversedHistory),
        reverse(ReversedHistory, History),

        (last(History, step(q_accept, CV, _, _, _)) -> FinalResult = CV ; FinalResult = 'error')
    ).

% run/4 is the main recursive loop of the state machine.
run(state(q_accept, CV, 0, _), _, Acc, FinalHistory) :-
    format(string(Interpretation), 'S fully subtracted. Result=~w.', [CV]),
    HistoryEntry = step(q_accept, CV, 0, 0, Interpretation),
    FinalHistory = [HistoryEntry | Acc].

run(CurrentState, Base, Acc, FinalHistory) :-
    transition(CurrentState, Base, NextState, Interpretation),
    CurrentState = state(Name, CV, S_Rem, Chunk),
    HistoryEntry = step(Name, CV, S_Rem, Chunk, Interpretation),
    run(NextState, Base, [HistoryEntry | Acc], FinalHistory).

% transition/4 defines the logic for moving from one state to the next.

% From q_init, proceed to identify the first chunk.
transition(state(q_init, M, S, _), _, state(q_identify_chunk, M, S, 0), Interp) :-
```

```
    format(string(Interp), 'Set CurrentValue=~w. S_Remaining=~w.', [M, S]).

% In q_identify_chunk, determine the next chunk of S to subtract.
% The chunk is the largest part of S based on place value (e.g., hundreds, tens).
transition(state(q_identify_chunk, CV, S_Rem, _), Base, state(q_subtract_chunk, CV, S_Rem, Chunk), I
    S_Rem > 0,
    Power is floor(log(S_Rem) / log(Base)),
    PowerValue is Base^Power,
    Chunk is floor(S_Rem / PowerValue) * PowerValue,
    format(string(Interp), 'Identified chunk to subtract: ~w.', [Chunk]).
% If no subtrahend remains, the process is finished.
transition(state(q_identify_chunk, CV, 0, _), _, state(q_accept, CV, 0, 0),
           'S fully subtracted.').

% In q_subtract_chunk, perform the subtraction and loop back to identify the next chunk.
transition(state(q_subtract_chunk, CV, S_Rem, Chunk), _, state(q_identify_chunk, NewCV, NewSRem, 0),
    NewCV is CV - Chunk,
    NewSRem is S_Rem - Chunk,
    format(string(Interp), 'Subtracted ~w. New Value=~w.', [Chunk, NewCV]).
```

32 sar_sub_chunking_b.pl

```
/** <module> Student Subtraction Strategy: Chunking Forwards from Part (Missing Addend)
 *
 * This module implements a "counting up" or "missing addend" strategy for
 * subtraction (M - S), modeled as a finite state machine. It solves the
 * problem by calculating what needs to be added to S to reach M.
 *
 * The process is as follows:
 * 1. Start at the subtrahend (S). The goal is to reach the minuend (M).
 * 2. Identify a "strategic" chunk to add. This could be:
 *    a. The amount `K` needed to get from the current value to the next
 *       multiple of 10 (or 100, etc.).
 *    b. If that's not suitable, the largest possible place-value chunk of the
 *       *remaining distance* to M.
 * 3. Add the selected chunk. The size of the chunk is added to a running
 *    total, `Distance`.
 * 4. Repeat until the current value reaches M. The final `Distance` is the
 *    answer to the subtraction problem.
 * 5. The strategy fails if S > M.
 *
 * The state is represented by the term:
 * `state(Name, CurrentValue, Distance, K, TargetBase, InternalTemp, Minuend)`
 *
 * The history of execution is captured as a list of steps:
 * `step(Name, CurrentValue, Distance, K, Interpretation)`
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(sar_sub_chunking_b,
          [ run_chunking_b/4
          ]).

:- use_module(library(lists)).
:- use_module(library(clpfd)).

%!      run_chunking_b(+M:integer, +S:integer, -FinalResult:integer, -History:list) is det.
%
```

```prolog
%       Executes the 'Chunking Forwards from Part' (missing addend) subtraction
%       strategy for M - S.
%
%       This predicate initializes and runs a state machine that models the
%       "counting up" process. It first checks if the subtraction is possible (M >= S).
%       If so, it calculates the difference by adding chunks to S until it reaches M.
%       The sum of these chunks is the result. It traces the entire execution,
%       providing a step-by-step history.
%
%       @param M The Minuend, the target number to count up to.
%       @param S The Subtrahend, the number to start counting from.
%       @param FinalResult The resulting difference (M - S). If S > M, this
%       will be the atom `'error'`.
%       @param History A list of `step/5` terms that describe the state
%       machine's execution path and the interpretation of each step.

run_chunking_b(M, S, FinalResult, History) :-
    Base = 10,
    (S > M ->
        History = [step(q_error, 0, 0, 0, 'Error: Subtrahend > Minuend.')],
        FinalResult = 'error'
    ;
        InitialState = state(q_init, S, 0, 0, 0, 0, M),
        InitialHistoryEntry = step(q_start, 0, 0, 0, 'Start: Initialize.'),

        run(InitialState, Base, [InitialHistoryEntry], ReversedHistory),
        reverse(ReversedHistory, History),

        (last(History, step(q_accept, _, Dist, _, _)) -> FinalResult = Dist ; FinalResult = 'error')
    ).

% run/4 is the main recursive loop of the state machine.
run(state(q_accept, _, Dist, _, _, _, _), _, Acc, FinalHistory) :-
    format(string(Interpretation), 'Target reached. Result (Distance)=~w.', [Dist]),
    HistoryEntry = step(q_accept, 0, Dist, 0, Interpretation),
    FinalHistory = [HistoryEntry | Acc].

run(CurrentState, Base, Acc, FinalHistory) :-
    transition(CurrentState, Base, NextState, Interpretation),
    CurrentState = state(Name, CV, Dist, K, _, _, _),
    HistoryEntry = step(Name, CV, Dist, K, Interpretation),
    run(NextState, Base, [HistoryEntry | Acc], FinalHistory).

% transition/4 defines the logic for moving from one state to the next.

% From q_init, proceed to check if we are already at the target.
transition(state(q_init, S, _, _, _, _, M), _, state(q_check_status, S, 0, 0, 0, 0, M), Interp) :-
    format(string(Interp), 'Start at S (~w). Target is M (~w).', [S, M]).

% In q_check_status, decide whether to continue adding or accept the result.
transition(state(q_check_status, CV, Dist, _, _, _, M), _, state(q_init_K, CV, Dist, 0, 0, CV, M), '
    CV < M.
transition(state(q_check_status, M, Dist, _, _, _, M), _, state(q_accept, M, Dist, 0, 0, 0, M), 'Tar

% In q_init_K, determine the next friendly base number to aim for.
transition(state(q_init_K, CV, D, K, _, IT, M), Base, state(q_loop_K, CV, D, K, TB, IT, M), Interp)
    find_target_base(CV, M, Base, 1, TB),
    format(string(Interp), 'Calculating K: Counting from ~w to ~w.', [CV, TB]).
```

```
% In q_loop_K, count up to the target base to find the distance K.
transition(state(q_loop_K, CV, D, K, TB, IT, M), _, state(q_loop_K, CV, D, NewK, TB, NewIT, M), _) :
    IT < TB,
    NewIT is IT + 1,
    NewK is K + 1.
transition(state(q_loop_K, CV, D, K, TB, IT, M), _, state(q_add_chunk, CV, D, K, TB, IT, M), _) :-
    IT >= TB.

% In q_add_chunk, add a strategic chunk or a large place-value chunk.
transition(state(q_add_chunk, CV, D, K, _TB, _IT, M), Base, state(q_check_status, NewCV, NewD, 0, 0,
    Remaining is M - CV,
    (K > 0, K =< Remaining ->
        Chunk = K,
        format(string(Interp), 'Add strategic chunk (+~w) to reach base.', [Chunk])
    ;
        (Remaining > 0 ->
            Power is floor(log(Remaining) / log(Base)),
            PowerValue is Base^Power,
            C is floor(Remaining / PowerValue) * PowerValue,
            (C > 0 -> Chunk = C ; Chunk = Remaining),
            format(string(Interp), 'Add large/remaining chunk (+~w).', [Chunk])
        )
    ),
    NewCV is CV + Chunk,
    NewD is D + Chunk.

% find_target_base/5 is a helper to find the next "friendly" number to aim for.
find_target_base(CV, M, Base, Power, TargetBase) :-
    BasePower is Base^Power,
    (CV mod BasePower =\= 0 ->
        TargetBase is (floor(CV / BasePower) + 1) * BasePower
    ;
        (BasePower > M ->
            TargetBase = CV
        ;
            NewPower is Power + 1,
            find_target_base(CV, M, Base, NewPower, TargetBase)
        )
    ).
```

33 sar_sub_chunking_c.pl

```
/** <module> Student Subtraction Strategy: Chunking Backwards to Part
 *
 * This module implements a "counting down" or "take away in chunks" strategy
 * for subtraction (M - S), modeled as a finite state machine. It solves the
 * problem by calculating what needs to be subtracted from M to reach S.
 *
 * The process is as follows:
 * 1. Start at the minuend (M). The goal is to reach the subtrahend (S).
 * 2. Identify a "strategic" chunk to subtract. This could be:
 *    a. The amount `K` needed to get from the current value down to the next
 *       lower multiple of 10 (or 100, etc.).
 *    b. If that's not suitable, the largest possible place-value chunk of the
 *       *remaining distance* to S.
 * 3. Subtract the selected chunk. The size of the chunk is added to a running
 *    total, `Distance`.
 * 4. Repeat until the current value reaches S. The final `Distance` is the
 *    answer to the subtraction problem.
```

```prolog
 * 5. The strategy fails if S > M.
 *
 * The state is represented by the term:
 * `state(Name, CurrentValue, Distance, K, TargetBase, InternalTemp, S_target)`
 *
 * The history of execution is captured as a list of steps:
 * `step(Name, CurrentValue, Distance, K, Interpretation)`
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(sar_sub_chunking_c,
          [ run_chunking_c/4
          ]).

:- use_module(library(lists)).
:- use_module(library(clpfd)).

%!      run_chunking_c(+M:integer, +S:integer, -FinalResult:integer, -History:list) is det.
%
%       Executes the 'Chunking Backwards to Part' subtraction strategy for M - S.
%
%       This predicate initializes and runs a state machine that models the
%       "counting down" process. It first checks if the subtraction is possible (M >= S).
%       If so, it calculates the difference by subtracting chunks from M until it reaches S.
%       The sum of these chunks is the result. It traces the entire execution,
%       providing a step-by-step history.
%
%       @param M The Minuend, the number to start counting down from.
%       @param S The Subtrahend, the target number to reach.
%       @param FinalResult The resulting difference (M - S). If S > M, this
%       will be the atom `'error'`.
%       @param History A list of `step/5` terms that describe the state
%       machine's execution path and the interpretation of each step.

run_chunking_c(M, S, FinalResult, History) :-
    Base = 10,
    (S > M ->
        History = [step(q_error, 0, 0, 0, 'Error: Subtrahend > Minuend.')],
        FinalResult = 'error'
    ;
        InitialState = state(q_init, M, 0, 0, 0, 0, S),
        InitialHistoryEntry = step(q_start, 0, 0, 0, 'Start: Initialize.'),

        run(InitialState, Base, [InitialHistoryEntry], ReversedHistory),
        reverse(ReversedHistory, History),

        (last(History, step(q_accept, _, Dist, _, _)) -> FinalResult = Dist ; FinalResult = 'error')
    ).

% run/4 is the main recursive loop of the state machine.
run(state(q_accept, _, Dist, _, _, _, _), _, Acc, FinalHistory) :-
    format(string(Interpretation), 'Target reached. Result (Distance)=~w.', [Dist]),
    HistoryEntry = step(q_accept, 0, Dist, 0, Interpretation),
    FinalHistory = [HistoryEntry | Acc].

run(CurrentState, Base, Acc, FinalHistory) :-
    transition(CurrentState, Base, NextState, Interpretation),
    CurrentState = state(Name, CV, Dist, K, _, _, _),
```

```
    HistoryEntry = step(Name, CV, Dist, K, Interpretation),
    run(NextState, Base, [HistoryEntry | Acc], FinalHistory).

% transition/4 defines the logic for moving from one state to the next.

% From q_init, proceed to check if we are already at the target.
transition(state(q_init, M, _, _, _, _, S), _, state(q_check_status, M, 0, 0, 0, 0, S), Interp) :-
    format(string(Interp), 'Start at M (~w). Target is S (~w).', [M, S]).

% In q_check_status, decide whether to continue subtracting or accept the result.
transition(state(q_check_status, CV, Dist, _, _, _, S), _, state(q_init_K, CV, Dist, 0, 0, CV, S), '
    CV > S.
transition(state(q_check_status, S, Dist, _, _, _, S), _, state(q_accept, S, Dist, 0, 0, 0, S), 'Tar

% In q_init_K, determine the next friendly base number to aim for (counting down).
transition(state(q_init_K, CV, D, K, _, IT, S), Base, state(q_loop_K, CV, D, K, TB, IT, S), Interp)
    find_target_base_back(CV, S, Base, 1, TB),
    format(string(Interp), 'Calculating K: Counting back from ~w to ~w.', [CV, TB]).

% In q_loop_K, count down to the target base to find the distance K.
transition(state(q_loop_K, CV, D, K, TB, IT, S), _, state(q_loop_K, CV, D, NewK, TB, NewIT, S), _) :
    IT > TB,
    NewIT is IT - 1,
    NewK is K + 1.
transition(state(q_loop_K, CV, D, K, TB, IT, S), _, state(q_sub_chunk, CV, D, K, TB, IT, S), _) :-
    IT =< TB.

% In q_sub_chunk, subtract a strategic chunk or a large place-value chunk.
transition(state(q_sub_chunk, CV, D, K, _, _, S), Base, state(q_check_status, NewCV, NewD, 0, 0, 0,
    Remaining is CV - S,
    (K > 0, K =< Remaining ->
        Chunk = K,
        format(string(Interp), 'Subtract strategic chunk (-~w) to reach base.', [Chunk])
    ;
        (Remaining > 0 ->
            Power is floor(log(Remaining) / log(Base)),
            PowerValue is Base^Power,
            C is floor(Remaining / PowerValue) * PowerValue,
            (C > 0 -> Chunk = C ; Chunk = Remaining),
            format(string(Interp), 'Subtract large/remaining chunk (-~w).', [Chunk])
        )
    ),
    NewCV is CV - Chunk,
    NewD is D + Chunk.

