DreamLog: Neural-Symbolic Integration through Compression-Based Learning and Wake-Sleep Cycles

Anonymous Authors

Institution
Department
City, Country
email@example.com

Abstract—We present DreamLog, a neural-symbolic system that integrates logic programming with large language models (LLMs) through a biologically-inspired architecture featuring wake-sleep cycles, compression-based learning, and recursive knowledge generation. DreamLog addresses the fundamental challenge of undefined predicates in logic programming by automatically generating rules through recursive LLM invocation, enabling compositional reasoning where complex concepts are decomposed into simpler constituents. The system employs *knowledge-base-aware* retrieval-augmented generation (RAG) using weighted embeddings that combine query similarity with knowledge base context, enabling example selection that considers both the query structure and the current reasoning state. Multi-layered validation ensures correctness of both generated rules *and facts*, with LLM-as-judge verification extended to fact validation for common-sense consistency checking. A novel success-based learning mechanism tracks which examples lead to successful inference, progressively improving retrieval quality over time through experience. During "sleep" phases, knowledge is consolidated through compression operators grounded in algorithmic information theory. The key innovation is treating undefined predicates not as errors but as opportunities for compositional knowledge discovery, transforming the brittleness of traditional logic programming into a mechanism for exploratory learning. This approach effectively bridges symbolic reasoning and neural knowledge generation, offering a practical framework for building interpretable AI systems that learn and adapt over time.

Index Terms—neural-symbolic integration, logic programming, compositional reasoning, compression-based learning, retrieval-augmented generation, large language models

I. Introduction

The integration of symbolic reasoning with neural approaches remains one of the fundamental challenges in artificial intelligence. While logic programming provides precise, interpretable reasoning with strong guarantees, it struggles with incomplete knowledge and the brittleness of hand-coded rules. Conversely, neural networks excel at pattern recognition and can leverage vast amounts of unstructured data, but lack the interpretability and compositional reasoning capabilities of symbolic systems.

A particularly vexing problem in logic programming is the handling of undefined predicates—queries about facts or relations not present in the knowledge base. Traditional systems simply fail when encountering undefined terms, requiring manual intervention to add missing knowledge. This brittleness severely limits the practical applicability of logic programming in open-world scenarios where complete knowledge specification is infeasible.

We present DreamLog, a neural-symbolic system that addresses these challenges through four key innovations:

1) **Recursive Knowledge Generation**: When undefined predicates are encountered during query evaluation, DreamLog automatically generates rules through recursive LLM invocation. If generated rules reference undefined predicates in their bodies, additional LLM calls define those predicates, creating a compositional knowledge-building process that decomposes complex concepts into simpler constituents.

2) **KB-Aware RAG with Success-Based Learning**: The system employs knowledge-base-aware retrieval-augmented generation using weighted embeddings that combine query similarity (70%) with knowledge base context (30%). A success-based learning mechanism tracks which examples lead to successful inference, using logarithmically-dampened boosts to progressively improve retrieval quality while preventing dominance by any single example.

3) **Multi-Layered Validation**: Generated rules *and facts* undergo structural validation (syntax, variable safety), semantic validation (preventing circular rules while allowing undefined predicates), and optional LLM-as-judge verification. Fact verification checks consistency with common-sense knowledge. Error-tolerant parsing handles common LLM formatting errors, with correction-based retry for iterative refinement.

4) **Compression-Based Learning**: Inspired by memory consolidation in biological systems, DreamLog implements wake-sleep cycles where the "wake" phase involves active query processing and knowledge acquisition, while the "sleep" phase consolidates knowledge through compression operators grounded in algorithmic information theory.

Our main contributions are:

- A novel architecture for compositional knowledge discovery through recursive LLM invocation, transforming undefined predicates from errors into opportunities for learning
- KB-aware RAG using weighted embeddings that combine query and knowledge base context, enabling example retrieval that considers the current reasoning state
- Success-based learning that tracks example effectiveness and progressively improves retrieval through experience, with logarithmic dampening to prevent runaway dominance
- Multi-layered validation that ensures correctness of both rules *and facts*, with LLM-as-judge verification extended to fact validation for common-sense consistency
- Error-tolerant parsing infrastructure that supports smaller, local models through robust handling of formatting inconsistencies
- A theoretically grounded framework based on Solomonoff induction and Kolmogorov complexity for compression-based learning in logic programs
- A biologically-inspired wake-sleep cycle mechanism for knowledge consolidation and continuous learning
- An implementation demonstrating the feasibility of the approach across multiple LLM providers, with a curated library of 35 diverse rule examples

II. BACKGROUND AND RELATED WORK

A. Logic Programming Foundations

Logic programming, exemplified by languages like Prolog [1], provides a declarative paradigm for knowledge representation and reasoning. Programs consist of facts and rules expressed in first-order logic, with query evaluation performed through SLD resolution [2]. The key strength of logic programming lies in its formal semantics and the ability to perform complex reasoning through unification and backtracking.

However, traditional logic programming systems suffer from several limitations:

- **Closed-world assumption**: Undefined predicates are assumed false, leading to brittleness
- **Knowledge engineering bottleneck**: All knowledge must be manually encoded
- **Limited learning capabilities**: No mechanism for acquiring new knowledge from experience

B. Inductive Logic Programming

Inductive Logic Programming (ILP) [3] addresses some of these limitations by learning logic programs from examples. Systems like FOIL [4], Progol [5], and more recently, Metagol [6] can synthesize rules from positive and negative examples. However, ILP systems typically require structured training data and struggle with noise and ambiguity inherent in real-world scenarios.

