Cognitive Prompt Machines: A BASIC-Inspired Language for Cognitive Prompt Agents

Oliver Kramer
Computational Intelligence Group
Carl von Ossietzky Universität Oldenburg
Oldenburg, Germany
oliver.kramer@uni-oldenburg.de

Abstract. The Cognitive Prompt Machine (CPM) is a lightweight architecture for simulating cognitive functions in large language models (LLMs) through structured prompt programming. It introduces the Cognitive Prompt Language (CPL), a compact, line-executable instruction set for reasoning over declarative and procedural memory. Inspired by the simplicity of early BASIC interpreters, CPM provides a transparent, stepwise execution model entirely within LLM prompt space. It supports core cognitive tasks such as fact extraction, confidence evaluation, and belief revision. An example demonstrates that even this minimal setup enables transparent reasoning loops and belief updates in LLMs. The design is fully LLM-native and model-agnostic, requiring no external tooling or fine-tuning. CPM represents a step toward interpretable, self-contained reasoning agents and the emergence of Large Reasoning Models (LRMs).

1 Introduction

Cognitive systems must represent and adapt knowledge, learn from experience, and revise beliefs in light of new evidence or contradictions. This requires not only factual and procedural learning but also flexible reasoning and self-consistency mechanisms. We introduce the Cognitive Prompt Machine (CPM), an architecture that enables such capabilities in large language models (LLMs) using structured prompts called cognitive commands. Each command maps to a specific operation, e.g., fact extraction, confidence evaluation, or conflict detection, and is executed through modular prompt templates designed for cognitive prompting [1]. To operationalize this, we present a streamlined variant of CPM built around the Cognitive Prompt Language (CPL), a minimal, interpretable scripting language. CPL provides a BASIC-style execution model that directly manipulates memory through LLM calls, supporting transparent loops, reasoning traces, and belief updates. This prompt-native design enables experimentation with cognitive workflows without relying on external symbolic systems.

2 CPM Architecture

2.1 CPL: The Cognitive Prompt Language

The CPL is a lightweight, interpreted prompt language designed to enable structured reasoning and memory interaction within large language models. Unlike

traditional programming environments, CPL does not rely on external host languages such as Python. Instead, it defines all logic and execution behavior through LLM-readable command prompts. This makes CPL natively executable by any capable language model, without requiring wrappers or custom runtime environments.

Each CPL command corresponds to a predefined prompt template that guides the model to perform specific cognitive operations, such as extracting facts from input, evaluating belief strength, or resolving inconsistencies in memory. This enables a modular, transparent reasoning process with each step logged and transparently interpretable.

CPL follows a procedural programming style inspired by early languages like BASIC. Program lines are prefixed by line numbers and executed sequentially unless control flow is explicitly altered. The command `LET` sets the goal or input text. Conditional logic is handled via `IF`, `THEN`, and optionally `ELSE`. Iterative processing of memory entries is supported through the combination of `FOR` and `NEXT`. Direct jumps can be made using `GOTO`, and program execution concludes with `END`.

These commands allow for the construction of cognitively meaningful workflows, such as extracting all declarative facts from a passage, looping through them to assign confidence scores, checking for contradictions, and applying revision mechanisms. As demonstrated in Section 3, a simple CPL script can evaluate a chain of transitive relationships, identify logical conflicts, and generalize the memory content accordingly.

The design is minimal yet extensible. New commands can be defined via prompt templates and added to the interpreter without altering its execution logic. This enables the system to grow in functionality as needed. Because CPL relies only on textual prompts and structured memory access, it can be deployed across various LLM backends, locally or via APIs, making it a highly portable framework for cognitive-level prompting.

2.2 Memory Structure

The CPM framework manages an internal memory composed of three main components: the `goal`, which defines the current task or inquiry guiding the system's reasoning; the `working memory`, a short-term volatile buffer for intermediate results, loop variables, and conflict states; and the `mental models`, which encode structured long-term knowledge. Both working memory and mental models operate using two core types of knowledge representation:

`Declarative memory` stores factual assertions extracted from text, each annotated with a confidence score reflecting the system's degree of belief. For example, in a reasoning scenario involving visual similarity (*"A looks like B"*), declarative memory may include transitive statements such as *"B looks like C"* and *"A looks like C"*, along with revisions like *"A is not a typical C"* if contradictions arise.