% find_target_base_back/5 is a helper to find the next "friendly" number (counting down).
find_target_base_back(CV, S, Base, Power, TargetBase) :-
    BasePower is Base^Power,
    (CV mod BasePower =\= 0 ->
        TargetBase is floor(CV / BasePower) * BasePower
    ;
        (BasePower > CV ->
            TargetBase = CV
        ;
            NewPower is Power + 1,
            find_target_base_back(CV, S, Base, NewPower, TargetBase)
        )
    ).
```

34 sar_sub_cobo_missing_addend.pl

```prolog
/** <module> Student Subtraction Strategy: Counting On By Bases and Ones (Missing Addend)
 *
 * This module implements the 'Counting On by Bases and then Ones' (COBO)
 * strategy for subtraction, framed as a "missing addend" problem. It is
 * modeled as a finite state machine. It solves `M - S` by figuring out
 * what number needs to be added to `S` to reach `M`.
 *
 * The process is as follows:
 * 1. Start at the subtrahend (S). The goal is to reach the minuend (M).
 * 2. Count up from S by adding bases (tens) as many times as possible without
 *    exceeding M. The amount added is tracked as `Distance`.
 * 3. Once adding another base would overshoot M, switch to counting up by ones.
 * 4. Continue counting up by ones until M is reached.
 * 5. The total `Distance` accumulated is the result of the subtraction.
 * 6. The strategy fails if S > M.
 *
 * The state of the automaton is represented by the term:
 * `state(Name, CurrentValue, Distance, Target)`
 *
 * The history of execution is captured as a list of steps:
 * `step(Name, CurrentValue, Distance, Interpretation)`
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(sar_sub_cobo_missing_addend,
          [ run_cobo_ma/4
          ]).

:- use_module(library(lists)).

%!      run_cobo_ma(+M:integer, +S:integer, -FinalResult:integer, -History:list) is det.
%
%       Executes the 'Counting On by Bases and Ones' (Missing Addend) subtraction
%       strategy for M - S.
%
%       This predicate initializes and runs a state machine that models the
%       COBO "missing addend" strategy. It first checks if the subtraction is
%       possible (M >= S). If so, it finds the difference by counting up from
%       S to M, first by tens and then by ones. The total amount counted up
%       is the result. It traces the entire execution.
%
%       @param M The Minuend, the target number to count up to.
%       @param S The Subtrahend, the number to start counting from.
%       @param FinalResult The resulting difference (M - S). If S > M, this
%       will be the atom `'error'`.
%       @param History A list of `step/4` terms that describe the state
%       machine's execution path and the interpretation of each step.

run_cobo_ma(M, S, FinalResult, History) :-
    Base = 10,
    (S > M ->
        History = [step(q_error, 0, 0, 'Error: Subtrahend > Minuend.')],
        FinalResult = 'error'
    ;
        InitialState = state(q_init, S, 0, M),
        format(string(InitialInterpretation), 'Initialize at S (~w). Target is M (~w).', [S, M]),
```

```
        InitialHistoryEntry = step(q_start, 0, 0, InitialInterpretation),

        run(InitialState, Base, [InitialHistoryEntry], ReversedHistory),
        reverse(ReversedHistory, History),

        (last(History, step(q_accept, _, Dist, _)) -> FinalResult = Dist ; FinalResult = 'error')
    ).

% run/4 is the main recursive loop of the state machine.
run(state(q_accept, CV, Dist, _), _, Acc, FinalHistory) :-
    format(string(Interpretation), 'Target reached. Result (Distance) = ~w.', [Dist]),
    HistoryEntry = step(q_accept, CV, Dist, Interpretation),
    FinalHistory = [HistoryEntry | Acc].

run(CurrentState, Base, Acc, FinalHistory) :-
    transition(CurrentState, Base, NextState, Interpretation),
    CurrentState = state(Name, CV, Dist, _),
    HistoryEntry = step(Name, CV, Dist, Interpretation),
    run(NextState, Base, [HistoryEntry | Acc], FinalHistory).

% transition/4 defines the logic for moving from one state to the next.

% From q_init, proceed to add bases (tens).
transition(state(q_init, CV, Dist, T), _, state(q_add_bases, CV, Dist, T),
           'Proceed to add bases.').

% Loop in q_add_bases, counting on by one base (10) at a time, as long as it doesn't overshoot the t
transition(state(q_add_bases, CV, Dist, T), Base, state(q_add_bases, NewCV, NewDist, T), Interp) :-
    CV + Base =< T,
    NewCV is CV + Base,
    NewDist is Dist + Base,
    format(string(Interp), 'Count on by base (+~w). New Value=~w.', [Base, NewCV]).
% When adding the next base would overshoot, transition to adding ones.
transition(state(q_add_bases, CV, Dist, T), Base, state(q_add_ones, CV, Dist, T),
           'Next base overshoots target. Switching to ones.') :-
    CV + Base > T.

% Loop in q_add_ones, counting on by one at a time until the target is reached.
transition(state(q_add_ones, CV, Dist, T), _, state(q_add_ones, NewCV, NewDist, T), Interp) :-
    CV < T,
    NewCV is CV + 1,
    NewDist is Dist + 1,
    format(string(Interp), 'Count on by one (+1). New Value=~w.', [NewCV]).
% When the target is reached, transition to the final accept state.
transition(state(q_add_ones, T, Dist, T), _, state(q_accept, T, Dist, T),
           'Target reached.') :-
    true.
```

35 sar_sub_decomposition.pl

```
/** <module> Student Subtraction Strategy: Decomposition (Standard Algorithm)
 *
 * This module implements the standard "decomposition" or "borrowing"
 * algorithm for subtraction, modeled as a finite state machine.
 *
 * The process is as follows:
 * 1. Decompose both the minuend (M) and subtrahend (S) into tens and ones.
 * 2. Subtract the tens components.
 * 3. Check if the ones component of M is sufficient to subtract the ones
```

```
 *    component of S.
 * 4. If not, "borrow" or "decompose" a ten from M's tens component, adding
 *    it to M's ones component. This is the key step of the algorithm.
 * 5. Subtract the ones components.
 * 6. Recombine the resulting tens and ones to get the final answer.
 * 7. The strategy fails if S > M.
 *
 * The state is represented by the term:
 * `state(StateName, Result_Tens, Result_Ones, Subtrahend_Tens, Subtrahend_Ones)`
 *
 * The history of execution is captured as a list of steps:
 * `step(StateName, Result_Tens, Result_Ones, Interpretation)`
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(sar_sub_decomposition,
          [ run_decomposition/4
          ]).

:- use_module(library(lists)).

%!      run_decomposition(+M:integer, +S:integer, -FinalResult:integer, -History:list) is det.
%
%       Executes the 'Decomposition' (borrowing) subtraction strategy for M - S.
%
%       This predicate initializes and runs a state machine that models the
%       standard schoolbook subtraction algorithm. It first checks if the
%       subtraction is possible (M >= S). If so, it decomposes both numbers
%       and performs the subtraction column by column, handling borrowing
%       when necessary. It traces the entire execution.
%
%       @param M The Minuend, the number to subtract from.
%       @param S The Subtrahend, the number to subtract.
%       @param FinalResult The resulting difference (M - S). If S > M, this
%       will be the atom `'error'`.
%       @param History A list of `step/4` terms that describe the state
%       machine's execution path and the interpretation of each step.

run_decomposition(M, S, FinalResult, History) :-
    Base = 10,
    (S > M ->
        History = [step(q_error, 0, 0, 'Error: Subtrahend > Minuend.')],
        FinalResult = 'error'
    ;
        % Initial state: Decompose both M and S into tens and ones.
        S_T is S // Base, S_O is S mod Base,
        M_T is M // Base, M_O is M mod Base,

        InitialState = state(q_init, M_T, M_O, S_T, S_O),

        format(string(InitialInterpretation), 'Inputs: M=~w, S=~w. Decompose M (~wT+~wO) and S (~wT+
        InitialHistoryEntry = step(q_start, M_T, M_O, InitialInterpretation),

        run(InitialState, Base, [InitialHistoryEntry], ReversedHistory),
        reverse(ReversedHistory, History),

        (last(History, step(q_accept, RT, RO, _)) ->
            FinalResult is RT * Base + RO
```

```
        ;
            FinalResult = 'computation_error'
        )
    ).

% run/4 is the main recursive loop of the state machine.
run(state(q_accept, R_T, R_O, _, _), Base, AccHistory, FinalHistory) :-
    Result is R_T * Base + R_O,
    format(string(Interpretation), 'Accept. Final Result: ~w.', [Result]),
    HistoryEntry = step(q_accept, R_T, R_O, Interpretation),
    FinalHistory = [HistoryEntry | AccHistory].

run(CurrentState, Base, AccHistory, FinalHistory) :-
    transition(CurrentState, Base, NextState, Interpretation),
    CurrentState = state(Name, R_T, R_O, _, _),
    HistoryEntry = step(Name, R_T, R_O, Interpretation),
    run(NextState, Base, [HistoryEntry | AccHistory], FinalHistory).

% transition/4 defines the logic for moving from one state to the next.

% From q_init, proceed to subtract the tens column.
transition(state(q_init, R_T, R_O, S_T, S_O), _Base, state(q_sub_bases, R_T, R_O, S_T, S_O),
           'Proceed to subtract bases.').

% In q_sub_bases, subtract the tens and move to check the ones column.
transition(state(q_sub_bases, R_T, R_O, S_T, S_O), _Base, state(q_check_ones, New_R_T, R_O, S_T, S_O
    New_R_T is R_T - S_T,
    format(string(Interpretation), 'Subtract Bases: ~wT - ~wT = ~wT.', [R_T, S_T, New_R_T]).

% In q_check_ones, determine if borrowing is needed.
transition(state(q_check_ones, R_T, R_O, S_T, S_O), _Base, state(q_sub_ones, R_T, R_O, S_T, S_O), In
    R_O >= S_O,
    format(string(Interpretation), 'Sufficient Ones (~w >= ~w). Proceed.', [R_O, S_O]).

transition(state(q_check_ones, R_T, R_O, S_T, S_O), _Base, state(q_decompose, R_T, R_O, S_T, S_O), I
    R_O < S_O,
    format(string(Interpretation), 'Insufficient Ones (~w < ~w). Need decomposition.', [R_O, S_O]).

% In q_decompose, perform the "borrow" from the tens column.
transition(state(q_decompose, R_T, R_O, S_T, S_O), Base, state(q_sub_ones, New_R_T, New_R_O, S_T, S_
    R_T > 0,
    New_R_T is R_T - 1,
    New_R_O is R_O + Base,
    format(string(Interpretation), 'Decomposed 1 Ten. New state: ~wT, ~wO.', [New_R_T, New_R_O]).

% In q_sub_ones, subtract the ones column and transition to the final accept state.
transition(state(q_sub_ones, R_T, R_O, S_T, S_O), _Base, state(q_accept, R_T, New_R_O, S_T, S_O), In
    New_R_O is R_O - S_O,
    format(string(Interpretation), 'Subtract Ones: ~wO - ~wO = ~wO.', [R_O, S_O, New_R_O]).
```

36 sar_sub_rounding.pl

```
/** <module> Student Subtraction Strategy: Double Rounding
 *
 * This module implements a "double rounding" strategy for subtraction (M - S),
 * sometimes used by students to simplify the calculation. It is modeled as a
 * finite state machine.
 *
 * The process is as follows:
```

```
 * 1. Round both the minuend (M) and the subtrahend (S) down to the nearest
 *    multiple of 10. Let the rounded values be MR and SR, and the amounts
 *    they were rounded by be KM and KS respectively.
 * 2. Perform a simplified subtraction on the rounded numbers: `TR = MR - SR`.
 * 3. Adjust this temporary result. First, add back the amount M was rounded by: `TR + KM`.
 * 4. Second, subtract the amount S was rounded by: `(TR + KM) - KS`.
 *    This final adjustment is modeled as a chunking/counting-back process.
 * 5. The strategy fails if S > M.
 *
 * The state is represented by the term:
 * `state(Name, K_M, K_S, TempResult, K_S_Rem, Chunk, M, S, MR, SR)`
 *
 * The history of execution is captured as a list of steps:
 * `step(Name, K_M, K_S, TempResult, K_S_Rem, Interpretation)`
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(sar_sub_rounding,
          [ run_sub_rounding/4
          ]).

:- use_module(library(lists)).

%!      run_sub_rounding(+M:integer, +S:integer, -FinalResult:integer, -History:list) is det.
%
%       Executes the 'Double Rounding' subtraction strategy for M - S.
%
%       This predicate initializes and runs a state machine that models the
%       double rounding process. It first checks if the subtraction is possible
%       (M >= S). If so, it rounds both numbers down, subtracts them, and then
%       performs two adjustments to arrive at the final answer. It traces
%       the entire execution, providing a step-by-step history.
%
%       @param M The Minuend.
%       @param S The Subtrahend.
%       @param FinalResult The resulting difference (M - S). If S > M, this
%       will be the atom `'error'`.
%       @param History A list of `step/6` terms that describe the state
%       machine's execution path and the interpretation of each step.

run_sub_rounding(M, S, FinalResult, History) :-
    Base = 10,
    (S > M ->
        History = [step(q_error, 0, 0, 0, 0, 'Error: Subtrahend > Minuend.')],
        FinalResult = 'error'
    ;
        InitialState = state(q_start, 0, 0, 0, 0, 0, M, S, 0, 0),
        InitialHistoryEntry = step(q_start, 0, 0, 0, 0, 'Start.'),

        run(InitialState, Base, [InitialHistoryEntry], ReversedHistory),
        reverse(ReversedHistory, History),

        (last(History, step(q_accept, _, _, TR, _, _)) -> FinalResult = TR ; FinalResult = 'error')
    ).

% run/4 is the main recursive loop of the state machine.
run(state(q_accept, KM, KS, TR, 0, _, _, _, _, _), _, Acc, FinalHistory) :-
    format(string(Interpretation), 'Adjustment for S complete. Final Result = ~w.', [TR]),
```

```
    HistoryEntry = step(q_accept, KM, KS, TR, 0, Interpretation),
    FinalHistory = [HistoryEntry | Acc].

run(CurrentState, Base, Acc, FinalHistory) :-
    transition(CurrentState, Base, NextState, Interpretation),
    CurrentState = state(Name, KM, KS, TR, KSR, _, _, _, _, _),
    HistoryEntry = step(Name, KM, KS, TR, KSR, Interpretation),
    run(NextState, Base, [HistoryEntry | Acc], FinalHistory).

% transition/4 defines the logic for moving from one state to the next.

% Initial state, proceeds to rounding the Minuend.
transition(state(q_start, _, _, _, _, _, M, S, _, _), _, state(q_round_M, 0, 0, 0, 0, 0, M, S, 0, 0)

% Round M down and record the amount it was rounded by (KM).
transition(state(q_round_M, _, _, _, _, _, M, S, _, _), Base, state(q_round_S, KM, 0, 0, 0, 0, M, S,
    KM is M mod Base,
    MR is M - KM,
    format(string(Interp), 'Round M down: ~w -> ~w. (K_M = ~w).', [M, MR, KM]).