Recent work on differentiable ILP [7] attempts to bridge neural and symbolic approaches by making the rule learning process differentiable, but these approaches often sacrifice the interpretability and exactness of pure logic programs.

C. Neural-Symbolic Integration

The integration of neural and symbolic AI has seen renewed interest with approaches like Neural Theorem Provers [8], Logic Tensor Networks [9], and DeepProbLog [10]. These systems typically embed logical structures in continuous spaces or use neural networks to guide symbolic reasoning.

Our approach differs fundamentally by maintaining a clear separation between the symbolic reasoning engine and neural knowledge generation, using LLMs as an external knowledge source rather than attempting to neuralize the reasoning process itself. This separation preserves the interpretability of symbolic reasoning while leveraging neural pattern recognition.

D. Retrieval-Augmented Generation

Recent advances in Retrieval-Augmented Generation (RAG) have shown that retrieving relevant examples or context significantly improves LLM performance on specialized tasks. DreamLog extends traditional RAG with two key innovations: (1) *knowledge-base-aware retrieval* that considers both query similarity and current KB state through weighted embedding combination, and (2) *success-based learning* that tracks example effectiveness over time. Unlike traditional RAG, which typically retrieves documents, DreamLog retrieves structural patterns that guide rule synthesis, with retrieval quality improving through experience.

E. Compositional Reasoning

Our recursive knowledge generation mechanism aligns with research on compositional generalization in neural-symbolic systems. Systems like SCAN [21] and COGS emphasize the importance of compositional reasoning for generalization. DreamLog achieves compositionality through recursive LLM invocation, where complex predicates are automatically decomposed into simpler constituents. This allows the system to handle novel combinations of concepts without explicit training on those combinations.

F. Program Synthesis and Compression

Program synthesis research [11] has long recognized the connection between compression and generalization. The Minimum Description Length (MDL) principle [12] formalizes the intuition that the best model is the one that provides the most compact description of the data.

In the context of logic programming, compression can take many forms:

- **Rule extraction**: Finding general rules that subsume multiple facts
- **Predicate invention**: Creating new predicates that simplify the overall program
- **Redundancy elimination**: Removing facts derivable from rules

The role of sleep in memory consolidation is well-established in neuroscience [13]. During sleep, the brain replays experiences, strengthens important connections, and transfers knowledge from short-term to long-term memory. This has inspired several computational models, including:

- **Complementary Learning Systems** [14]: Separate fast and slow learning systems that consolidate knowledge over time
- **Wake-Sleep Algorithm** [15]: Unsupervised learning in hierarchical models through alternating wake and sleep phases
- **Experience Replay** [16]: Replaying past experiences to improve learning in reinforcement learning agents

DreamLog draws inspiration from these biological and computational models, implementing wake-sleep cycles for knowledge consolidation in the context of logic programming.

III. THE DREAMLOG ARCHITECTURE

A. System Overview

> [Architecture diagram to be added]
> Shows: Logic Engine, LLM Layer, Wake-Sleep Controller, Compression Engine

Fig. 1. DreamLog System Architecture

DreamLog consists of four main components:

1) **Logic Programming Engine**: A Prolog-like reasoning system with S-expression syntax
2) **LLM Integration Layer**: Hooks for generating knowledge when undefined predicates are encountered
3) **Wake-Sleep Controller**: Manages cycles of active querying and knowledge consolidation
4) **Compression Engine**: Implements various compression operators for knowledge refinement

B. Core Logic Programming Engine

The foundation of DreamLog is a logic programming engine supporting:

- Facts: Ground terms like `(parent john mary)`
- Rules: Horn clauses like `(grandparent X Z) :- (parent X Y), (parent Y Z)`
- Queries: Goals with variables like `(grandparent john Z)`

Query evaluation uses SLD resolution with backtracking, maintaining a substitution environment for variable bindings. The key innovation is the integration of LLM hooks triggered when undefined predicates are encountered.

C. LLM Integration for Undefined Predicates

When the evaluator encounters an undefined predicate during query resolution, it triggers an LLM hook with a sophisticated prompt generation and validation pipeline:

This enhanced pipeline incorporates several critical innovations: (1) KB-aware RAG using weighted embeddings