`Procedural memory` holds action-oriented knowledge—what the system knows how to do. Each entry represents a generalized reasoning step or cognitive op-

eration, such as *"Trace inheritance relationships"* or *"Resolve logical contradictions"*. These procedures can be evaluated based on their expected utility or effectiveness in achieving the current goal, allowing the system to adaptively refine its strategies over time—for instance, when reasoning with LLM tools.

This modular architecture enables CPM agents to reason, detect contradictions, update beliefs through commands like `Assimilate`, and store and reuse patterns of thought, making the system both reflective and adaptive.

2.3 Cognitive Commands

The CPM framework supports a range of cognitive commands that emulate core mental operations. In the declarative domain, `ExtractFacts` is responsible for parsing natural language input and converting it into discrete, structured factual assertions. These assertions are stored in memory along with an initial confidence value. For example, the input *"A looks like B. B can never look like C. A looks like C."* yields several transitive and potentially contradictory facts. Each is subsequently refined using the `EvaluateFacts` command, which assigns or adjusts confidence levels based on internal consistency and prior knowledge.

To manage procedural knowledge, `EvaluateSteps` updates the utility of stored action patterns, such as transitive reasoning or conflict detection, based on how effective they are in resolving reasoning tasks. For higher-level cognitive adaptation, the command `IdentifyConflict` checks whether newly added facts contradict existing ones in declarative memory. When contradictions are found, the `Assimilate` command is triggered. This command attempts to resolve inconsistencies by applying belief revision strategies such as weakening confidence, adding qualifiers, or generalizing statements. In the example, the contradictory input *"A looks like C"* and *"B can never look like C"* triggers a revision of the conclusion to *"A is not a typical C"*.

The interpreter-level definition of `Assimilate` is shown below:

```
COMMAND: Assimilate
PROMPT: Reconcile contradictions in declarative memory by generalizing,
    adding exceptions, or revising conflicting facts.
WRITE: declarative <- updated memory with resolved contradictions
```

These modular commands together simulate essential cognitive functions such as knowledge extraction, belief calibration, contradiction detection, and knowledge restructuring.

2.4 Output Format

Each CPL program is executed line by line, with the system producing a transparent and interpretable log of its reasoning process. The output for every step follows a structured format that includes the prompt, the language model's response, any modifications to memory, and the next line to be executed. This format supports traceability and introspection, which are essential for analyzing cognitive processes in LLMs.

```
===[LINE: <LineNumber>]===
PROMPT -> <filled prompt>
RESPONSE -> <LLM output>
MEMORY[...] <- <memory changes>
NEXT -> <next line number or END>
```

3 Example

This example demonstrates how the CPL can be used to detect and resolve a logical contradiction in a transitive reasoning chain. The program receives a short input about visual similarity, extracts relevant declarative facts, evaluates their confidence, and triggers conflict resolution if necessary. The entire process runs on GPT-4o, a capable LLM that interprets each CPL instruction as a structured prompt and updates its internal memory accordingly. This showcases how even lightweight cognitive programs can be executed efficiently on modern language models without additional infrastructure.

3.1 CPL Program

```
10 LET input = "A looks like B. B can never look like C. A looks like C."
20 ExtractFacts input
30 FOR fact in declarative
40  EvaluateFacts fact
50  IF IdentifyConflict fact == True THEN
60    Assimilate fact
70 NEXT
80 END
```

The program begins by setting the input and extracting facts. Each fact is then evaluated individually for confidence, and potential contradictions are identified. If a conflict arises, it is resolved using the `Assimilate` command, which revises memory entries to restore coherence.

3.2 Output During Runtime

In the transitivity example, the program automatically loops over extracted facts, evaluates their confidence, detects logical contradictions, and applies conflict resolution strategies. Each operation is logged with its prompt, response, memory changes, and next-step directive.
Example:

```
===[LINE: 50]===
PROMPT -> Identify conflicts with: "A looks like C"
RESPONSE -> Conflict with: "B can never look like C"
MEMORY[working_memory] <- conflict = TRUE
NEXT -> 60
```

3.3 Final Memory State

After execution, the memory reflects the updated understanding of conflicting statements. This illustrates how CPL enables belief revision and adaptive reasoning through structured prompts and memory tracking.

```
GOAL: Understand the text: A is a B, B is a D, A is not a D
WORKING MEMORY: fact: A is not a D, conflict: TRUE
DECLARATIVE MEMORY:
  A is a B [confidence: high]
  B is a D [confidence: high]
  A is not a typical D [confidence: revised]
PROCEDURAL MEMORY:
  Trace inheritance relationships
  Detect transitive inconsistencies
  Resolve logical contradictions
```

4 Related Work

The Cognitive Prompt Machine (CPM) relates to several strands of research, including cognitive architectures, prompt-based agents, memory-augmented language models, and neurosymbolic reasoning. Its main novelty lies in combining these threads into a lightweight, prompt-native framework that operates entirely within the language model's context, without relying on external symbolic infrastructure.