% Round S down and record the amount it was rounded by (KS).
transition(state(q_round_S, KM, _, _, _, _, M, S, MR, _), Base, state(q_subtract, KM, KS, 0, 0, 0, M
    KS is S mod Base,
    SR is S - KS,
    format(string(Interp), 'Round S down: ~w -> ~w. (K_S = ~w).', [S, SR, KS]).

% Perform the intermediate subtraction with the rounded numbers.
transition(state(q_subtract, KM, KS, _, _, _, M, S, MR, SR), _, state(q_adjust_M, KM, KS, TR, 0, 0,
    TR is MR - SR,
    format(string(Interp), 'Intermediate Subtraction: ~w - ~w = ~w.', [MR, SR, TR]).

% First adjustment: Add back the amount M was rounded by (KM).
transition(state(q_adjust_M, KM, KS, TR, _, _, M, S, MR, SR), _, state(q_init_adjust_S, KM, KS, NewT
    NewTR is TR + KM,
    format(string(Interp), 'Adjust for M (Add K_M): ~w + ~w = ~w.', [TR, KM, NewTR]).

% Prepare for the second adjustment: subtracting KS.
transition(state(q_init_adjust_S, KM, KS, TR, _, _, M, S, MR, SR), _, state(q_loop_adjust_S, KM, KS,
    format(string(Interp), 'Begin Adjust for S (Subtract K_S): Need to subtract ~w.', [KS]).

% Second adjustment is complete when the remainder (KSR) is zero.
transition(state(q_loop_adjust_S, KM, KS, TR, 0, _, M, S, MR, SR), _, state(q_accept, KM, KS, TR, 0,
% Perform the second adjustment by subtracting KS in chunks.
transition(state(q_loop_adjust_S, KM, KS, TR, KSR, _, M, S, MR, SR), Base, state(q_loop_adjust_S, KM
    KSR > 0,
    K_to_prev_base is TR mod Base,
    (K_to_prev_base > 0, KSR >= K_to_prev_base -> Chunk = K_to_prev_base ; Chunk = KSR),
    NewTR is TR - Chunk,
    NewKSR is KSR - Chunk,
    format(string(Interp), 'Chunking Adjustment: ~w - ~w = ~w.', [TR, Chunk, NewTR]).
```

37 sar_sub_sliding.pl

```
/** <module> Student Subtraction Strategy: Sliding (Constant Difference)
 *
 * This module implements the "sliding" or "constant difference" strategy for
 * subtraction (M - S), modeled as a finite state machine.
 *
 * The core idea of this strategy is that the difference between two numbers
```

```
 * remains the same if both numbers are shifted by the same amount. The
 * strategy simplifies the problem `M - S` by transforming it into
 * `(M + K) - (S + K)`, where `K` is chosen to make `S + K` a "friendly"
 * number (a multiple of 10).
 *
 * The process is as follows:
 * 1. Determine the amount `K` needed to "slide" the subtrahend (S) up to the
 *    next multiple of 10.
 * 2. Add `K` to both the minuend (M) and the subtrahend (S) to get the new
 *    numbers, `M_adj` and `S_adj`.
 * 3. Perform the simplified subtraction `M_adj - S_adj`.
 * 4. The strategy fails if S > M.
 *
 * The state is represented by the term:
 * `state(Name, K, M_adj, S_adj, TargetBase, TempCounter, M, S)`
 *
 * The history of execution is captured as a list of steps:
 * `step(Name, K, M_adj, S_adj, Interpretation)`
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(sar_sub_sliding,
          [ run_sliding/4
          ]).

:- use_module(library(lists)).

%!      run_sliding(+M:integer, +S:integer, -FinalResult:integer, -History:list) is det.
%
%       Executes the 'Sliding' (Constant Difference) subtraction strategy for M - S.
%
%       This predicate initializes and runs a state machine that models the
%       sliding strategy. It first checks if the subtraction is possible (M >= S).
%       If so, it calculates the amount `K` to slide both numbers, performs the
%       adjustment, and then executes the final, simpler subtraction. It
%       traces the entire execution.
%
%       @param M The Minuend.
%       @param S The Subtrahend.
%       @param FinalResult The resulting difference (M - S). If S > M, this
%       will be the atom `'error'`.
%       @param History A list of `step/5` terms that describe the state
%       machine's execution path and the interpretation of each step.

run_sliding(M, S, FinalResult, History) :-
    Base = 10,
    (S > M ->
        History = [step(q_error, 0, 0, 0, 'Error: Subtrahend > Minuend.')],
        FinalResult = 'error'
    ;
        (S > 0, S mod Base =\= 0 -> TB is ((S // Base) + 1) * Base ; TB is S),
        InitialState = state(q_init_K, 0, 0, 0, TB, S, M, S),
        InitialHistoryEntry = step(q_start, 0, 0, 0, 'Start.'),

        run(InitialState, Base, [InitialHistoryEntry], ReversedHistory),
        reverse(ReversedHistory, History),

        (last(History, step(q_accept, _, M_adj, S_adj, _)) -> FinalResult is M_adj - S_adj ; FinalRe
```

```
    ).

% run/4 is the main recursive loop of the state machine.
run(state(q_accept, K, M_adj, S_adj, _, _, _, _), _, Acc, FinalHistory) :-
    Result is M_adj - S_adj,
    format(string(Interpretation), 'Perform Subtraction: ~w - ~w = ~w.', [M_adj, S_adj, Result]),
    HistoryEntry = step(q_accept, K, M_adj, S_adj, Interpretation),
    FinalHistory = [HistoryEntry | Acc].

run(CurrentState, Base, Acc, FinalHistory) :-
    transition(CurrentState, Base, NextState, Interpretation),
    CurrentState = state(Name, K, M_adj, S_adj, _, _, _, _),
    HistoryEntry = step(Name, K, M_adj, S_adj, Interpretation),
    run(NextState, Base, [HistoryEntry | Acc], FinalHistory).

% transition/4 defines the logic for moving from one state to the next.

% From q_init_K, determine the amount K needed to slide S to a multiple of 10.
transition(state(q_init_K, _, _, _, TB, _, M, S), _, state(q_loop_K, 0, 0, 0, TB, S, M, S), Interp)
    format(string(Interp), 'Initializing K calculation: Counting from ~w to ~w.', [S, TB]).

% Loop in q_loop_K to count up from S to the target base, calculating K.
transition(state(q_loop_K, K, M_adj, S_adj, TB, TC, M, S), _, state(q_loop_K, NewK, M_adj, S_adj, TB
    TC < TB,
    NewTC is TC + 1,
    NewK is K + 1,
    format(string(Interp), 'Counting Up: ~w, K=~w', [NewTC, NewK]).
% Once K is found, transition to q_adjust to apply the slide.
transition(state(q_loop_K, K, _, _, TB, TC, M, S), _, state(q_adjust, K, 0, 0, TB, TC, M, S), Interp
    TC >= TB,
    format(string(Interp), 'K needed to reach base is ~w.', [K]).

% In q_adjust, "slide" both M and S by adding K.
transition(state(q_adjust, K, _, _, _, _, M, S), _, state(q_subtract, K, M_adj, S_adj, 0, 0, M, S),
    S_adj is S + K,
    M_adj is M + K,
    format(string(Interp), 'Sliding both by +~w. New problem: ~w - ~w.', [K, M_adj, S_adj]).

% In q_subtract, the new problem is set up. Proceed to accept to perform the final calculation.
transition(state(q_subtract, K, M_adj, S_adj, _, _, _, _), _, state(q_accept, K, M_adj, S_adj, 0, 0,
```

38 script.js

```
// --- Configuration ---
const API_BASE_URL = 'http://localhost:8083';

// --- Prolog API Backend ---
const PrologBackend = {
    // Brandom's Incompatibility Semantics
    async analyzeSemantics(statement) {
        try {
            const response = await fetch(`${API_BASE_URL}/analyze_semantics`, {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json',
                },
                body: JSON.stringify({ statement: statement })
            });
```

```
            if (!response.ok) {
                throw new Error(`HTTP error! status: ${response.status}`);
            }

            return await response.json();
        } catch (error) {
            console.error('Error analyzing semantics:', error);
            return {
                statement: statement,
                implies: ['Error: Could not connect to Prolog server'],
                incompatibleWith: [`Please ensure the Prolog server is running on port ${API_BASE_UR
            };
        }
    },

    // CGI and Piagetian Analysis
    async analyzeStrategy(problemContext, strategyDescription) {
        try {
            const response = await fetch(`${API_BASE_URL}/analyze_strategy`, {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json',
                },
                body: JSON.stringify({
                    problemContext: problemContext,
                    strategy: strategyDescription
                })
            });

            if (!response.ok) {
                throw new Error(`HTTP error! status: ${response.status}`);
            }

            return await response.json();
        } catch (error) {
            console.error('Error analyzing strategy:', error);
            return {
                classification: "Connection Error",
                stage: "Unknown",
                implications: `Could not connect to Prolog server. Please ensure the server is runni
                incompatibility: "",
                recommendations: `Check that the Prolog API server is started and accessible at ${AP
            };
        }
    }
};

// --- Frontend Logic ---

function openTab(evt, tabName) {
    var i, tabcontent, tablinks;

    tabcontent = document.getElementsByClassName("tab-content");
    for (i = 0; i < tabcontent.length; i++) {
        tabcontent[i].classList.remove("active");
    }

    tablinks = document.getElementsByClassName("tab-button");
    for (i = 0; i < tablinks.length; i++) {
```

```
        tablinks[i].classList.remove("active");
    }

    document.getElementById(tabName).classList.add("active");
    // Check if evt is defined (for the initial load)
    if (evt) {
        evt.currentTarget.classList.add("active");
    }
}

async function analyzeIncompatibility() {
    const input = document.getElementById('conceptInput').value;
    const resultDiv = document.getElementById('incompatibilityResult');

    if (!input.trim()) {
        resultDiv.innerHTML = "<i>Please enter a statement to analyze.</i>";
        return;
    }

    // Show loading state
    resultDiv.innerHTML = "<i>Analyzing...</i>";

    const results = await PrologBackend.analyzeSemantics(input);

    if (results) {
        let html = `<h3>Semantic Analysis for: "${results.statement}"</h3>`;

        html += `<h4>Entailments (What it implies):</h4><ul>`;
        results.implies.forEach(item => {
            html += `<li>${item}</li>`;
        });
        html += `</ul>`;

        html += `<h4>Incompatibilities (What it excludes):</h4><ul>`;
        results.incompatibleWith.forEach(item => {
            html += `<li>${item}</li>`;
        });
        html += `</ul>`;

        resultDiv.innerHTML = html;
    } else {
        resultDiv.innerHTML = "<i>Error occurred during analysis.</i>";
    }
}

async function analyzeCGI() {
    const problemContext = document.getElementById('problemContext').value;
    const strategyInput = document.getElementById('strategyInput').value;
    const resultDiv = document.getElementById('cgiResult');

    if (!strategyInput.trim()) {
        resultDiv.innerHTML = "<i>Please describe the student's strategy.</i>";
        return;
    }

    // Show loading state
    resultDiv.innerHTML = "<i>Analyzing strategy...</i>";

    const analysis = await PrologBackend.analyzeStrategy(problemContext, strategyInput);
```

```
    if (analysis) {
        let html = `<h3>Analysis Results</h3>`;
        html += `<p><strong>Context:</strong> ${problemContext}</p>`;

        if (analysis.classification !== "Unclassified" && analysis.classification !== "Connection Er
            html += `<p><strong>Strategy Classification (CGI):</strong> ${analysis.classification}</
            html += `<p><strong>Developmental Stage (Piaget):</strong> ${analysis.stage}</p>`;
        }

        html += `<h4>Conceptual Implications:</h4><p>${analysis.implications}</p>`;

        if (analysis.incompatibility) {
            html += `<h4>Semantic Conflict:</h4>`;
            html += `<div class="incompatibility-highlight">${analysis.incompatibility}</div>`;
        }

        if (analysis.recommendations) {
            html += `<h4>Pedagogical Recommendations:</h4><p>${analysis.recommendations}</p>`;
        }

        resultDiv.innerHTML = html;

    } else {
        resultDiv.innerHTML = "<i>Error occurred during analysis.</i>";
    }
}

// Initialize the first tab on load
document.addEventListener('DOMContentLoaded', (event) => {
    //openTab(null, 'CGI');
});
```

39 simple_api_server.pl

```
/** <module> Simple, Self-Contained API Server
 *
 * This module provides a lightweight, self-contained HTTP server that offers
 * semantic and strategy analysis endpoints. Unlike `api_server.pl` or
 * `working_server.pl`, this file includes the analysis logic directly within it,
 * making it independent of other modules like `incompatibility_semantics.pl`.
 *
 * It is likely intended for testing, demonstration, or as a simplified
 * alternative to the more complex, modularized servers.
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- use_module(library(http/thread_httpd)).
:- use_module(library(http/http_dispatch)).
:- use_module(library(http/http_json)).
:- use_module(library(http/json_convert)).
:- use_module(library(http/http_cors)).

% Define the REST API endpoints
:- http_handler(root(analyze_semantics), analyze_semantics_handler, [method(post)]).
:- http_handler(root(analyze_strategy), analyze_strategy_handler, [method(post)]).

% Enable CORS for all endpoints
```

```
:- set_setting(http:cors, [*]).

%!      server(+Port:integer) is det.
%
%       Starts the HTTP server on the specified Port.
%
%       @param Port The port number for the server to listen on.
server(Port) :-
    http_server(http_dispatch, [port(Port)]).

% --- Endpoint Handlers ---

%!      analyze_semantics_handler(+Request:list) is det.
%
%       Handles POST requests to the `/analyze_semantics` endpoint.
%       It expects a JSON object with a `statement` key, e.g., `{"statement": "The object is red"}`.
%       It performs a semantic analysis of the statement using its internal helper predicates.
%
%       @param Request The incoming HTTP request.
analyze_semantics_handler(Request) :-
    cors_enable(Request, [methods([post, options])]),
    (   http_read_json_dict(Request, In) ->
        Statement = In.statement,
        analyze_statement_semantics(Statement, Analysis),
        reply_json_dict(Analysis)
    ;   reply_json_dict(_{error: "Invalid JSON input"})
    ).

%!      analyze_strategy_handler(+Request:list) is det.
%
%       Handles POST requests to the `/analyze_strategy` endpoint.
%       It expects a JSON object with `problemContext` and `strategy` keys,
%       e.g., `{"problemContext": "Math-JRU", "strategy": "student counted all"}`.
%       It returns a CGI/Piagetian analysis of the described student strategy.
%
%       @param Request The incoming HTTP request.
analyze_strategy_handler(Request) :-
    cors_enable(Request, [methods([post, options])]),
    (   http_read_json_dict(Request, In) ->
        ProblemContext = In.problemContext,
        StrategyDescription = In.strategy,
        analyze_cgi_strategy(ProblemContext, StrategyDescription, Analysis),
        reply_json_dict(Analysis)
    ;   reply_json_dict(_{error: "Invalid JSON input"})
    ).