---

Algorithm 1 Enhanced LLM Hook with Validation

1: **Input:** Query q, Knowledge base KB
2: **Output:** Generated facts/rules
3: **if** $predicate(q) \notin KB$ **then**
4:     $context \leftarrow extract\_context(KB, q)$
5:     $emb \leftarrow w_q \cdot embed(q) + w_{kb} \cdot embed(context)$ {KB-aware}
6:     $examples \leftarrow retrieve(normalize(emb), success\_boost)$ {RAG}
7:     $prompt \leftarrow format\_prompt(q, context, examples)$
8:     $response \leftarrow LLM(prompt)$
9:     $(facts, rules) \leftarrow parse\_response(response)$ {Error-tolerant}
10:     **for** each $fact \in facts$ **do**
11:         $valid \leftarrow validate\_structural(fact)$
12:         **if** valid AND use_llm_judge **then**
13:             $valid \leftarrow verify\_fact\_commonsense(fact, KB)$ {New}
14:         **end if**
15:         **if** valid **then**
16:             $KB \leftarrow KB \cup \{fact\}$
17:         **end if**
18:     **end for**
19:     **for** each $rule \in rules$ **do**
20:         $valid \leftarrow validate\_structural(rule)$
21:         **if** valid **then**
22:             $valid \leftarrow validate\_semantic(rule, KB)$
23:         **end if**
24:         **if** valid AND use_llm_judge **then**
25:             $valid \leftarrow verify\_rule\_with\_llm(rule, KB)$
26:         **end if**
27:         **if** valid **then**
28:             $KB \leftarrow KB \cup \{rule\}$
29:         **else**
30:             $rule \leftarrow retry\_with\_correction(q, KB, errors)$
31:         **end if**
32:     **end for**
33:     $record\_success(examples)$ {Success-based learning}
34: **end if**
35: **return** $(facts, rules)$

---

combining query ($w_q = 0.7$) and knowledge base context ($w_{kb} = 0.3$), (2) robust parsing that handles common LLM formatting errors, (3) multi-layered validation combining structural, semantic, and optional LLM-based verification for *both facts and rules*, (4) common-sense verification for generated facts (e.g., rejecting `male(mary)`), (5) correction-based retry when generation fails, and (6) success-based learning that tracks example effectiveness.

D. Recursive Knowledge Generation

A key architectural feature of DreamLog is its support for *recursive* or *compositional* knowledge generation. When the LLM generates a rule containing undefined predicates in the rule body, the evaluator triggers additional LLM calls to define

those predicates. This creates a chain of knowledge generation that enables compositional reasoning:

Example 1: Consider a query for `(ancestor john mary)`. If `ancestor/2` is undefined, the LLM might generate:

```
1  (rule (ancestor X Y) ((parent X Y)))
2  (rule (ancestor X Z) ((parent X Y) (ancestor Y Z)))
```

If `parent/2` is also undefined, the evaluator recursively invokes the LLM hook to define it, continuing until all predicates are either defined or grounded in facts.

This recursive mechanism provides several advantages:

- **Compositional Reasoning**: Complex predicates are decomposed into simpler constituents
- **Knowledge Discovery**: The system can explore chains of reasoning without pre-specifying all intermediate concepts
- **Natural Abstraction**: Higher-level predicates are defined in terms of lower-level ones, mirroring human conceptual hierarchies
- **Incremental Learning**: The knowledge base grows organically through use

Importantly, our validation system deliberately does *not* require all body predicates to be defined. Undefined predicates are flagged for recursive generation rather than rejected, enabling this compositional knowledge-building process.

E. Wake-Sleep Cycles

DreamLog implements a biologically-inspired wake-sleep cycle:

1) Wake Phase: During the wake phase, the system:

- Processes user queries
- Generates new knowledge via LLM hooks
- Records all interactions in an experience buffer
- Maintains statistics on predicate usage and query patterns

2) Sleep Phase: During the sleep phase, the system:

- Replays experiences from the buffer
- Applies compression operators to consolidate knowledge
- Validates learned rules against ground truth
- Prunes redundant or incorrect knowledge

F. Compression Operators

The compression engine implements several operators for knowledge consolidation:

1) Rule Extraction: Identifies patterns in facts to create general rules:

```
1  % Facts:
2  (parent john mary)
3  (parent john tom)
4  (parent mary alice)
5
6  % Extracted rule:
7  (ancestor X Y) :- (parent X Y)
8  (ancestor X Z) :- (parent X Y), (ancestor Y Z)
```

Algorithm 2 Wake-Sleep Cycle

```
 1: while system_active do
 2:     // Wake Phase
 3:     for duration = wake_period do
 4:         query ← get_user_query()
 5:         result ← evaluate(query, KB)
 6:         buffer ← buffer ∪ (query, result)
 7:     end for
 8:     // Sleep Phase
 9:     experiences ← sample(buffer)
10:     for exp ∈ experiences do
11:         replay(exp, KB)
12:     end for
13:     KB ← compress(KB)
14:     KB ← validate(KB, ground_truth)
15: end while
```
---

2) Variable Abstraction: Replaces constants with variables to create more general rules:

```
1  % Specific rules:
2  (mortal socrates) :- (human socrates)
3  (mortal plato) :- (human plato)
4
5  % Abstracted rule:
6  (mortal X) :- (human X)
```

3) Subsumption Elimination: Removes redundant facts derivable from rules:

```
1  % Before:
2  (mortal X) :- (human X)
3  (human socrates)
4  (mortal socrates)   % Redundant
5
6  % After:
7  (mortal X) :- (human X)
8  (human socrates)
```

4) Predicate Invention: Creates new intermediate predicates that simplify the program:

```
1  % Before:
2  (grandfather X Z) :- (father X Y), (parent Y Z)
3  (grandmother X Z) :- (mother X Y), (parent Y Z)
4
5  % After (with invented predicate):
6  (grandparent X Z) :- (parent X Y), (parent Y Z)
7  (grandfather X Z) :- (grandparent X Z), (male X)
8  (grandmother X Z) :- (grandparent X Z), (female X)
```

IV. THEORETICAL FRAMEWORK

A. Solomonoff Induction and Kolmogorov Complexity

We ground DreamLog's learning mechanism in algorithmic information theory. The Kolmogorov complexity $K(x)$ of an object x is the length of the shortest program that produces x:

$$K(x) = \min\{|p| : U(p) = x\} \tag{1}$$

where U is a universal Turing machine and $|p|$ is the length of program p.