Cognitive architectures such as ACT-R [2], SOAR [3], and OpenCog Hyperon [4] implement modular memory systems and rule-based control mechanisms. CPM adopts the declarative–procedural memory distinction from these systems but replaces explicit symbolic rules with natural language prompt templates, preserving modularity while enhancing interpretability.

Prompt-based agent frameworks like ReAct [5], Chain-of-Thought (CoT) prompting [6], Toolformer [7], LangChain, and LangGraph focus on reasoning through tool use, intermediate steps, and planning chains. In contrast, CPM introduces CPL as a minimal domain-specific language that provides transparent memory access and structured, step-by-step execution semantics for LLM reasoning.

Memory-augmented architectures such as MemGPT [8] externalize memory to support extended reasoning. CPM internalizes memory structures—including declarative facts, procedural steps, and mental models—within its own prompt-executed interpreter, enabling internal cognitive processes without architectural modifications.

Finally, neurosymbolic systems like DreamCoder [9] and Neural Programmer-Interpreters [10] integrate neural networks with symbolic components for program synthesis and reasoning. CPM takes a different approach: rather than blending symbolic logic with neural models, it encodes reasoning and control flow

directly into prompt-level logic, maintaining transparency and LLM-compatibility throughout.

5 Conclusion

CPM bridges the gap between symbolic reasoning systems and prompt-based LLMs. Our implementation demonstrates that structured declarative and procedural knowledge can be maintained, evaluated, and updated entirely through prompt-level control, without reliance on external programming frameworks. The interpreter supports modular mental models, which enable the storage and reuse of domain-specific knowledge, and are already implemented in the GitHub prototype[1] as flexible lists of key–model pairs. Future work will explore extending CPL with higher-level reasoning operators, richer memory structures (e.g., episodic or analogical memory), and automatic learning of command sequences, paving the way for autonomous, self-improving cognitive agents.

References

[1] Oliver Kramer and Jill Baumann. Unlocking structured thinking in language models with cognitive prompting. In *European Symposium on Artificial Neural Networks (ESANN)*, 2025.

[2] John R. Anderson, Daniel Bothell, Michael D. Byrne, Scott Douglass, Christian Lebiere, and Yulin Qin. An integrated theory of the mind. *Psychological Review*, 111(4):1036–1060, 2004.

[3] John E. Laird. *The Soar Cognitive Architecture*. MIT Press Cognition and Perception. MIT Press, 1st edition, 2012.

[4] Ben Goertzel, Vitaly Bogdanov, Michael Duncan, Deborah Duong, Zarathustra Goertzel, Jan Horlings, Matthew Ikle', Lucius Greg Meredith, Alexey Potapov, André Luiz de Senna, Hedra Seid Andres Suarez, Adam Vandervorst, and Robert Werko. Opencog hyperon: A framework for agi at the human level and beyond. *arXiv preprint arXiv:2310.18318*, 2023.

[5] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *Neural Information Processing Systems (NeurIPS)*, pages 1–14, 2022.

[6] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, pages 24877–24891. Curran Associates, Inc., 2022.

[1] `https://github.com/evolution-strategies/cognitive-prompt-machine`

[7] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Neural Information Processing Systems (NeurIPS 2023)*, 2023.

[8] Charles Packer, Vivian Fang, Shishir G. Patil, Kevin Lin, Sarah Wooders, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems. *CoRR*, abs/2310.08560, 2023.

[9] Kevin Ellis, Lionel Wong, Maxwell Nye, Mathias Sablé-Meyer, Luc Cary, Lore Anaya Pozo, Luke Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *Philosophical Transactions of the Royal Society A*, 381(2251), 2023.

[10] Scott Reed and Nando de Freitas. Neural programmer-interpreters. *CoRR*, abs/1511.06279, 2015.