% --- Helper Predicates for Analysis ---

% analyze_statement_semantics(+Statement, -Analysis)
% Analyzes a statement using incompatibility semantics
analyze_statement_semantics(Statement, Analysis) :-
    atom_string(StatementAtom, Statement),
    downcase_atom(StatementAtom, Normalized),

    findall(Implication, get_implications(Normalized, Implication), Implies),
    findall(Incompatibility, get_incompatibilities(Normalized, Incompatibility), IncompatibleWith),

    Analysis = _{
        statement: Statement,
```

```
        implies: Implies,
        incompatibleWith: IncompatibleWith
    }.

% get_implications(+NormalizedStatement, -Implication)
% Determines what a statement implies
get_implications(Statement, 'The object is colored') :-
    sub_atom(Statement, _, _, _, red).
get_implications(Statement, 'The shape is a rectangle') :-
    sub_atom(Statement, _, _, _, square).
get_implications(Statement, 'The shape is a polygon') :-
    sub_atom(Statement, _, _, _, square).
get_implications(Statement, 'The shape has 4 sides of equal length') :-
    sub_atom(Statement, _, _, _, square).
get_implications(Statement, 'This statement has semantic content') :-
    Statement \= ''.

% get_incompatibilities(+NormalizedStatement, -Incompatibility)
% Determines what a statement is incompatible with
get_incompatibilities(Statement, 'The object is entirely blue') :-
    sub_atom(Statement, _, _, _, red).
get_incompatibilities(Statement, 'The object is monochromatic and green') :-
    sub_atom(Statement, _, _, _, red).
get_incompatibilities(Statement, 'The shape is a circle') :-
    sub_atom(Statement, _, _, _, square).
get_incompatibilities(Statement, 'The shape has exactly 3 sides') :-
    sub_atom(Statement, _, _, _, square).
get_incompatibilities(Statement, 'The negation of this statement') :-
    Statement \= ''.

% analyze_cgi_strategy(+ProblemContext, +StrategyDescription, -Analysis)
% Analyzes a student strategy using CGI and Piagetian frameworks
analyze_cgi_strategy(ProblemContext, StrategyDescription, Analysis) :-
    atom_string(StrategyAtom, StrategyDescription),
    downcase_atom(StrategyAtom, Normalized),

    classify_strategy(ProblemContext, Normalized, Classification, Stage, Implications, Incompatibili

    Analysis = _{
        classification: Classification,
        stage: Stage,
        implications: Implications,
        incompatibility: Incompatibility,
        recommendations: Recommendations
    }.

% classify_strategy(+Context, +NormalizedStrategy, -Classification, -Stage, -Implications, -Incompat
classify_strategy(Context, Strategy, Classification, Stage, Implications, Incompatibility, Recommend
    atom_string(Context, ContextStr),
    sub_atom(ContextStr, 0, 4, _, "Math"),
    (   (sub_atom(Strategy, _, _, _, 'count all') ;
         sub_atom(Strategy, _, _, _, 'starting from one') ;
         sub_atom(Strategy, _, _, _, '1, 2, 3')) ->
        Classification = "Direct Modeling: Counting All",
        Stage = "Preoperational (Piaget)",
        Implications = "The student needs to represent the quantities concretely and cannot treat th
        Incompatibility = "A commitment to 'Counting All' is incompatible with the concept of 'Cardi
        Recommendations = "Encourage 'Counting On'. Ask: 'You know there are 5 here. Can you start c
    ;   (sub_atom(Strategy, _, _, _, 'count on') ;
```

```
         sub_atom(Strategy, _, _, _, 'started at 5')) ->
        Classification = "Counting Strategy: Counting On",
        Stage = "Concrete Operational (Early)",
        Implications = "The student understands the cardinality of the first number. This is a signi
        Incompatibility = "Reliance on 'Counting On' is incompatible with the immediate retrieval re
        Recommendations = "Work on derived facts. Ask: 'If you know 5 + 5 = 10, how can that help yo
    ;   (sub_atom(Strategy, _, _, _, 'known fact') ;
         sub_atom(Strategy, _, _, _, 'just knew')) ->
        Classification = "Known Fact / Fluency",
        Stage = "Concrete Operational",
        Implications = "The student has internalized the number relationship.",
        Incompatibility = "",
        Recommendations = "Introduce more complex problem structures (e.g., Join Change Unknown or m
    ;
        Classification = "Unclassified",
        Stage = "Unknown",
        Implications = "Could not clearly identify the strategy based on the description. Please pro
        Incompatibility = "",
        Recommendations = ""
    ).

classify_strategy("Science-Float", Strategy, Classification, Stage, Implications, Incompatibility, R
    (   (sub_atom(Strategy, _, _, _, heavy) ; sub_atom(Strategy, _, _, _, big)) ->
        Classification = "Perceptual Reasoning: Weight/Size as defining factor",
        Stage = "Preoperational",
        Implications = "The student is focusing on salient perceptual features (size, weight) rather
        Incompatibility = "The concept that 'heavy things sink' is incompatible with observations of
        Recommendations = "Introduce an incompatible observation (disequilibrium). Show a very large
    ;
        Classification = "Unclassified",
        Stage = "Unknown",
        Implications = "Could not clearly identify the strategy based on the description. Please pro
        Incompatibility = "",
        Recommendations = ""
    ).

% Default case for unmatched contexts
classify_strategy(_, _, "Unclassified", "Unknown", "Could not clearly identify the strategy based on

% To run the server from the command line:
% swipl -g "server(8080)" simple_api_server.pl
:- initialization(server(8080), main).
```

40 smr_div_cbo.pl

```
/** <module> Student Division Strategy: Conversion to Groups Other than Bases (CBO)
 *
 * This module implements a sophisticated division strategy, sometimes called
 * "Conversion to Groups Other than Bases," modeled as a finite state machine.
 * It solves a division problem (T / S) by leveraging knowledge of a counting
 * base (e.g., 10).
 *
 * The process is as follows:
 * 1.  Decompose the total (T) into a number of bases (TB) and ones (TO).
 * 2.  Analyze the base itself: determine how many groups of size S can be
 *     made from one base, and what the remainder is. (e.g., "how many 4s in 10?").
 * 3.  Use this knowledge to quickly calculate the quotient and remainder that
 *     result from the "bases" part of the total (TB).
 * 4.  Combine the remainder from the bases with the original "ones" part (TO).
```

```
 * 5.  Process this combined final remainder to see how many more groups of
 *     size S can be made.
 * 6.  Sum the quotients from the base and remainder parts to get the final answer.
 * 7.  The strategy fails if the divisor (S) is not positive.
 *
 * The state is represented by the term:
 * `state(Name, T_Bases, T_Ones, Quotient, Remainder, S_in_Base, Rem_in_Base, Total, Divisor)`
 *
 * The history of execution is captured as a list of steps:
 * `step(Name, Quotient, Remainder, Interpretation)`
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(smr_div_cbo,
          [ run_cbo_div/5
          ]).

:- use_module(library(lists)).

%!      run_cbo_div(+T:integer, +S:integer, +Base:integer, -FinalQuotient:integer, -FinalRemainder:i
%
%       Executes the 'Conversion to Groups Other than Bases' division strategy
%       for T / S, using the specified Base.
%
%       This predicate initializes and runs a state machine that models the CBO
%       division strategy. It first checks for a positive divisor. If valid, it
%       decomposes the dividend `T` and uses knowledge about the `Base` to find
%       the quotient and remainder. It traces the entire execution.
%
%       @param T The Dividend (Total).
%       @param S The Divisor (Size of groups).
%       @param Base The numerical base to use for decomposition (e.g., 10).
%       @param FinalQuotient The quotient of the division.
%       @param FinalRemainder The remainder of the division. If S is not
%       positive, this will be the atom `'error'`.

run_cbo_div(T, S, Base, FinalQuotient, FinalRemainder) :-
    (S =< 0 ->
        % History is not exposed, but we could create it here if needed.
        % History = [step(q_error, 0, 0, 'Error: Divisor must be positive.')],
        FinalQuotient = 'error', FinalRemainder = 'error'
    ;
        TB is T // Base,
        TO is T mod Base,
        InitialState = state(q_init, TB, TO, 0, 0, 0, 0, T, S),

        run(InitialState, Base, [], ReversedHistory),
        reverse(ReversedHistory, _History), % History is generated but not returned.

        (last(ReversedHistory, step(q_accept, FinalQuotient, FinalRemainder, _)) -> true ;
         (FinalQuotient = 'error', FinalRemainder = 'error'))
    ).

% run/4 is the main recursive loop of the state machine.
run(state(q_accept, _, _, Q, R, _, _, _, _), _, Acc, FinalHistory) :-
    format(string(Interpretation), 'Finished. Total Quotient = ~w.', [Q]),
    HistoryEntry = step(q_accept, Q, R, Interpretation),
    FinalHistory = [HistoryEntry | Acc].
```

```
run(CurrentState, Base, Acc, FinalHistory) :-
    transition(CurrentState, Base, NextState, Interpretation),
    CurrentState = state(Name, _, _, Q, R, _, _, _, _),
    HistoryEntry = step(Name, Q, R, Interpretation),
    run(NextState, Base, [HistoryEntry | Acc], FinalHistory).

% transition/4 defines the logic for moving from one state to the next.

% From q_init, decompose T and proceed to analyze the base.
transition(state(q_init, TB, TO, Q, R, SiB, RiB, T, S), _, state(q_analyze_base, TB, TO, Q, R, SiB,
    format(string(Interp), 'Initialize: ~w/~w. Decompose T: ~w Bases + ~w Ones.', [T, S, TB, TO]).

% In q_analyze_base, determine how many groups of S fit in one Base.
transition(state(q_analyze_base, TB, TO, Q, R, _, _, T, S), Base, state(q_process_bases, TB, TO, Q,
    SiB is Base // S,
    RiB is Base mod S,
    format(string(Interp), 'Analyze Base: One Base (~w) = ~w group(s) of ~w + Remainder ~w.', [Base,

% In q_process_bases, calculate the quotient and remainder from the "bases" part of T.
transition(state(q_process_bases, TB, TO, _, _, SiB, RiB, T, S), _, state(q_combine_R, TB, TO, NewQ,
    NewQ is TB * SiB,
    NewR is TB * RiB,
    format(string(Interp), 'Process ~w Bases: Yields ~w groups and ~w remainder.', [TB, NewQ, NewR])

% In q_combine_R, add the remainder from the bases to the original ones part of T.
transition(state(q_combine_R, _, TO, Q, R, SiB, RiB, T, S), _, state(q_process_R, _, TO, Q, NewR, Si
    NewR is R + TO,
    format(string(Interp), 'Combine Remainders: ~w (from Bases) + ~w (from Ones) = ~w.', [R, TO, New

% In q_process_R, find the quotient and remainder from the combined remainder, then accept.
transition(state(q_process_R, _, _, Q, R, _, _, T, S), _, state(q_accept, _, _, NewQ, NewR, _, _, T,
    Q_from_R is R // S,
    NewR is R mod S,
    NewQ is Q + Q_from_R,
    format(string(Interp), 'Process Remainder: Yields ~w additional group(s).', [Q_from_R]).
```

41 smr_div_dealing_by_ones.pl

```
/** <module> Student Division Strategy: Dealing by Ones
 *
 * This module implements a basic "dealing" or "sharing one by one" strategy
 * for division (T / N), modeled as a finite state machine. It simulates
 * distributing a total number of items (T) one at a time into a number of
 * groups (N) until the items run out.
 *
 * The process is as follows:
 * 1. Initialize N empty groups.
 * 2. Deal one item from the total T to the first group.
 * 3. Deal one item to the second group, and so on, cycling through the groups.
 * 4. Continue until all T items have been dealt.
 * 5. The quotient is the number of items in any one group (assuming fair sharing,
 *    i.e., the remainder is 0). This model does not explicitly calculate a remainder.
 * 6. The strategy fails if the number of groups (N) is not positive.
 *
 * The state is represented by the term:
 * `state(Name, RemainingItems, Groups, CurrentGroupIndex)`
 *
 * The history of execution is captured as a list of steps:
```

```
 * `step(Name, RemainingItems, Groups, Interpretation)`
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(smr_div_dealing_by_ones,
          [ run_dealing_by_ones/4
          ]).

:- use_module(library(lists)).

%!      run_dealing_by_ones(+T:integer, +N:integer, -FinalQuotient:integer, -History:list) is det.
%
%       Executes the 'Dealing by Ones' division strategy for T / N.
%
%       This predicate initializes and runs a state machine that models the
%       process of dealing `T` items one by one into `N` groups. It first
%       checks for a positive number of groups `N`. If valid, it simulates
%       the dealing process and traces the execution. The quotient is the
%       final number of items in one of the groups.
%
%       @param T The Dividend (Total number of items to deal).
%       @param N The Divisor (Number of groups to deal into).
%       @param FinalQuotient The result of the division (items per group).
%       If N is not positive, this will be the atom `'error'`.
%       @param History A list of `step/4` terms that describe the state
%       machine's execution path and the interpretation of each step.

run_dealing_by_ones(T, N, FinalQuotient, History) :-
    (N =< 0, T > 0 ->
        History = [step(q_error, T, [], 'Error: Cannot divide by N.')],
        FinalQuotient = 'error'
    ;
        % Create a list of N zeros to represent the groups.
        length(Groups, N),
        maplist(=(0), Groups),
        InitialState = state(q_init, T, Groups, 0),

        run(InitialState, N, [], ReversedHistory),
        reverse(ReversedHistory, History),

        (last(History, step(q_accept, _, FinalGroups, _)), nth0(0, FinalGroups, FinalQuotient) -> tr
    ).

% run/4 is the main recursive loop of the state machine.
run(state(q_accept, 0, Groups, _), _, Acc, FinalHistory) :-
    (nth0(0, Groups, R) -> Result = R ; Result = 0),
    format(string(Interpretation), 'Dealing complete. Result: ~w per group.', [Result]),
    HistoryEntry = step(q_accept, 0, Groups, Interpretation),
    FinalHistory = [HistoryEntry | Acc].

run(CurrentState, N, Acc, FinalHistory) :-
    transition(CurrentState, N, NextState, Interpretation),
    CurrentState = state(Name, Rem, Gs, _),
    HistoryEntry = step(Name, Rem, Gs, Interpretation),
    run(NextState, N, [HistoryEntry | Acc], FinalHistory).

% transition/4 defines the logic for moving from one state to the next.
```

```prolog
% From q_init, proceed to the main dealing loop.
transition(state(q_init, T, Gs, Idx), _, state(q_loop_deal, T, Gs, Idx), Interp) :-
    length(Gs, N),
    format(string(Interp), 'Initialize: ~w items to deal into ~w groups.', [T, N]).