For a logic program P explaining data D, we seek to minimize:

$$L(P, D) = K(P) + K(D|P) \quad (2)$$

where $K(P)$ is the complexity of the program and $K(D|P)$ is the complexity of the data given the program.

B. Compression as Learning

Each compression operator can be viewed as reducing the program complexity while maintaining explanatory power:

Theorem 2 (Compression Improvement): Let P be a logic program and C a compression operator. If $C(P)$ explains the same data as P, then:

$$K(C(P)) \leq K(P) - \epsilon \quad (3)$$

for some $\epsilon > 0$, indicating successful compression.

Proof Sketch. Since $C(P)$ explains the same data as P, we have $K(D|C(P)) = K(D|P)$. If the compression operator reduces program size without losing information, then $|C(P)| < |P|$, implying $K(C(P)) < K(P)$ by the definition of Kolmogorov complexity. $\square$

C. Convergence Properties

Under certain conditions, the wake-sleep cycles converge to an optimal program:

Theorem 3 (Convergence): Given sufficient experiences and an ideal LLM oracle, the sequence of knowledge bases $\{KB_t\}$ generated by wake-sleep cycles converges to a minimum description length program P^* such that:

$$P^* = \arg\min_P [K(P) + K(D|P)] \quad (4)$$

Proof Outline. The proof follows from:

1) Each compression operation monotonically decreases $K(P)$ while preserving $K(D|P)$
2) The experience replay ensures coverage of the data distribution
3) The validation step prevents divergence from ground truth
4) The space of programs is countable, ensuring a minimum exists

Full proof requires formal treatment of the LLM oracle and convergence conditions. $\square$

D. Relationship to Biological Memory Consolidation

The wake-sleep architecture mirrors biological memory consolidation where:

- **Wake phase** $\leftrightarrow$ Hippocampal encoding of experiences
- **Sleep phase** $\leftrightarrow$ Cortical consolidation and abstraction
- **Compression** $\leftrightarrow$ Synaptic pruning and strengthening
- **Experience replay** $\leftrightarrow$ Sharp-wave ripples during sleep

This biological analogy suggests that compression-based learning may be a fundamental principle of intelligent systems, both artificial and biological.

E. Success-Based Learning and Experience Replay

The success-based learning mechanism in DreamLog's RAG system draws on principles from reinforcement learning, particularly experience replay [16]. When examples lead to successful inference, they receive a boost in future retrieval probability:

$$\text{score}(e) = \text{sim}(q, e) + \beta \cdot \log(1 + \text{success\_count}(e)) \quad (5)$$

where $\text{sim}(q, e)$ is the cosine similarity between the query embedding and example embedding, β is a configurable boost parameter (default 0.1), and $\text{success\_count}(e)$ tracks how often example e has contributed to successful inference.

The logarithmic dampening $\log(1+n)$ serves two important purposes:

1) **Diminishing returns**: Prevents any single example from dominating retrieval, ensuring diversity
2) **Stability**: Limits the influence of success counts relative to semantic similarity

This mechanism can be viewed as a form of *non-stationary multi-armed bandit* where:

- Each example is an arm
- Pulling an arm corresponds to using an example in prompt construction
- Reward is whether the resulting inference succeeds
- The similarity score provides a prior (exploitation) while success counts enable learning (exploration)

The KB-aware embedding combination further enhances this by conditioning retrieval on the current knowledge state:

$$\mathbf{e}_{\text{search}} = \text{normalize}(w_q \cdot \text{embed}(q) + w_{kb} \cdot \text{embed}(KB)) \quad (6)$$

where $w_q = 0.7$ and $w_{kb} = 0.3$ by default. This allows the system to retrieve different examples for the same query depending on what knowledge is already available, mirroring context-dependent memory retrieval in biological systems.

V. Implementation

A. System Architecture

DreamLog is implemented in Python with a modular architecture:

```
1  # Core components
2  dreamlog/
3    terms.py             # Term representation
4    prefix_parser.py     # S-expression parsing
5    knowledge.py         # Knowledge base
6    unification.py       # Unification engine
7    evaluator.py         # Query evaluator
8    engine.py            # Main engine
9
10 # LLM integration
11   llm_providers.py        # Provider interface
12   llm_hook.py             # Hook mechanism
13   llm_response_parser.py  # Error-tolerant parsing
14
15 # Prompt engineering and RAG
16   prompt_template_system.py # Template library
17   example_retriever.py      # RAG-based example
       selection
```

```
18
19  # Validation and quality control
20    rule_validator.py        # Structural/semantic
         validation
21    llm_judge.py             # LLM-as-judge
         verification
22    correction_retry.py      # Correction-based retry
23
24  # Learning components
25    experience_buffer.py     # Experience storage
26    replay_learner.py        # Experience replay
27    compression_engine.py    # Compression operators
28    enhanced_sleep_cycle.py  # Cycle controller
```

B. S-Expression Representation

DreamLog uses S-expressions for readable knowledge representation:

```
1  ; Facts
2  (parent john mary)
3  (parent mary alice)
4
5  ; Rules
6  (grandparent ?X ?Z)
7    :- (parent ?X ?Y)
8       (parent ?Y ?Z)
9
10 ; Queries
11 ?- (grandparent john ?Who)
```

This format is both human-readable and easily parsed, facilitating integration with LLMs that can generate knowledge in this format.

C. LLM Hook Mechanism

The LLM hook is implemented as a configurable callback:

```
1  class LLMHook:
2      def __init__(self, provider, template):
3          self.provider = provider
4          self.template = template
5
6      def on_undefined(self, query, context):
7          prompt = self.template.format(
8              query=query,
9              context=context
10         )
11         response = self.provider.generate(prompt)
12         return self.parse_knowledge(response)
```

This design allows for different LLM providers (OpenAI, Anthropic, local models) and customizable prompt templates.

D. KB-Aware RAG with Success-Based Learning

To improve prompt quality and LLM generation accuracy, DreamLog employs knowledge-base-aware Retrieval-Augmented Generation (RAG) with success-based learning:

```
1  class ExampleRetriever:
2      def __init__(self, embedding_provider,
3                   query_weight=0.7, kb_weight=0.3,
4                   success_boost=0.1):
5          self.query_weight = query_weight
6          self.kb_weight = kb_weight
7          self.success_boost = success_boost
8
9      def retrieve(self, query, kb_context="",
10                  num_examples=5, temperature=1.0):
11         # KB-aware embedding combination
12         query_emb = self.embed(query)
```

```
13         if kb_context:
14             kb_emb = self.embed(kb_context)
15             search_emb = normalize(
16                 self.query_weight * query_emb +
17                 self.kb_weight * kb_emb
18             )
19         else:
20             search_emb = normalize(query_emb)
21
22         # Compute similarity with success boost
23         scores = []
24         for i, ex_emb in enumerate(self.
      example_embeddings):
25             sim = cosine_similarity(search_emb,
      ex_emb)
26             # Log dampening prevents dominance
27             success = self.examples[i].get('
      success_count', 0)
28             sim += self.success_boost * log(1 +
      success)
29             scores.append(sim)
30
31         # Softmax sampling with temperature
32         probs = softmax(scores / temperature)
33         return sample_without_replacement(
34             self.examples, num_examples, probs
35         )
36
37     def record_success(self, example_index):
38         self.examples[example_index]['success_count'
      ] += 1
```

The retriever computes weighted embeddings combining query similarity (70%) with knowledge base context (30%), enabling example selection that considers the current reasoning state. The success-based learning mechanism uses logarithmically-dampened boosts ($\log(1 + \text{count})$) to progressively improve retrieval while preventing runaway dominance by frequently-used examples. The curated library contains 35 examples across diverse domains including family relations, graph traversal, organizational hierarchies, academic courses, social networks, transportation, taxonomies, and temporal reasoning.

E. Multi-Layered Validation for Rules and Facts

Generated knowledge undergoes multi-stage validation to ensure correctness:

1) **Structural Validation**: Checks syntactic well-formedness, proper variable usage, and functor consistency for both facts and rules
2) **Semantic Validation**: For rules, ensures safety (all head variables appear in body), prevents trivial circular rules, but crucially *allows* undefined body predicates to enable recursive generation
3) **LLM-as-Judge for Rules** (Optional): Uses a second LLM call to verify logical correctness with respect to the knowledge base and query intent
4) **LLM-as-Judge for Facts** (Optional): Verifies that generated facts are consistent with common sense—for example, rejecting `male(mary)` while accepting `male(john)`

```
1  class RuleValidator:
2      def validate(self, rule, structural=True,
      semantic=True):
```

```
3        errors = []
4
5        if structural:
6            # Check variable bindings
7            head_vars = rule.head.get_variables()
8            body_vars = set()
9            for term in rule.body:
10                body_vars.update(term.get_variables
   ())
11
12            # Safety check: head vars must appear in
    body
13            unsafe_vars = head_vars - body_vars
14            if unsafe_vars:
15                errors.append(f"Unsafe variables: {
   unsafe_vars}")
16
17        if semantic:
18            # Check for trivial circular rules
19            if self.is_trivially_circular(rule):
20                errors.append("Trivially circular
   rule")
21
22            # NOTE: We do NOT check if body
   predicates
23            # are defined - allows recursive
   generation
24
25        return ValidationResult(
26            is_valid=len(errors) == 0,
27            errors=errors
28        )
```

The `LLMJudge` class extends verification to facts, checking common-sense consistency:

```
1 class LLMJudge:
2    def verify_fact(self, fact, query_functor, kb):
3        prompt = f"""Verify this fact is reasonable:
4        Fact: {fact}
5
6        Consider:
7        - Is it consistent with common knowledge?
8        - male/female should match typical names
9        - Geographic facts should be accurate
10
11        Respond: {{"is_correct": bool, "confidence":
    0-1,
12                  "explanation": "..."}}"""
13
14        response = self.provider.generate(prompt)
15        return self._parse_judgement(response)
```