% In q_loop_deal, deal one item to the current group and cycle to the next.
transition(state(q_loop_deal, Rem, Gs, Idx), N, state(q_loop_deal, NewRem, NewGs, NewIdx), Interp) :
    Rem > 0,
    NewRem is Rem - 1,
    % Increment value in the list at the current group index.
    nth0(Idx, Gs, OldVal, Rest),
    NewVal is OldVal + 1,
    nth0(Idx, NewGs, NewVal, Rest),
    NewIdx is (Idx + 1) mod N,
    format(string(Interp), 'Dealt 1 item to Group ~w.', [Idx+1]).
% If no items remain, transition to the accept state.
transition(state(q_loop_deal, 0, Gs, Idx), _, state(q_accept, 0, Gs, Idx), 'Dealing complete.').
```

42 smr_div_idp.pl

```prolog
/** <module> Student Division Strategy: Inverse of Distributive Property (IDP)
 *
 * This module implements a division strategy based on the inverse of the
 * distributive property, modeled as a finite state machine. It solves a
 * division problem (T / S) by using a knowledge base (KB) of known
 * multiplication facts for the divisor S.
 *
 * The process is as follows:
 * 1.  Given a knowledge base of facts for S (e.g., 2*S, 5*S, 10*S), find the
 *     largest known multiple of S that is less than or equal to the
 *     remaining total (T).
 * 2.  Subtract this multiple from T.
 * 3.  Add the corresponding factor to a running total for the quotient.
 * 4.  Repeat the process with the new, smaller remainder until no more known
 *     multiples can be subtracted.
 * 5.  The final quotient is the sum of the factors, and the final remainder
 *     is what's left of the total.
 * 6.  The strategy fails if the divisor (S) is not positive.
 *
 * The state is represented by the term:
 * `state(Name, Remaining, TotalQuotient, PartialTotal, PartialQuotient, KB, Divisor)`
 *
 * The history of execution is captured as a list of steps:
 * `step(Name, Remainder, TotalQuotient, PartialTotal, PartialQuotient, Interpretation)`
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(smr_div_idp,
          [ run_idp/5
          ]).

:- use_module(library(lists)).

%!      run_idp(+T:integer, +S:integer, +KB_in:list, -FinalQuotient:integer, -FinalRemainder:integer
%
%       Executes the 'Inverse of Distributive Property' division strategy for T / S.
%
%       This predicate initializes and runs a state machine that models the IDP
```

```prolog
%       strategy. It first checks for a positive divisor. If valid, it uses the
%       provided knowledge base `KB_in` to repeatedly subtract the largest
%       possible known multiple of `S` from `T`, accumulating the quotient.
%       It traces the entire execution.
%
%       @param T The Dividend (Total).
%       @param S The Divisor.
%       @param KB_in A list of `Multiple-Factor` pairs representing known
%       multiplication facts for `S`. Example: `[20-2, 50-5, 100-10]` for S=10.
%       @param FinalQuotient The calculated quotient of the division.
%       @param FinalRemainder The calculated remainder. If S is not positive,
%       this will be `T`.

run_idp(T, S, KB_in, FinalQuotient, FinalRemainder) :-
    (S =< 0 ->
        % History is not exposed, but we could create it here if needed.
        % History = [step(q_error, T, 0, 0, 0, 'Error: Divisor must be positive.')],
        FinalQuotient = 'error', FinalRemainder = T
    ;
        % Sort KB descending by the multiple (the key) for the greedy search.
        keysort(KB_in, SortedKB_asc),
        reverse(SortedKB_asc, KB),

        InitialState = state(q_init, T, 0, 0, 0, KB, S),

        run(InitialState, [], ReversedHistory),
        reverse(ReversedHistory, _History), % History is generated but not returned.

        (last(ReversedHistory, step(q_accept, FinalRemainder, FinalQuotient, _, _, _)) -> true ;
         (FinalQuotient = 'error', FinalRemainder = 'error'))
    ).

% run/3 is the main recursive loop of the state machine.
run(state(q_accept, Rem, TQ, _, _, _, _), Acc, FinalHistory) :-
    format(string(Interpretation), 'Decomposition complete. Total Quotient = ~w.', [TQ]),
    HistoryEntry = step(q_accept, Rem, TQ, 0, 0, Interpretation),
    FinalHistory = [HistoryEntry | Acc].

run(CurrentState, Acc, FinalHistory) :-
    transition(CurrentState, NextState, Interpretation),
    CurrentState = state(Name, Rem, TQ, PT, PQ, _, _),
    HistoryEntry = step(Name, Rem, TQ, PT, PQ, Interpretation),
    run(NextState, [HistoryEntry | Acc], FinalHistory).

% transition/3 defines the logic for moving from one state to the next.

% From q_init, proceed to search the knowledge base.
transition(state(q_init, T, TQ, PT, PQ, KB, S), state(q_search_KB, T, TQ, PT, PQ, KB, S), Interp) :-
    format(string(Interp), 'Initialize: ~w / ~w. Loaded known facts for ~w.', [T, S, S]).

% In q_search_KB, find the best known multiple to subtract.
transition(state(q_search_KB, Rem, TQ, _, _, KB, S), state(q_apply_fact, Rem, TQ, Multiple, Factor,
    find_best_fact(KB, Rem, Multiple, Factor),
    format(string(Interp), 'Found known multiple: ~w (~w x ~w).', [Multiple, Factor, S]).
% If no suitable fact is found, the process is complete.
transition(state(q_search_KB, Rem, TQ, _, _, KB, S), state(q_accept, Rem, TQ, 0, 0, KB, S), 'No suit
    \+ find_best_fact(KB, Rem, _, _).

% In q_apply_fact, subtract the found multiple and add the factor to the quotient.
```

```prolog
transition(state(q_apply_fact, Rem, TQ, PT, PQ, KB, S), state(q_search_KB, NewRem, NewTQ, 0, 0, KB,
    NewRem is Rem - PT,
    NewTQ is TQ + PQ,
    format(string(Interp), 'Applied fact. Subtracted ~w. Added ~w to Quotient.', [PT, PQ]).

% find_best_fact/4 is a helper to greedily find the largest applicable known fact.
% It assumes KB is sorted in descending order of multiples.
find_best_fact([Multiple-Factor | _], Rem, Multiple, Factor) :-
    Multiple =< Rem.
find_best_fact([_ | Rest], Rem, BestMultiple, BestFactor) :-
    find_best_fact(Rest, Rem, BestMultiple, BestFactor).
```

43 smr_div_ucr.pl

```prolog
/** <module> Student Division Strategy: Using Commutative Reasoning (Repeated Addition)
 *
 * This module implements a division strategy based on the concept of
 * commutative reasoning, modeled as a finite state machine. It solves a
 * partitive division problem (E items into G groups) by reframing it as a
 * missing factor multiplication problem: `? * G = E`.
 *
 * The process is as follows:
 * 1.  Start with an accumulated total of 0 and a quotient (items per group) of 0.
 * 2.  In each step, simulate adding one item to each of the `G` groups. This
 *     is equivalent to adding `G` to the accumulated total and `1` to the quotient.
 * 3.  Continue this process of repeated addition until the accumulated total
 *     equals the target number of items `E`.
 * 4.  The final quotient represents the number of items that were placed in
 *     each group, which is the answer to the division problem.
 * 5.  This strategy implicitly uses the commutative property by solving
 *     `E / G = ?` as `? * G = E`.
 *
 * The state is represented by the term:
 * `state(Name, Total_Accumulated, Quotient_PerGroup, E_Total, G_Groups)`
 *
 * The history of execution is captured as a list of steps:
 * `step(Name, Total_Accumulated, Quotient_PerGroup, Interpretation)`
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(smr_div_ucr,
          [ run_ucr/4
          ]).

:- use_module(library(lists)).

%!      run_ucr(+E:integer, +G:integer, -FinalQuotient:integer, -History:list) is det.
%
%       Executes the 'Using Commutative Reasoning' division strategy for E / G.
%
%       This predicate initializes and runs a state machine that models the
%       process of solving a division problem by finding the missing factor
%       through repeated addition. It traces the entire execution, providing
%       a step-by-step history of how the quotient is built up.
%
%       @param E The Dividend (Total number of items).
%       @param G The Divisor (Number of groups).
%       @param FinalQuotient The result of the division (items per group).
```

```
%       @param History A list of `step/4` terms that describe the state
%       machine's execution path and the interpretation of each step.

run_ucr(E, G, FinalQuotient, History) :-
    InitialState = state(q_start, 0, 0, E, G),

    run(InitialState, [], ReversedHistory),
    reverse(ReversedHistory, History),

    (last(History, step(q_accept, _, FinalQuotient, _)) -> true ; FinalQuotient = 'error').

% run/3 is the main recursive loop of the state machine.
run(state(q_accept, _, Q, _, _), Acc, FinalHistory) :-
    format(string(Interpretation), 'Total reached. Problem solved. Output Q=~w.', [Q]),
    HistoryEntry = step(q_accept, 0, Q, Interpretation),
    FinalHistory = [HistoryEntry | Acc].

run(CurrentState, Acc, FinalHistory) :-
    transition(CurrentState, NextState, Interpretation),
    CurrentState = state(Name, T, Q, _, _),
    HistoryEntry = step(Name, T, Q, Interpretation),
    run(NextState, [HistoryEntry | Acc], FinalHistory).

% transition/3 defines the logic for moving from one state to the next.

% From q_start, identify the problem parameters.
transition(state(q_start, T, Q, E, G), state(q_initialize, T, Q, E, G),
           'Identify total items and number of groups.').

% From q_initialize, begin the iterative process.
transition(state(q_initialize, T, Q, E, G), state(q_iterate, T, Q, E, G),
           'Initialize distribution total and count per group.').

% In q_iterate, perform one round of distribution (repeated addition).
transition(state(q_iterate, T, Q, E, G), state(q_check, NewT, NewQ, E, G), Interp) :-
    NewT is T + G,
    NewQ is Q + 1,
    format(string(Interp), 'Distribute round ~w. Total distributed: ~w.', [NewQ, NewT]).

% In q_check, compare the accumulated total to the target total.
transition(state(q_check, T, Q, E, G), state(q_iterate, T, Q, E, G), Interp) :-
    T < E,
    format(string(Interp), 'Check: T (~w) < E (~w); continue distributing.', [T, E]).
transition(state(q_check, E, Q, E, G), state(q_accept, E, Q, E, G), Interp) :-
    format(string(Interp), 'Check: T (~w) == E (~w); total reached.', [E, E]).
transition(state(q_check, T, _, E, G), state(q_error, T, 0, E, G), Interp) :-
    T > E,
    format(string(Interp), 'Error: Accumulated total (~w) exceeded E (~w).', [T, E]).
```

44 smr_mult_c2c.pl

```
/** <module> Student Multiplication Strategy: Coordinating Two Counts (C2C)
 *
 * This module implements a foundational multiplication strategy, "Coordinating
 * Two Counts" (C2C), modeled as a finite state machine. This strategy
 * represents a direct modeling approach where a student literally counts every
 * single item across all groups.
 *
 * The cognitive process involves two simultaneous counting acts:
```

```
 * 1.  Tracking the number of items counted within the current group.
 * 2.  Tracking which group is currently being counted.
 *
 * This is a direct simulation of `N * S` where the total is found by
 * counting `1` for each item, `S` times for each of the `N` groups.
 *
 * The state is represented by the term:
 * `state(Name, GroupsDone, ItemInGroup, Total, NumGroups, GroupSize)`
 *
 * The history of execution is captured as a list of steps:
 * `step(Name, GroupsDone, ItemInGroup, Total, Interpretation)`
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(smr_mult_c2c,
          [ run_c2c/4
          ]).

:- use_module(library(lists)).

%!      run_c2c(+N:integer, +S:integer, -FinalTotal:integer, -History:list) is det.
%
%       Executes the 'Coordinating Two Counts' multiplication strategy for N * S.
%
%       This predicate initializes and runs a state machine that models the
%       C2C strategy. It simulates a student counting every item, one by one,
%       across all `N` groups of size `S`. It traces the entire execution,
%       providing a step-by-step history of the two coordinated counts.
%
%       @param N The number of groups.
%       @param S The size of each group (number of items).
%       @param FinalTotal The resulting product of N * S.
%       @param History A list of `step/5` terms that describe the state
%       machine's execution path and the interpretation of each step.

run_c2c(N, S, FinalTotal, History) :-
    InitialState = state(q_init, 0, 0, 0, N, S),

    run(InitialState, [], ReversedHistory),
    reverse(ReversedHistory, History),

    (last(History, step(q_accept, _, _, FinalTotal, _)) -> true ; FinalTotal = 'error').

% run/3 is the main recursive loop of the state machine.
run(state(q_accept, _, _, T, _, _), Acc, FinalHistory) :-
    format(string(Interpretation), 'All groups counted. Result = ~w.', [T]),
    HistoryEntry = step(q_accept, 0, 0, T, Interpretation),
    FinalHistory = [HistoryEntry | Acc].

run(CurrentState, Acc, FinalHistory) :-
    transition(CurrentState, NextState, Interpretation),
    CurrentState = state(Name, G, I, T, _, _),
    HistoryEntry = step(Name, G, I, T, Interpretation),
    run(NextState, [HistoryEntry | Acc], FinalHistory).

% transition/3 defines the logic for moving from one state to the next.

% From q_init, proceed to check the group counter.
```

```
transition(state(q_init, G, I, T, N, S), state(q_check_G, G, I, T, N, S), Interp) :-
    format(string(Interp), 'Inputs: ~w groups of ~w. Initialize counters.', [N, S]).

% In q_check_G, decide whether to count another group or finish.
transition(state(q_check_G, G, I, T, N, S), state(q_count_items, G, I, T, N, S), Interp) :-
    G < N,
    G1 is G + 1,
    format(string(Interp), 'G < N. Starting Group ~w.', [G1]).
transition(state(q_check_G, N, _, T, N, S), state(q_accept, N, 0, T, N, S), 'G = N. All groups count

% In q_count_items, count one item and increment the total. Loop until the group is full.
transition(state(q_count_items, G, I, T, N, S), state(q_count_items, G, NewI, NewT, N, S), Interp) :
    I < S,
    NewI is I + 1,
    NewT is T + 1,
    G1 is G + 1,
    format(string(Interp), 'Count: ~w. (Item ~w in Group ~w).', [NewT, NewI, G1]).
% When the current group is fully counted, move to the next group.
transition(state(q_count_items, G, S, T, N, S), state(q_next_group, G, S, T, N, S), Interp) :-
    G1 is G + 1,
    format(string(Interp), 'Group ~w finished.', [G1]).

% In q_next_group, increment the group counter and reset the item counter, then loop back.
transition(state(q_next_group, G, _, T, N, S), state(q_check_G, NewG, 0, T, N, S), 'Increment G. Res
    NewG is G + 1.
```

45 smr_mult_cbo.pl

```
/** <module> Student Multiplication Strategy: Conversion to Bases and Ones (CBO)
 *
 * This module implements a multiplication strategy based on the physical act
 * of creating groups and then re-grouping (converting) them into a standard
 * base, like 10. It's modeled as a finite state machine.
 *
 * The process is as follows:
 * 1.  Start with `N` groups, each containing `S` items.
 * 2.  Systematically take items from one "source" group and redistribute them
 *     one-by-one into other "target" groups.
 * 3.  The goal of the redistribution is to fill the target groups until they
 *     contain `Base` items (e.g., 10).
 * 4.  This process continues until the source group is empty.
 * 5.  The final total is calculated by summing the items in all the rearranged
 *     groups. This demonstrates the principle of conservation of number, as the
 *     total remains `N * S` despite the redistribution.
 *
 * The state is represented by the term:
 * `state(Name, Groups, SourceIndex, TargetIndex)`
 *
 * The history of execution is captured as a list of steps:
 * `step(Name, Groups, Interpretation)`
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(smr_mult_cbo,
          [ run_cbo_mult/5
          ]).

:- use_module(library(lists)).
```

```prolog
%!      run_cbo_mult(+N:integer, +S:integer, +Base:integer, -FinalTotal:integer, -History:list) is d
%
%       Executes the 'Conversion to Bases and Ones' multiplication strategy
%       for N * S, using a target Base for re-grouping.
%
%       This predicate initializes and runs a state machine that models the
%       conceptual process of redistribution. It creates `N` groups of `S` items
%       and then shuffles items between them to form groups of size `Base`.
%       The final total demonstrates that the quantity is conserved.
%
%       @param N The number of initial groups.
%       @param S The size of each initial group.
%       @param Base The target size for the re-grouping.
%       @param FinalTotal The resulting product (N * S).
%       @param History A list of `step/3` terms that describe the state
%       machine's execution path and the interpretation of each step.

run_cbo_mult(N, S, Base, FinalTotal, History) :-
    (N > 0 -> length(Groups, N), maplist(=(S), Groups) ; Groups = []),
    (N > 0 -> SourceIdx is N - 1 ; SourceIdx = -1),
    InitialState = state(q_init, Groups, SourceIdx, 0),

    run(InitialState, Base, [], ReversedHistory),
    reverse(ReversedHistory, History),

    (last(History, step(q_accept, FinalGroups, _)),
     calculate_total(FinalGroups, Base, FinalTotal) -> true ; FinalTotal = 'error').

% run/4 is the main recursive loop of the state machine.
run(state(q_accept, Gs, _, _), Base, Acc, FinalHistory) :-
    calculate_total(Gs, Base, Total),
    format(string(Interpretation), 'Final Tally. Total = ~w.', [Total]),
    HistoryEntry = step(q_accept, Gs, Interpretation),
    FinalHistory = [HistoryEntry | Acc].

run(CurrentState, Base, Acc, FinalHistory) :-
    transition(CurrentState, Base, NextState, Interpretation),
    CurrentState = state(Name, Gs, _, _),
    HistoryEntry = step(Name, Gs, Interpretation),
    run(NextState, Base, [HistoryEntry | Acc], FinalHistory).

% transition/4 defines the logic for moving from one state to the next.

% From q_init, select a source group to begin redistribution.
transition(state(q_init, Gs, SourceIdx, TI), _, state(q_select_source, Gs, SourceIdx, TI), 'Initiali

% From q_select_source, confirm the source and begin the transfer process.
transition(state(q_select_source, Gs, SourceIdx, TI), _, state(q_init_transfer, Gs, SourceIdx, TI),
    (SourceIdx >= 0 ->
        SI1 is SourceIdx + 1,
        format(string(Interp), 'Selected Group ~w as the source.', [SI1])
    ;
        Interp = 'No groups to process.'
    ).

% From q_init_transfer, start the main redistribution loop.
transition(state(q_init_transfer, Gs, SI, _), _, state(q_loop_transfer, Gs, SI, 0),
           'Starting redistribution loop.').
```

```
% In q_loop_transfer, move one item from the source group to a target group.
transition(state(q_loop_transfer, Gs, SI, TI), Base, state(q_loop_transfer, NewGs, SI, NewTI), Inter
    % Conditions for transfer: source has items, target is not full.
    nth0(SI, Gs, SourceItems), SourceItems > 0,
    length(Gs, N), TI < N,
    (TI =\= SI ->
        nth0(TI, Gs, TargetItems), TargetItems < Base,
        % Perform transfer of one item.
        update_list(Gs, SI, SourceItems - 1, Gs_mid),
        update_list(Gs_mid, TI, TargetItems + 1, NewGs),
        % Check if target is now full, if so, advance target index.
        (TargetItems + 1 =:= Base -> NewTI is TI + 1 ; NewTI is TI),
        format(string(Interp), 'Transferred 1 from ~w to ~w.', [SI+1, TI+1])
    ;
        % Skip transferring to the source index itself.
        NewTI is TI + 1, NewGs = Gs, Interp = 'Skipping source index.'
    ).
% Exit the loop when the source is empty or all targets have been considered.
transition(state(q_loop_transfer, Gs, SI, TI), _, state(q_finalize, Gs, SI, TI), 'Redistribution com
    (nth0(SI, Gs, 0) ; length(Gs, N), TI >= N).

% From q_finalize, move to the accept state.
transition(state(q_finalize, Gs, SI, TI), _, state(q_accept, Gs, SI, TI), 'Finalizing.').

% update_list/4 is a helper to non-destructively update a list element at an index.
update_list(List, Index, NewVal, NewList) :-
    nth0(Index, List, _, Rest),
    nth0(Index, NewList, NewVal, Rest).

% calculate_total/3 is a helper to sum the elements of the final groups list.
% Note: The Base is not used, as this just verifies the total number of items.
calculate_total([], _, 0).
calculate_total([H|T], Base, Total) :-
    calculate_total(T, Base, RestTotal),
    Total is H + RestTotal.
```

46 smr_mult_commutative_reasoning.pl

```
/** <module> Student Multiplication Strategy: Commutative Reasoning (Repeated Addition)
 *
 * This module implements a multiplication strategy based on repeated addition,
 * modeled as a finite state machine. The name "Commutative Reasoning" implies
 * that a student understands that `A * B` is equivalent to `B * A` and can
 * choose the more efficient path. However, this model directly implements
 * `A * B` as adding `B` to itself `A` times.
 *
 * The process is as follows:
 * 1.  Start with a total of 0.
 * 2.  Repeatedly add the number of items (`B`) to the total.
 * 3.  Use a counter, initialized to the number of groups (`A`), to track
 *     how many times to perform the addition.
 * 4.  The process stops when the counter reaches zero. The accumulated total
 *     is the final product.
 *
 * The state is represented by the term:
 * `state(Name, Groups, Items, Total, Counter)`
 *
 * The history of execution is captured as a list of steps:
```

```prolog
 * `step(Name, Groups, Items, Total, Interpretation)`
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(smr_mult_commutative_reasoning,
          [ run_commutative_mult/4
          ]).

:- use_module(library(lists)).

%!      run_commutative_mult(+A:integer, +B:integer, -FinalTotal:integer, -History:list) is det.
%
%       Executes the 'Commutative Reasoning' (Repeated Addition) multiplication
%       strategy for A * B.
%
%       This predicate initializes and runs a state machine that models the
%       process of calculating `A * B` by adding `B` to an accumulator `A` times.
%       It traces the entire execution, providing a step-by-step history of
%       the repeated addition.
%
%       @param A The number of groups (effectively, the number of additions).
%       @param B The number of items in each group (the number being added).
%       @param FinalTotal The resulting product of A * B.
%       @param History A list of `step/5` terms that describe the state
%       machine's execution path and the interpretation of each step.

run_commutative_mult(A, B, FinalTotal, History) :-
    Groups = A,
    Items = B,
    InitialState = state(q_init_calc, Groups, Items, 0, Groups),
    InitialHistoryEntry = step(q_start, 0, 0, 0, 'Start'),

    run(InitialState, [InitialHistoryEntry], ReversedHistory),
    reverse(ReversedHistory, History),

    (last(History, step(q_accept, _, _, Total, _)) -> FinalTotal = Total ; FinalTotal = 'error').

% run/3 is the main recursive loop of the state machine.
run(state(q_accept, _, _, Total, _), Acc, FinalHistory) :-
    format(string(Interpretation), 'Calculation complete. Result = ~w.', [Total]),
    HistoryEntry = step(q_accept, 0, 0, Total, Interpretation),
    FinalHistory = [HistoryEntry | Acc].

run(CurrentState, Acc, FinalHistory) :-
    transition(CurrentState, NextState, Interpretation),
    CurrentState = state(Name, Gs, Items, Total, _),
    HistoryEntry = step(Name, Gs, Items, Total, Interpretation),
    run(NextState, [HistoryEntry | Acc], FinalHistory).

% transition/3 defines the logic for moving from one state to the next.

% From q_init_calc, start the iterative calculation loop.
transition(state(q_init_calc, Gs, Items, _, _), state(q_loop_calc, Gs, Items, 0, Gs),
           'Initializing iterative calculation.').

% In q_loop_calc, add the number of items to the total and decrement the counter.
transition(state(q_loop_calc, Gs, Items, Total, Counter), state(q_loop_calc, Gs, Items, NewTotal, Ne
    Counter > 0,
```

```
    NewTotal is Total + Items,
    NewCounter is Counter - 1,
    format(string(Interp), 'Iterate: Added ~w. Total = ~w.', [Items, NewTotal]).
% When the counter reaches zero, the calculation is complete.
transition(state(q_loop_calc, _, _, Total, 0), state(q_accept, 0, 0, Total, 0),
           'Calculation complete.').
```

47 smr_mult_dr.pl

```
/** <module> Student Multiplication Strategy: Distributive Reasoning (DR)
 *
 * This module implements a multiplication strategy based on the distributive
 * property of multiplication over addition, modeled as a finite state machine.
 * It solves `N * S` by breaking `S` into two easier parts (`S1` and `S2`).
 *
 * The process is as follows:
 * 1.  Split the group size `S` into two smaller, more manageable parts,
 *     `S1` and `S2`, using a simple heuristic. For example, 7 might be
 *     split into 2 + 5.
 * 2.  Calculate the first partial product, `P1 = N * S1`, using repeated addition.
 * 3.  Calculate the second partial product, `P2 = N * S2`, also using repeated addition.
 * 4.  Sum the two partial products to get the final answer: `Total = P1 + P2`.
 *     This demonstrates the distributive property: `N * (S1 + S2) = (N * S1) + (N * S2)`.
 *
 * The state is represented by the term:
 * `state(Name, S1, S2, P1, P2, Total, Counter, N_Groups, S_Size)`
 *
 * The history of execution is captured as a list of steps:
 * `step(Name, S1, S2, P1, P2, Total, Interpretation)`
 *
 * @author Tilo Wiedera
 * @license MIT
 */
:- module(smr_mult_dr,
          [ run_dr/4
          ]).

:- use_module(library(lists)).

%!      run_dr(+N:integer, +S:integer, -FinalTotal:integer, -History:list) is det.
%
%       Executes the 'Distributive Reasoning' multiplication strategy for N * S.
%
%       This predicate initializes and runs a state machine that models the DR
%       strategy. It heuristically splits the multiplier `S` into two parts,
%       calculates the partial product for each part via repeated addition, and
%       then sums the partial products. It traces the entire execution.
%
%       @param N The number of groups.
%       @param S The size of each group (this is the number that will be split).
%       @param FinalTotal The resulting product of N * S.
%       @param History A list of `step/7` terms that describe the state
%       machine's execution path and the interpretation of each step.

run_dr(N, S, FinalTotal, History) :-
    Base = 10,
    InitialState = state(q_init, 0, 0, 0, 0, 0, 0, N, S),

    run(InitialState, Base, [], ReversedHistory),
```

```prolog
    reverse(ReversedHistory, History),

    (last(History, step(q_accept, _, _, _, _, FinalTotal, _)) -> true ; FinalTotal = 'error').

% run/4 is the main recursive loop of the state machine.
run(state(q_accept, _, _, P1, P2, Total, _, _, _), _, Acc, FinalHistory) :-
    format(string(Interpretation), 'Summing partials: ~w + ~w = ~w.', [P1, P2, Total]),
    HistoryEntry = step(q_accept, 0, 0, P1, P2, Total, Interpretation),
    FinalHistory = [HistoryEntry | Acc].

run(CurrentState, Base, Acc, FinalHistory) :-
    transition(CurrentState, Base, NextState, Interpretation),
    CurrentState = state(Name, S1, S2, P1, P2, Total, _, _, _),
    HistoryEntry = step(Name, S1, S2, P1, P2, Total, Interpretation),
    run(NextState, Base, [HistoryEntry | Acc], FinalHistory).

% transition/4 defines the logic for moving from one state to the next.

% From q_init, proceed to split the group size S.
transition(state(q_init, _, _, _, _, _, _, N, S), _, state(q_split, 0, 0, 0, 0, 0, 0, N, S), Interp)
    format(string(Interp), 'Inputs: ~w x ~w.', [N, S]).

% In q_split, split S into two parts, S1 and S2, using a heuristic.
transition(state(q_split, _, _, P1, P2, T, C, N, S), Base, state(q_init_P1, S1, S2, P1, P2, T, C, N,
    heuristic_split(S, Base, S1, S2),
    (S2 > 0 -> format(string(Interp), 'Split S (~w) into ~w + ~w.', [S, S1, S2])
    ; format(string(Interp), 'S (~w) is easy. No split needed.', [S])).

% In q_init_P1, prepare to calculate the first partial product (N * S1).
transition(state(q_init_P1, S1, S2, _, P2, T, _, N, S), _, state(q_loop_P1, S1, S2, 0, P2, T, N, N,
    format(string(Interp), 'Initializing calculation of P1 (~w x ~w).', [N, S1]).

% In q_loop_P1, calculate P1 using repeated addition.
transition(state(q_loop_P1, S1, S2, P1, P2, T, C, N, S), _, state(q_loop_P1, S1, S2, NewP1, P2, T, N
    C > 0,
    NewP1 is P1 + S1,
    NewC is C - 1,
    format(string(Interp), 'Iterate P1: Added ~w. P1 = ~w.', [S1, NewP1]).
% After P1 is calculated, decide whether to calculate P2 or just sum.
transition(state(q_loop_P1, S1, 0, P1, _, _, 0, N, S), _, state(q_sum, S1, 0, P1, 0, 0, 0, N, S), In
    format(string(Interp), 'P1 complete. P1 = ~w.', [P1]).
transition(state(q_loop_P1, S1, S2, P1, _, _, 0, N, S), _, state(q_init_P2, S1, S2, P1, 0, 0, 0, N,
    S2 > 0,
    format(string(Interp), 'P1 complete. P1 = ~w.', [P1]).

% In q_init_P2, prepare to calculate the second partial product (N * S2).
transition(state(q_init_P2, S1, S2, P1, _, T, _, N, S), _, state(q_loop_P2, S1, S2, P1, 0, T, N, N,
    format(string(Interp), 'Initializing calculation of P2 (~w x ~w).', [N, S2]).