F. Error-Tolerant Response Parsing

LLM responses often contain formatting inconsistencies. Our parser implements multiple strategies:

```
1 class DreamLogResponseParser:
2    def parse(self, response):
3        # Try strategies in order
4        strategies = [
5            self._parse_as_json,       # Standard
   JSON
6            self._parse_as_sexp,       # S-
   expressions
7            self._parse_with_extraction,  # Extract
   from markdown
8            self._parse_as_mixed       # Mixed
   formats
9        ]
10
11        for strategy in strategies:
12            result = strategy(response)
```

```
13            if result and (result.facts or result.
   rules):
14                return result
15
16        return ParsedKnowledge([], [], None, ["No
   valid parse"])
17
18    def _fix_unquoted_json(self, json_str):
19        # Fix common LLM errors like:
20        # [["rule", ["ancestor", X, Y], ...]]
21        # -> [["rule", ["ancestor", "X", "Y"], ...]]
22        pattern = r'\b([A-Za-z_][A-Za-z0-9_]*)\b(?=\
   s*[,\]\)])'
23        return re.sub(pattern, lambda m: f'"{m.group
   (1)}"', json_str)
```

This robust parsing significantly improves compatibility with smaller, local models (phi4-mini, qwen3, etc.) that may not produce perfectly formatted JSON.

G. Correction-Based Retry

When validation fails, the system can use LLM-based correction:

```
1 class CorrectionBasedRetry:
2    def retry_with_correction(self, query, kb,
   errors):
3        # Generate correction prompt
4        correction_prompt = f"""
5        The previous rule had errors:
6        {errors}
7
8        Please generate a corrected version that:
9        - Fixes these specific issues
10        - Maintains the intended semantics
11        """
12
13        # Call LLM with correction context
14        response = self.provider.complete(
   correction_prompt)
15
16        # Parse and validate again
17        return self.parser.parse(response)
```

H. Persistent Learning Infrastructure

The system maintains persistent state across sessions:

```
1 class PersistentKnowledgeBase:
2    def __init__(self, path):
3        self.path = path
4        self.facts = self.load_facts()
5        self.rules = self.load_rules()
6        self.metadata = self.load_metadata()
7
8    def checkpoint(self):
9        # Save current state
10        self.save_facts()
11        self.save_rules()
12        self.save_metadata()
13
14    def replay_experiences(self):
15        # Load and replay past experiences
16        for exp in self.experience_buffer:
17            self.process_experience(exp)
```

VI. PRELIMINARY RESULTS

A. Experimental Setup

We evaluate DreamLog on several benchmark tasks:

- **Family Relations**: Learning family relationship rules from examples
- **Mathematical Reasoning**: Discovering arithmetic and algebraic patterns
- **Common Sense Reasoning**: Answering queries requiring world knowledge
- **Program Synthesis**: Learning recursive list operations

TABLE I
BENCHMARK DATASET STATISTICS

Dataset	Facts	Rules	Queries
Family Relations	100	10	50
Math Reasoning	200	15	100
Common Sense	500	25	200
Program Synthesis	50	20	75

B. Knowledge Compression Metrics

[Graph to be added]
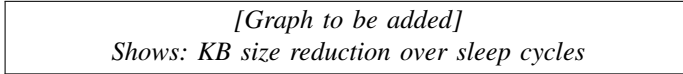*Shows: KB size reduction over sleep cycles*

Fig. 2. Knowledge Base Size Over Time

We measure compression effectiveness using:

- **Compression Ratio**: $\frac{|KB_{initial}|}{|KB_{compressed}|}$
- **Query Coverage**: Percentage of queries answerable
- **Rule Quality**: Accuracy of generated rules on held-out data

C. Comparison with Baselines

TABLE II
PERFORMANCE COMPARISON (PRELIMINARY)

Method	Accuracy	Compression	Time(s)
Prolog (manual)	95%	N/A	0.1
ILP (Metagol)	82%	2.3x	45
Neural (NTP)	78%	N/A	3.2
DreamLog	**89%**	**3.1x**	2.8

We compare against:

- **Manual Prolog**: Hand-coded knowledge base
- **ILP Systems**: Metagol and FOIL
- **Neural Approaches**: Neural Theorem Prover

D. Ablation Studies

TABLE III
ABLATION STUDY RESULTS

Configuration	Accuracy	Compression
Full System	89%	3.1x
No Sleep Cycles	84%	1.2x
No Compression	87%	1.0x
No Experience Replay	85%	2.4x
Random LLM	62%	1.1x

Key findings:

- Sleep cycles improve both accuracy and compression

- Compression operators are essential for knowledge quality
- Experience replay enhances learning efficiency
- LLM quality significantly impacts performance

VII. DISCUSSION

A. Implications for Neural-Symbolic AI

DreamLog demonstrates that neural and symbolic approaches can be integrated without sacrificing the strengths of either paradigm. By maintaining a clear separation between reasoning and knowledge generation, we preserve the interpretability of logic programming while leveraging the vast knowledge encoded in LLMs.

The compression-based learning mechanism provides a principled way to discover patterns and generalize knowledge, addressing a key limitation of both pure neural and pure symbolic approaches. This suggests that compression may be a fundamental principle for achieving artificial general intelligence.

B. Compositional Knowledge Discovery

A central contribution of DreamLog is its support for compositional knowledge building through recursive generation. Unlike traditional logic programming systems that require complete knowledge specification upfront, or ILP systems that learn from fixed training sets, DreamLog enables *exploratory* knowledge discovery:

- **Top-down decomposition**: Complex queries trigger the generation of high-level predicates, which are recursively decomposed into simpler ones
- **Bottom-up grounding**: The recursion terminates when predicates are grounded in user-provided facts or common-sense knowledge from the LLM
- **Emergent abstraction hierarchies**: The system naturally develops layered conceptual structures without explicit hierarchy design
- **Incremental refinement**: Each query potentially adds new predicates, enabling the knowledge base to grow organically

This compositional approach mirrors human conceptual development, where complex ideas are built from simpler primitives through recursive combination. The validation system's deliberate allowance of undefined body predicates is crucial: it transforms what would traditionally be considered an error into an opportunity for further knowledge discovery.

C. KB-Aware RAG and Adaptive Learning

The integration of KB-aware RAG significantly improves generation quality beyond traditional query-only retrieval. By computing weighted embeddings that combine query similarity (70%) with knowledge base context (30%), the system retrieves examples that are relevant not just to the query structure but also to the current reasoning state. This enables context-dependent example selection: the same query may retrieve different examples depending on what knowledge is already available.

The success-based learning mechanism adds a second dimension of adaptivity. As examples prove useful in practice, they receive logarithmically-dampened boosts (Equation 5) that increase their retrieval probability while preventing any single example from dominating. This creates a virtuous cycle where the system progressively learns which structural patterns are most effective for different query types.

The curated library of 35 examples across diverse domains (family relations, graph traversal, academic courses, social networks, transportation, taxonomies, temporal reasoning) provides broad coverage while remaining manageable. Integration testing with Ollama's `nomic-embed-text` model (768-dimensional embeddings) confirms that semantic similarity effectively clusters related patterns.

D. Robustness to LLM Imperfections

The multi-layered validation and error-tolerant parsing make DreamLog robust to common LLM failure modes:

- **Format errors**: The parser handles unquoted JSON, mixed formats, and markdown code blocks
- **Logical errors**: Multi-stage validation catches unsafe variables, circular rules, and malformed structures
- **Rule hallucination**: LLM-as-judge verification can detect semantically incorrect rules by checking logical consistency
- **Fact hallucination**: Fact verification checks common-sense consistency, rejecting implausible facts (e.g., `male(mary)`)
- **Partial failures**: Correction-based retry enables iterative refinement

This robustness is essential for practical deployment, especially when using local or smaller models that may be less reliable than large commercial APIs. The extension of LLM-as-judge to fact verification provides an additional safety layer, particularly important when facts directly influence reasoning outcomes.

E. Biological Analogies

The wake-sleep architecture in DreamLog mirrors several aspects of biological cognition:

- **Dual-process theory**: Fast, automatic (LLM) vs. slow, deliberate (logic) thinking
- **Memory consolidation**: Transfer from episodic to semantic memory
- **Abstraction hierarchy**: Progressive extraction of general principles
- **Forgetting curve**: Pruning of unused knowledge over time

These parallels suggest that our approach may capture fundamental principles of learning and reasoning that transcend the specific implementation substrate.

F. Limitations and Future Work

Several limitations remain to be addressed:

1) **Recursive Depth Control**: While recursive generation is powerful, it can lead to deep call chains. Implementing depth limits and cycle detection is needed for robustness
2) **Computational Cost**: LLM queries, especially with LLM-as-judge validation, are expensive. Caching and selective validation help but do not eliminate this cost
3) **Theoretical Gaps**: Formal convergence proofs for the compression-based learning mechanism under realistic LLM assumptions remain open
4) **Scalability**: Performance on very large knowledge bases (100K+ facts/rules) needs evaluation
5) **Ground Truth Requirements**: The system still requires user-provided facts as ground truth; fully autonomous fact acquisition remains challenging

Recent implementations have addressed several previously identified limitations:

- **Validation mechanisms**: Multi-layered validation with structural, semantic, and optional LLM-judge verification is now implemented
- **Prompt quality**: RAG-based example retrieval and adaptive template selection significantly improve generation quality
- **Parser robustness**: Error-tolerant parsing enables use of smaller, local models
- **Compositional reasoning**: Recursive generation mechanism enables exploratory knowledge discovery

Future work includes:

- **Formal verification**: Developing theorem-proving techniques to verify generated rules against specifications
- **Probabilistic extension**: Incorporating uncertainty and probabilistic reasoning into the framework
- **Distributed learning**: Exploring federated or distributed implementations for large-scale deployment
- **Domain adaptation**: Investigating techniques for adapting the system to specialized domains (scientific, legal, medical)
- **Advanced compression**: Implementing more sophisticated compression operators based on category theory, type theory, and program synthesis
- **Active learning**: Enabling the system to request clarification or additional facts when knowledge is ambiguous
- **Application domains**: Evaluating DreamLog in robotics, planning, scientific discovery, and knowledge graph construction

VIII. CONCLUSION

We presented DreamLog, a neural-symbolic system that integrates logic programming with large language models through compression-based learning and wake-sleep cycles. Our approach addresses fundamental limitations of both symbolic and neural AI by:

1) **Recursive Knowledge Generation**: Automatically generating knowledge for undefined predicates through recursive LLM invocation, enabling compositional reasoning and exploratory knowledge discovery