% In q_loop_P2, calculate P2 using repeated addition.
transition(state(q_loop_P2, S1, S2, P1, P2, T, C, N, S), _, state(q_loop_P2, S1, S2, P1, NewP2, T, N
    C > 0,
    NewP2 is P2 + S2,
    NewC is C - 1,
    format(string(Interp), 'Iterate P2: Added ~w. P2 = ~w.', [S2, NewP2]).
transition(state(q_loop_P2, S1, S2, P1, P2, _, 0, N, S), _, state(q_sum, S1, S2, P1, P2, 0, 0, N, S)
    format(string(Interp), 'P2 complete. P2 = ~w.', [P2]).

% In q_sum, add the partial products to get the final total.
```

```
transition(state(q_sum, _, _, P1, P2, _, _, N, S), _, state(q_accept, 0, 0, P1, P2, Total, 0, N, S),
    Total is P1 + P2.

% heuristic_split/4 is a helper to split a number S into two parts, S1 and S2.
% It uses a simple set of rules to find an "easy" part to split off.
heuristic_split(Value, Base, S1, S2) :-
    (Value > Base -> S1 = Base, S2 is Value - Base ;
    (Base mod 2 =:= 0, Value > Base / 2 -> S1 is Base / 2, S2 is Value - S1 ;
    (Value > 2 -> S1 = 2, S2 is Value - 2 ;
    (Value > 1 -> S1 = 1, S2 is Value - 1 ;
    S1 = Value, S2 = 0)))).
```

48 strategies.pl

```
/** <module> Standardized Strategy Loader
 *
 * This module serves as a centralized loader for all defined student
 * reasoning strategies. It imports all `sar_*.pl` (Student Addition/Subtraction
 * Reasoning) and `smr_*.pl` (Student Multiplication/Division Reasoning)
 * modules.
 *
 * By centralizing the loading process, we ensure that the full library of
 * strategies is available to the reorganization engine for analysis,
 * synthesis, and validation.
 *
 * @author Jules
 * @license MIT
 */

:- module(strategies, []).

% Addition and Subtraction Strategies
:- use_module(sar_add_chunking).
:- use_module(sar_add_cobo).
:- use_module(sar_add_rmb).
:- use_module(sar_add_rounding).
:- use_module(sar_sub_cbbo_take_away).
:- use_module(sar_sub_chunking_a).
:- use_module(sar_sub_chunking_b).
:- use_module(sar_sub_chunking_c).
:- use_module(sar_sub_cobo_missing_addend).
:- use_module(sar_sub_decomposition).
:- use_module(sar_sub_rounding).
:- use_module(sar_sub_sliding).

% Multiplication and Division Strategies
:- use_module(smr_div_cbo).
:- use_module(smr_div_dealing_by_ones).
:- use_module(smr_div_idp).
:- use_module(smr_div_ucr).
:- use_module(smr_mult_c2c).
:- use_module(smr_mult_cbo).
:- use_module(smr_mult_commutative_reasoning).
:- use_module(smr_mult_dr).
```

49 style.css

```
body {
```

```css
    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
    background-color: #f4f4f9;
    margin: 0;
    padding: 0;
    color: #333;
    line-height: 1.6;
}

header {
    background-color: #005f73;
    color: white;
    padding: 1rem 0;
    text-align: center;
}

header h1 {
    margin: 0;
    font-size: 2rem;
}

header p {
    margin: 0.5rem 0 0;
    font-size: 1rem;
    opacity: 0.9;
}

.container {
    max-width: 900px;
    margin: 30px auto;
    background-color: white;
    box-shadow: 0 4px 12px rgba(0,0,0,0.1);
    border-radius: 8px;
    overflow: hidden;
}

.tabs {
    display: flex;
    background-color: #e9f5f5;
}

.tab-button {
    flex: 1;
    padding: 15px;
    border: none;
    background-color: transparent;
    cursor: pointer;
    font-size: 16px;
    font-weight: bold;
    color: #005f73;
    transition: background-color 0.3s, color 0.3s;
}

.tab-button:hover {
    background-color: #cee8e8;
}

.tab-button.active {
    background-color: white;
    color: #2c3e50;
```

```
    border-bottom: 3px solid #0a9396;
}

.tab-content {
    display: none;
    padding: 25px;
}

.tab-content.active {
    display: block;
}

h2 {
    color: #2c3e50;
    border-bottom: 2px solid #ecf0f1;
    padding-bottom: 10px;
}

h3, h4 {
    color: #005f73;
}

.input-group {
    margin-bottom: 20px;
}

label {
    display: block;
    margin-bottom: 8px;
    font-weight: bold;
}

input[type="text"], select, textarea {
    width: 100%;
    padding: 12px;
    border: 1px solid #ccc;
    border-radius: 4px;
    box-sizing: border-box;
    font-size: 14px;
}

button {
    background-color: #0a9396;
    color: white;
    padding: 12px 20px;
    border: none;
    border-radius: 4px;
    cursor: pointer;
    font-size: 16px;
    transition: background-color 0.3s;
}

button:hover {
    background-color: #005f73;
}

.results {
    margin-top: 25px;
    padding: 20px;
```

```
    background-color: #f9f9f9;
    border-left: 5px solid #0a9396;
    min-height: 100px;
}

.incompatibility-highlight {
    background-color: #ffeedd;
    padding: 10px;
    border-radius: 4px;
    margin-top: 10px;
}
```

50 test_full_loop.pl

```
:- begin_tests(full_reorganization_loop).

:- use_module(execution_handler).
:- use_module(object_level).

% Helper to create a Peano number
int_to_peano(0, 0).
int_to_peano(I, s(P)) :-
    I > 0,
    I_prev is I - 1,
    int_to_peano(I_prev, P).

test(reorganization_on_add, [setup(retractall(object_level:add(_,_,_)))]) :-
    % Define an inefficient add rule for the test
    assertz((object_level:add(A, B, Sum) :-
        object_level:enumerate(A),
        object_level:enumerate(B),
        object_level:recursive_add(A, B, Sum))),

    % This goal is inefficient because 3 is smaller than 10.
    % The learner should discover the "Count On Bigger" (COB) strategy.
    int_to_peano(3, PeanoA),
    int_to_peano(10, PeanoB),
    Goal = add(PeanoA, PeanoB, _Result),

    % Set a low limit to ensure the initial attempt fails
    Limit = 15,

    % This should succeed after reorganization
    run_computation(Goal, Limit).

:- end_tests(full_reorganization_loop).
```

51 test_server.pl

```
/** <module> Basic Test HTTP Server
 *
 * This module provides a minimal HTTP server with a single endpoint (`/test`).
 * Its purpose is to serve as a basic test to confirm that the SWI-Prolog
 * HTTP libraries are working correctly and that a server can be started.
 *
 * It is not part of the main application logic but can be useful for
 * debugging or initial environment setup verification.
 *
```

```
 * @author Tilo Wiedera
 * @license MIT
 */
:- use_module(library(http/thread_httpd)).
:- use_module(library(http/http_dispatch)).
:- use_module(library(http/http_json)).

% Define a simple test endpoint
:- http_handler(root(test), test_handler, [method(get)]).

%!      server(+Port:integer) is det.
%
%       Starts the HTTP server on the specified Port.
%
%       @param Port The port number for the server to listen on.
server(Port) :-
    http_server(http_dispatch, [port(Port)]).

%!      test_handler(+Request:list) is det.
%
%       Handles GET requests to the `/test` endpoint.
%
%       It responds with a simple, fixed JSON object `_{message: "Hello from Prolog!"}`
%       to confirm that the server is running and able to handle requests.
%
%       @param _Request The incoming HTTP request (unused).
test_handler(_Request) :-
    reply_json_dict(_{message: "Hello from Prolog!"}).

% To run the server from the command line:
% swipl -g "server(8082)" test_server.pl
:- initialization(server(8082), main).
```

52 test_synthesis.pl

```
/** <module> Unit Tests for Incompatibility Semantics
 *
 * This module contains the unit tests for the `incompatibility_semantics`
 * module. It uses the `plunit` testing framework to verify the correctness
 * of the core logic across various domains.
 *
 * The tests are organized into sections:
 * 1.  **Core Logic**: Basic tests for identity, incoherence, and negation.
 * 2.  **Arithmetic**: Tests for commutativity and domain-specific constraints (e.g., subtraction in
 * 3.  **Embodied Modal Logic**: Tests for the EML state transition axioms.
 * 4.  **Quadrilateral Hierarchy**: Tests for geometric entailment and incompatibility.
 * 5.  **Number Theory**: Tests for Euclid's proof of the infinitude of primes.
 * 6.  **Fractions**: Tests for arithmetic and object collection over rational numbers.
 *
 * To run these tests, execute `run_tests(unified_synthesis).` from the
 * SWI-Prolog console after loading this file.
 *
 * @author Tilo Wiedera
 * @license MIT
 */
% Load the module under test. Explicitly qualify imports to avoid ambiguity in tests.
:- use_module(incompatibility_semantics, [
    proves/1, incoherent/1, set_domain/1, is_recollection/2, normalize/2
]).
```

```prolog
:- use_module(library(plunit)).

% Ensure operators are visible for the test definitions.
:- op(500, fx, neg).
:- op(500, fx, comp_nec).
:- op(500, fx, exp_nec).
:- op(500, fx, exp_poss).
:- op(500, fx, comp_poss).
:- op(1050, xfy, =>).
:- op(550, xfy, rdiv).

:- begin_tests(unified_synthesis).

% --- Tests for Part 1: Core Logic and Domains ---
test(identity_subjective) :- assertion(proves([s(p)] => [s(p)])).
test(incoherence_subjective) :- assertion(incoherent([s(p), s(neg(p))])).

test(negation_handling_subjective_lem) :-
    assertion(proves([] => [s(p), s(neg(p))])).

% --- Tests for Part 2: Arithmetic Coexistence and Fixes ---

test(arithmetic_commutativity_normative) :-
    assertion(proves([n(plus(2,3,5))] => [n(plus(3,2,5))])).

test(arithmetic_subtraction_limit_n, [setup(set_domain(n))]) :-
    % This tests that demanding a subtraction resulting in a negative number
    % is incoherent in the domain of natural numbers.
    assertion(incoherent([n(minus(3,5,_))])).

test(arithmetic_subtraction_limit_n_persistence, [setup(set_domain(n))]) :-
    assertion(incoherent([n(minus(3,5,_)), s(p)])).

test(arithmetic_subtraction_limit_z, [setup(set_domain(z))]) :-
    % The same subtraction is coherent in the domain of integers.
    \+ assertion(incoherent([n(minus(3,5,_))])).

% --- Tests for Part 3: Embodied Modal Logic (EML) - UPDATED ---
test(eml_dynamic_u_to_a) :- assertion(proves([s(u)] => [s(a)])).
test(eml_dynamic_full_cycle) :- assertion(proves([s(lg)] => [s(a)])).

% New Tests for Tension and Compressive Possibility
test(eml_tension_expansive_poss) :-
    % Commitment 3: Possibility of Release
    assertion(proves([s(a)] => [s(exp_poss lg)])).

test(eml_tension_compressive_poss) :-
    % Commitment 3: Possibility of Fixation (Temptation)
    assertion(proves([s(a)] => [s(comp_poss t)])).

test(eml_tension_conjunction) :-
    % Verify that both possibilities are entailed by Awareness (using conjunction reduction)
    assertion(proves([s(a)] => [s(conj(exp_poss lg, comp_poss t))])).

test(eml_fixation_consequence) :-
    % Commitment 4a: Fixation necessarily leads to a contraction that collapses unity.
    assertion(proves([s(t)] => [s(neg(u))])).

test(hegel_loop_prevention) :-
```

```
    assertion(\+(proves([s(t_b)] => [s(x)]))).

% --- Tests for New Feature: Quadrilateral Hierarchy (Chapter 2) ---

test(quad_incompatibility_square_r1) :-
    assertion(incoherent([n(square(x)), n(r1(x))])).

test(quad_compatibility_trapezoid_r1) :-
    assertion(\+(incoherent([n(trapezoid(x)), n(r1(x))]))).

test(quad_incompatibility_persistence) :-
    assertion(incoherent([n(square(x)), n(r1(x)), s(other)])).

test(quad_entailment_square_rectangle) :-
    assertion(proves([n(square(x))] => [n(rectangle(x))])).

test(quad_entailment_rectangle_square_fail) :-
    assertion(\+(proves([n(rectangle(x))] => [n(square(x))]))).

test(quad_entailment_rhombus_kite) :-
    assertion(proves([n(rhombus(x))] => [n(kite(x))])).

test(quad_entailment_transitive) :-
    assertion(proves([n(square(x))] => [n(parallelogram(x))])).

test(quad_projection_contrapositive) :-
    assertion(proves([n(neg(rectangle(x)))] => [n(neg(square(x)))])).

test(quad_projection_inversion_fail) :-
    assertion(\+(proves([n(neg(square(x)))] => [n(neg(rectangle(x)))]))).

% --- Tests for Number Theory (Euclid's Proof) ---

% Test Grounding Helpers
test(euclid_grounding_prime) :-
    assertion(proves([] => [n(prime(7))])),
    assertion(\+ proves([] => [n(prime(6))])).

test(euclid_grounding_composite) :-
    assertion(proves([] => [n(composite(6))])),
    assertion(\+ proves([] => [n(composite(7))])).

% Test Material Inferences (M4 and M5)
test(euclid_material_inference_m5) :-
    % L=[2,3], Product(L)+1 = 7. P=7.
    assertion(proves([n(prime(7)), n(divides(7, 7))] => [n(neg(member(7, [2, 3])))])).

test(euclid_material_inference_m4) :-
    assertion(proves([n(prime(5)), n(neg(member(5, [2, 3])))] => [n(neg(is_complete([2, 3])))] )).

% Test Forward Chaining (Combining M5 and M4)
test(euclid_forward_chaining) :-
    % L=[2,3], N=7, P=7.
    Premises = [n(prime(7)), n(divides(7, 7)), n(is_complete([2, 3]))],
    Conclusion = [n(neg(is_complete([2, 3])))],
    assertion(proves(Premises => Conclusion)).

% Test Case 1 (N is Prime)
test(euclid_case_1_incoherence) :-
```

```
    % L=[2,3], N=7.
    assertion(incoherent([n(prime(7)), n(is_complete([2, 3]))])).

% Test Case 2 (N is Composite)
test(euclid_case_2_incoherence) :-
    % L=[2,3,5,7,11,13]. N=30031 (Composite: 59*509).
    L = [2,3,5,7,11,13],
    N = 30031,
    Premises = [n(composite(N)), n(is_complete(L))],
    assertion(incoherent(Premises)).

% Test The Final Theorem (Euclid's Theorem)
test(euclid_theorem_infinitude_of_primes) :-
    L = [2, 5, 11],
    assertion(incoherent([n(is_complete(L))])).

test(euclid_theorem_empty_list) :-
    assertion(incoherent([n(is_complete([]))])).

% --- Tests for Fractions (Jason.pl integration) ---

test(fraction_is_recollection, [setup(set_domain(q))]) :-
    assertion(is_recollection(1 rdiv 2, _)),
    assertion(is_recollection(5, _)),
    assertion(\+ is_recollection(1 rdiv 0, _)).

test(integer_is_recollection, [setup(set_domain(n))]) :-
    % is_recollection is domain-independent; it checks constructive possibility.
    % A fraction can be a valid recollection even if its use is restricted by domain norms.
    assertion(is_recollection(1 rdiv 2, _)),
    assertion(is_recollection(5, _)).

test(fraction_normalization) :-
    assertion(normalize(4 rdiv 8, 1 rdiv 2)),
    assertion(normalize(10 rdiv 2, 5)).

test(fraction_addition_grounding, [setup(set_domain(q))]) :-
    % 1/2 + 1/3 = 5/6
    assertion(proves([] => [o(plus(1 rdiv 2, 1 rdiv 3, 5 rdiv 6))])).

test(fraction_addition_mixed, [setup(set_domain(q))]) :-
    % 2 + 1/4 = 9/4
    assertion(proves([] => [o(plus(2, 1 rdiv 4, 9 rdiv 4))])).

test(fraction_subtraction_grounding, [setup(set_domain(q))]) :-
    % 1/2 - 1/3 = 1/6
    assertion(proves([] => [o(minus(1 rdiv 2, 1 rdiv 3, 1 rdiv 6))])).

% Test subtraction constraints in N with fractions
test(fraction_subtraction_limit_n, [setup(set_domain(n))]) :-
    % 1/3 - 1/2 = -1/6. Incoherent in N.
    assertion(incoherent([n(minus(1 rdiv 3, 1 rdiv 2, _))])).

test(fraction_iteration_grounding, [setup(set_domain(q))]) :-
    % (1/3) * 4 = 4/3
    assertion(proves([] => [o(iterate(1 rdiv 3, 4, 4 rdiv 3))])).

test(fraction_partition_grounding, [setup(set_domain(q))]) :-
    % (4/3) / 4 = 1/3 (Normalized from 4/12)
```

```
    assertion(proves([] => [o(partition(4 rdiv 3, 4, 1 rdiv 3))])).

test(fraction_partition_integer, [setup(set_domain(q))]) :-
    % 5 / 2 = 5/2
    assertion(proves([] => [o(partition(5, 2, 5 rdiv 2))])).

:- end_tests(unified_synthesis).
```

53 working_server.pl

```
/** <module> Minimal working Prolog API server
 *
 * This server provides the semantic analysis and CGI strategy analysis endpoints
 * without depending on complex modules that may have loading issues.
 * It is the main entry point for the web application.
 *
 * @author Tilo Wiedera
 * @license MIT
 */

:- use_module(library(http/thread_httpd)).
:- use_module(library(http/http_dispatch)).
:- use_module(library(http/http_json)).

% Define API endpoints
:- http_handler(root(analyze_semantics), analyze_semantics_handler, [method(post)]).
:- http_handler(root(analyze_strategy), analyze_strategy_handler, [method(post)]).
:- http_handler(root(test), test_handler, [method(get)]).

%!      start_server(+Port:integer) is det.
%
%       Starts the Prolog HTTP server on the specified Port.
%       It registers the API handlers and prints a startup message.
%
%       @param Port The port number to listen on.

start_server(Port) :-
    format('Starting Prolog API server on port ~w~n', [Port]),
    http_server(http_dispatch, [port(Port)]),
    format('Server started successfully at http://localhost:~w~n', [Port]),
    format('Test with: curl http://localhost:~w/test~n', [Port]).


%!      test_handler(+Request:list) is det.
%
%       Handles GET requests to the /test endpoint.
%       Responds with a simple JSON object to confirm the server is running.
%
%       @param _Request The incoming HTTP request (unused).

test_handler(_Request) :-
    format('Content-type: application/json~n~n'),
    format('{"status": "ok", "message": "Prolog server is running"}~n').


%!      analyze_semantics_handler(+Request:list) is det.
%
%       Handles POST requests to the /analyze_semantics endpoint.
%       It reads a JSON object with a "statement" key, analyzes it using
```

```prolog
%       incompatibility semantics, and returns the analysis as a JSON object.
%
%       @param Request The incoming HTTP request.
%       @error reply_json_dict(_{error: "Invalid JSON input"}) if the request body is not valid JSON

analyze_semantics_handler(Request) :-
    % Add CORS headers
    format('Access-Control-Allow-Origin: *~n'),
    format('Access-Control-Allow-Methods: POST, OPTIONS~n'),
    format('Access-Control-Allow-Headers: Content-Type~n'),

    (   http_read_json_dict(Request, In) ->
        Statement = In.statement,
        analyze_statement_semantics(Statement, Analysis),
        reply_json_dict(Analysis)
    ;   reply_json_dict(_{error: "Invalid JSON input"})
    ).


%!      analyze_strategy_handler(+Request:list) is det.
%
%       Handles POST requests to the /analyze_strategy endpoint.
%       It reads a JSON object with "problemContext" and "strategy" keys,
%       analyzes the student's strategy, and returns the analysis as a JSON object.
%
%       @param Request The incoming HTTP request.
%       @error reply_json_dict(_{error: "Invalid JSON input"}) if the request body is not valid JSON

analyze_strategy_handler(Request) :-
    % Add CORS headers
    format('Access-Control-Allow-Origin: *~n'),
    format('Access-Control-Allow-Methods: POST, OPTIONS~n'),
    format('Access-Control-Allow-Headers: Content-Type~n'),

    (   http_read_json_dict(Request, In) ->
        ProblemContext = In.problemContext,
        StrategyDescription = In.strategy,
        analyze_cgi_strategy(ProblemContext, StrategyDescription, Analysis),
        reply_json_dict(Analysis)
    ;   reply_json_dict(_{error: "Invalid JSON input"})
    ).


%!      analyze_statement_semantics(+Statement:string, -Analysis:dict) is det.
%
%       Performs semantic analysis on a given statement.
%       It finds all implications and incompatibilities for the normalized
%       (lowercase) statement.
%
%       @param Statement The input string to analyze.
%       @param Analysis A dict containing the original statement, a list of
%       implications, and a list of incompatibilities.

analyze_statement_semantics(Statement, Analysis) :-
    atom_string(StatementAtom, Statement),
    downcase_atom(StatementAtom, Normalized),

    findall(Implication, get_implications(Normalized, Implication), Implies),
    findall(Incompatibility, get_incompatibilities(Normalized, Incompatibility), IncompatibleWith),
```

```
    Analysis = _{
        statement: Statement,
        implies: Implies,
        incompatibleWith: IncompatibleWith
    }.


%!      get_implications(+Statement:atom, -Implication:string) is nondet.
%
%       Generates implications for a given statement.
%       This predicate defines the semantic entailments based on keywords
%       found in the statement. It is a multi-clause predicate where each
%       clause represents a different implication rule.
%
%       @param Statement The normalized (lowercase) input atom.
%       @param Implication A string describing what the statement implies.

get_implications(Statement, 'The object is colored') :-
    sub_atom(Statement, _, _, _, red).
get_implications(Statement, 'The shape is a rectangle') :-
    sub_atom(Statement, _, _, _, square).
get_implications(Statement, 'The shape is a polygon') :-
    sub_atom(Statement, _, _, _, square).
get_implications(Statement, 'The shape has 4 sides of equal length') :-
    sub_atom(Statement, _, _, _, square).
get_implications(Statement, 'This statement has semantic content') :-
    Statement \= ''.


%!      get_incompatibilities(+Statement:atom, -Incompatibility:string) is nondet.
%
%       Generates incompatibilities for a given statement.
%       This predicate defines what a statement semantically rules out based
%       on keywords. It is a multi-clause predicate where each clause
%       represents a different incompatibility rule.
%
%       @param Statement The normalized (lowercase) input atom.
%       @param Incompatibility A string describing what the statement is incompatible with.

get_incompatibilities(Statement, 'The object is entirely blue') :-
    sub_atom(Statement, _, _, _, red).
get_incompatibilities(Statement, 'The object is monochromatic and green') :-
    sub_atom(Statement, _, _, _, red).
get_incompatibilities(Statement, 'The shape is a circle') :-
    sub_atom(Statement, _, _, _, square).
get_incompatibilities(Statement, 'The shape has exactly 3 sides') :-
    sub_atom(Statement, _, _, _, square).
get_incompatibilities(Statement, 'The negation of this statement') :-
    Statement \= ''.


%!      analyze_cgi_strategy(+ProblemContext:string, +StrategyDescription:string, -Analysis:dict) is
%
%       Analyzes a student's problem-solving strategy within a given context.
%       It normalizes the strategy description and uses `classify_strategy/7`
%       to get a detailed analysis.
%
%       @param ProblemContext The context of the problem (e.g., "Math-Addition").
```

```
%       @param StrategyDescription A text description of the student's strategy.
%       @param Analysis A dict containing the classification, developmental stage,
%       implications, incompatibilities, and pedagogical recommendations.

analyze_cgi_strategy(ProblemContext, StrategyDescription, Analysis) :-
    atom_string(StrategyAtom, StrategyDescription),
    downcase_atom(StrategyAtom, Normalized),

    classify_strategy(ProblemContext, Normalized, Classification, Stage, Implications, Incompatibili

    Analysis = _{
        classification: Classification,
        stage: Stage,
        implications: Implications,
        incompatibility: Incompatibility,
        recommendations: Recommendations
    }.


%!      classify_strategy(+Context:string, +Strategy:atom, -Classification:string, -Stage:string, -I
%
%       Classifies a student's strategy for a math problem.
%       This predicate uses keyword matching on the strategy description to
%       determine the CGI classification (e.g., "Direct Modeling", "Counting On"),
%       the Piagetian stage, and associated pedagogical insights. This is the
%       primary clause for handling math-related strategies.
%
%       @param Context The problem context (must contain "Math").
%       @param Strategy The normalized student strategy description.
%       @param Classification The CGI classification of the strategy.
%       @param Stage The associated Piagetian developmental stage.
%       @param Implications What the strategy implies about the student's understanding.
%       @param Incompatibility The conceptual conflict this strategy might lead to.
%       @param Recommendations Pedagogical suggestions to advance the student's understanding.

classify_strategy(Context, Strategy, Classification, Stage, Implications, Incompatibility, Recommend
    atom_string(Context, ContextStr),
    sub_string(ContextStr, 0, 4, _, "Math"),
    !,
    (   (sub_atom(Strategy, _, _, _, 'count all') ;
         sub_atom(Strategy, _, _, _, 'starting from one') ;
         sub_atom(Strategy, _, _, _, '1, 2, 3')) ->
        Classification = "Direct Modeling: Counting All",
        Stage = "Preoperational (Piaget)",
        Implications = "The student needs to represent the quantities concretely and cannot treat th
        Incompatibility = "A commitment to 'Counting All' is incompatible with the concept of 'Cardi
        Recommendations = "Encourage 'Counting On'. Ask: 'You know there are 5 here. Can you start c
    ;   (sub_atom(Strategy, _, _, _, 'count on') ;
         sub_atom(Strategy, _, _, _, 'started at 5')) ->
        Classification = "Counting Strategy: Counting On",
        Stage = "Concrete Operational (Early)",
        Implications = "The student understands the cardinality of the first number. This is a signi
        Incompatibility = "Reliance on 'Counting On' is incompatible with the immediate retrieval re
        Recommendations = "Work on derived facts. Ask: 'If you know 5 + 5 = 10, how can that help yo
    ;   (sub_atom(Strategy, _, _, _, 'known fact') ;
         sub_atom(Strategy, _, _, _, 'just knew')) ->
        Classification = "Known Fact / Fluency",
        Stage = "Concrete Operational",
        Implications = "The student has internalized the number relationship.",
```

```
        Incompatibility = "",
        Recommendations = "Introduce more complex problem structures (e.g., Join Change Unknown or m
    ;
        Classification = "Unclassified",
        Stage = "Unknown",
        Implications = "Could not clearly identify the strategy based on the description. Please pro
        Incompatibility = "",
        Recommendations = ""
    ).


%!      classify_strategy(+Context:string, +Strategy:atom, -Classification:string, -Stage:string, -I
%
%       Classifies a student's strategy for a science (floating) problem.
%       This clause handles strategies related to why objects float or sink.
%       It identifies common misconceptions (e.g., heavy things sink) and
%       provides recommendations for inducing cognitive conflict.
%
%       @param Context The problem context (must be "Science-Float").
%       @param Strategy The normalized student strategy description.
%       @param Classification The classification of the student's reasoning.
%       @param Stage The associated Piagetian developmental stage.
%       @param Implications What the strategy implies about the student's understanding.
%       @param Incompatibility The conceptual conflict this strategy might lead to.
%       @param Recommendations Pedagogical suggestions to advance the student's understanding.

classify_strategy("Science-Float", Strategy, Classification, Stage, Implications, Incompatibility, R
    !,
    (   (sub_atom(Strategy, _, _, _, heavy) ; sub_atom(Strategy, _, _, _, big)) ->
        Classification = "Perceptual Reasoning: Weight/Size as defining factor",
        Stage = "Preoperational",
        Implications = "The student is focusing on salient perceptual features (size, weight) rather
        Incompatibility = "The concept that 'heavy things sink' is incompatible with observations of
        Recommendations = "Introduce an incompatible observation (disequilibrium). Show a very large
    ;
        Classification = "Unclassified",
        Stage = "Unknown",
        Implications = "Could not clearly identify the strategy based on the description. Please pro
        Incompatibility = "",
        Recommendations = ""
    ).

%!      classify_strategy(?, ?, -Classification, -Stage, -Implications, -Incompatibility, -Recommend
%
%       Default catch-all for `classify_strategy/7`.
%       This clause is used when the context does not match any of the more
%       specific `classify_strategy` predicates. It returns a generic
%       "Unclassified" result.
%
%       @param _Context Unused context argument.
%       @param _Strategy Unused strategy argument.
%       @param Classification Set to "Unclassified".
%       @param Stage Set to "Unknown".
%       @param Implications A message indicating the strategy could not be identified.
%       @param Incompatibility Set to an empty string.
%       @param Recommendations Set to an empty string.

classify_strategy(_, _, "Unclassified", "Unknown", "Could not clearly identify the strategy based on
```

```prolog
%!      main is det.
%
%       The main entry point for the server.
%       It starts the server on port 8083 and then blocks, waiting for
%       messages, to keep the server process alive. This is the predicate
%       to run to launch the application.

main :-
    start_server(8083),
    % Block the main thread to keep the server alive.
    thread_get_message(_).
```