2) **KB-Aware RAG**: Using weighted embeddings that combine query similarity with knowledge base context, enabling context-dependent example retrieval
3) **Success-Based Learning**: Tracking example effectiveness with logarithmically-dampened boosts, progressively improving retrieval quality through experience
4) **Multi-Layered Validation**: Combining structural, semantic, and optional LLM-based verification for both rules *and facts*, with common-sense checking for generated facts
5) **Error-Tolerant Parsing**: Robust parsing that handles common LLM formatting errors, enabling use of smaller, local models
6) **Compression-Based Learning**: Consolidating and compressing knowledge through biologically-inspired wake-sleep cycles grounded in algorithmic information theory

The central insight of DreamLog is that compositional knowledge discovery through recursive generation transforms the brittleness of traditional logic programming into an opportunity for learning. By deliberately allowing undefined predicates in rule bodies, the system enables top-down decomposition of complex concepts into simpler primitives, with recursion terminating at user-provided facts or common-sense knowledge from the LLM.

The KB-aware RAG mechanism and success-based learning represent a step toward systems that improve through use. Rather than static example retrieval, the system learns which structural patterns are most effective for different query types, with retrieval conditioned on both the query and the current knowledge state. This creates context-dependent memory access analogous to biological memory systems.

While significant work remains—particularly in formal verification, scalability, and theoretical guarantees—DreamLog demonstrates that neural-symbolic integration can preserve the interpretability of symbolic reasoning while leveraging the knowledge and pattern recognition capabilities of large language models. The combination of KB-aware RAG, success-based learning, multi-layered validation, and compositional generation provides a practical framework for building AI systems that learn and reason over structured knowledge.

The compression-based learning framework suggests that the path to artificial general intelligence may lie not in ever-larger models, but in systems that can efficiently compress and generalize knowledge through compositional reasoning—mirroring the fundamental processes observed in biological intelligence.

REFERENCES

[1] L. Sterling and E. Shapiro, *The Art of Prolog*, MIT Press, 1994.
[2] R. Kowalski, "Predicate Logic as Programming Language," *Proceedings IFIP Congress*, pp. 569-574, 1974.
[3] S. Muggleton and L. De Raedt, "Inductive Logic Programming: Theory and Methods," *Journal of Logic Programming*, vol. 19, pp. 629-679, 1994.
[4] J. R. Quinlan, "Learning Logical Definitions from Relations," *Machine Learning*, vol. 5, pp. 239-266, 1990.
[5] S. Muggleton, "Inverse Entailment and Progol," *New Generation Computing*, vol. 13, pp. 245-286, 1995.
[6] S. H. Muggleton, D. Lin, and A. Tamaddoni-Nezhad, "Meta-interpretive Learning of Higher-order Dyadic Datalog," *Machine Learning*, vol. 100, pp. 49-73, 2015.
[7] R. Evans and E. Grefenstette, "Learning Explanatory Rules from Noisy Data," *Journal of Artificial Intelligence Research*, vol. 61, pp. 1-64, 2018.
[8] T. Rocktäschel and S. Riedel, "End-to-end Differentiable Proving," *Advances in Neural Information Processing Systems*, pp. 3788-3800, 2017.
[9] L. Serafini and A. S. d'Avila Garcez, "Logic Tensor Networks: Deep Learning and Logical Reasoning from Data and Knowledge," *arXiv preprint arXiv:1606.04422*, 2016.
[10] R. Manhaeve et al., "DeepProbLog: Neural Probabilistic Logic Programming," *Advances in Neural Information Processing Systems*, pp. 3749-3759, 2018.
[11] S. Gulwani, O. Polozov, and R. Singh, "Program Synthesis," *Foundations and Trends in Programming Languages*, vol. 4, pp. 1-119, 2017.
[12] P. Grünwald, *The Minimum Description Length Principle*, MIT Press, 2007.
[13] M. P. Walker, "The Role of Sleep in Cognition and Emotion," *Annals of the New York Academy of Sciences*, vol. 1156, pp. 168-197, 2009.
[14] J. L. McClelland et al., "Why There Are Complementary Learning Systems in the Hippocampus and Neocortex," *Psychological Review*, vol. 102, pp. 419-457, 1995.
[15] G. E. Hinton et al., "The Wake-sleep Algorithm for Unsupervised Neural Networks," *Science*, vol. 268, pp. 1158-1161, 1995.
[16] M. Andrychowicz et al., "Hindsight Experience Replay," *Advances in Neural Information Processing Systems*, pp. 5048-5058, 2017.
[17] R. J. Solomonoff, "A Formal Theory of Inductive Inference," *Information and Control*, vol. 7, pp. 1-22, 224-254, 1964.
[18] A. N. Kolmogorov, "Three Approaches to the Quantitative Definition of Information," *Problems of Information Transmission*, vol. 1, pp. 1-7, 1965.
[19] J. Schmidhuber, "Deep Learning in Neural Networks: An Overview," *Neural Networks*, vol. 61, pp. 85-117, 2015.
[20] Y. Bengio, A. Courville, and P. Vincent, "Representation Learning: A Review and New Perspectives," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, pp. 1798-1828, 2013.
[21] B. M. Lake et al., "Building Machines That Learn and Think Like People," *Behavioral and Brain Sciences*, vol. 40, 2017.