

Decompiling Language Models into Logic Programs

Jacinto Dávila Quintero¹

¹ Universidad de Los Andes, CESIMO, Mérida, Venezuela

Abstract

In this paper, we sketch a strategy to take advantage of LLM's capabilities with natural language processing while avoiding the risk of incorrect outputs. We aim to extract knowledge from LLMs into logic programs and queries and, therefore, add LLMs to the materials and tools used to teach logic programming.

Keywords

Decompiling, LLM, Logic Programming

1. Introduction

Large Language Models, LLM, are a compiled, sub-symbolic form of knowledge representation that can achieve human level performance in natural language processing[1]. The transformer architecture[2], that creates and processes LLMs, is an stochastic machine able to generate new, original, human-like outputs from natural language inputs (prompts), after having been trained with massive amounts of data[3]. This relatively new and increasingly popular brand of Artificial Intelligence Applications has been termed Generative AI[4]. Unfortunately, the stochastic machine is bound to produce some outputs that can not be logically justified from the inputs or the training data. These "original" outputs, sometimes called hallucinations, cannot be avoided[5] and can be regarded as serious errors in many domains of use.

In this paper, we sketch a strategy to take advantage of LLM's capabilities with natural language processing while avoiding the risk of incorrect outputs. We aim to extract knowledge from LLMs into logic programs and queries. We can then use the LLM facing the human user to receive natural language input, translate them into logic, call a separate logical engine to reason with them, and take whatever answer is produced by the engine back into the language the user understand.

2. Related Work

This type of hybrid systems, combining Generative AI with a form of Logical AI, some times also called neuro-symbolic systems, are being researched and tested intensively [6], [7], [8]. These are the latest developments of a trend to improve the reasoning capabilities of LLMs that started with "internal" solutions such as Chain-of-Thought[9], C-o-T, which prompted for a series of intermediate reasoning steps to improve the ability of the LLM to perform complex reasoning. C-o-t, however, does not prevent the aforementioned hallucination errors and even fails to justify that the referred reasoning is what the LLM is actually performing.

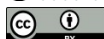
We are following the lead of solutions that integrate some well-known logical reasoner with the LLM so that the logic that it is been followed when a conclusion is reached or a question is answered is perfectly well accounted for [10], even when the reasoning must be adapted to provide a fit to the LLM, like in soft-unification on noisy facts [11]. A more extreme approach is to actually

Joint Proceedings of the Workshops and Doctoral Consortium of the 41st International Conference on Logic Programming, September 9—13, 2025, Rende, Italy

Corresponding author.

✉ jacinto@ula.ve (J.A. Dávila Quintero);

ORCID 0000-0003-1053-8763 (Jacinto Dávila)



© 2025 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

intervene the Transformer architecture with a representation closer connected with logical formulae and a process that matches a reasoning system, as done by Thuy & Yamamoto [12]. Unfortunately, this requires a different neural system and may compromise the efficiency of the resulting models for natural language processing.

There is increasing evidence that to guarantee trustworthiness of sub-symbolic models, they must be integrated with some robust, external reasoner (a hybrid system [13]). A latter study suggest that language reasoning models, LRMs, versions of LLMs optimized for reasoning, actually collapse while solving reasoning problems at a certain scale [14].

In this paper, as a teaching exercise, we want to step back before moving onto the integration of an LLM with a reasoner, to revisit the basic structures and practical elements that underlie language models as they are understood in Generative AI. After, we will relate some experiments to decompile knowledge from an LLM into logic programs that provide explainable answers to users' questions.

3. Back to first principles

3.1. Compiling and de-compiling

A **compiler** is a basic concept in Computer Science: “A compiler is a program that can read a program in one language, the source language, and translate it into an equivalent program in another language, the target language” [15]. In Computing, the code in the target language is known as a compiled set of instructions that a machine can take as input and then follow to perform some computation, generating some outputs in the process. Another related concept is an **interpreter**, “another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user” (ibid). In both cases, it is all about translating human instructions into machine actions. Figure 1 depicts both processes:

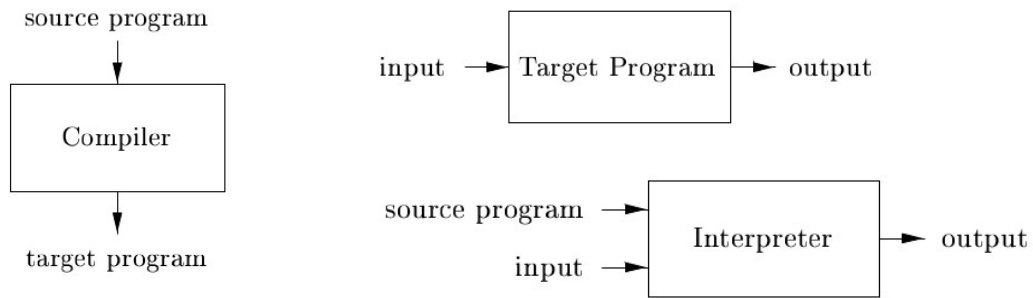


Figure 1: A compiler produces a target code from some source program. The target can be run to produce some output from a given input. An interpreter does something similar, except that it translates the source step by step while running it, processing inputs and producing the outputs (images taken from Aho *et al* [15]).

De-compiling would be the reverse process of obtaining the source program given the target program. When the target is a very basic code, like a binary representation, the process faces an enormous combinatorial complexity and it is very difficult to obtain the source. In some cases, however, with the use of inputs and corresponding outputs, and some other general information, like blue-prints or frameworks associated with the whole system, de-compilation is practically possible, as depicted in figure 2.

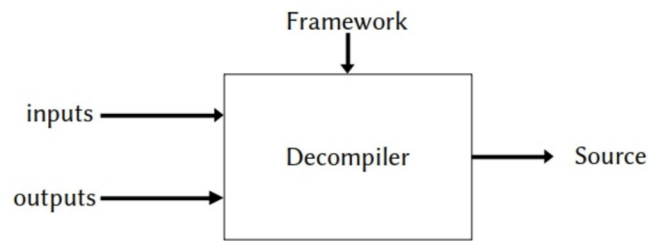


Figure 2: A *de-compiler* could generate the source code from some target enriched with contextual information (a framework) and examples of inputs and outputs.

A neural network can be regarded as a compiled representation of knowledge, as will be illustrated below. It is normally produced by a training process that short-circuits inputs and outputs and computes the weights of the network that generalize the function that associates those inputs and outputs and, hopefully, variants of them. Once a neural network is trained, its weights and biases represent the learned knowledge in an executable form. In this operational sense, the net is like a compiled program: an executable form of the original source code. We do not see the ‘source code’ (the training data and learning process) directly, but we have the ‘executable’ (the trained model) that embodies the learned relationships. Trusting these relationships implies trusting there is a meaningful connection between the ‘source code’ and the ‘executable’.

Thus, after training, the network can be used as a compiled program to produce outputs given the corresponding inputs. Figure 3 (top) represents this possibility and (middle) the training process.

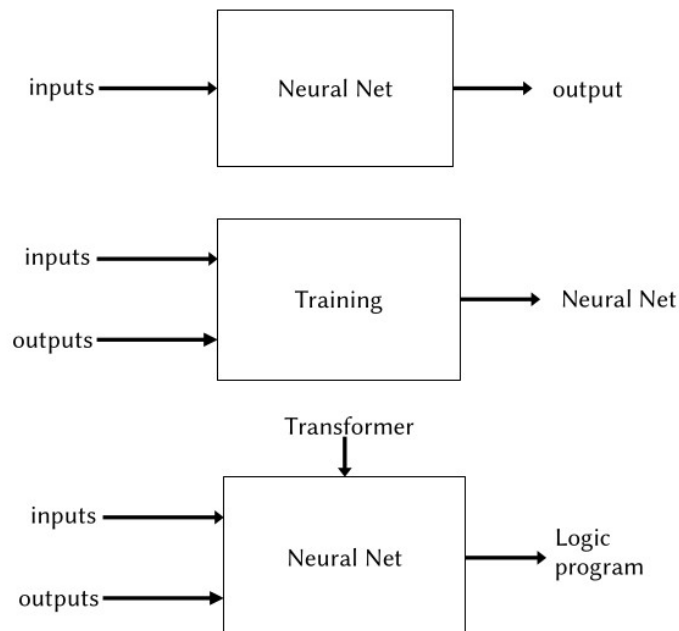


Figure 3: (top) A neural network produces outputs given some inputs. (middle) The neural network is itself produced by a training process on many pairs of inputs and outputs. (bottom) A

language model trained on logic programs could be used to reproduce them by appropriate prompting.

We envisage the possibility depicted at the bottom of Figure 3. An LLM, which is a specialized form of neural network coupled with a processing device (e. g. a transformer). Having being trained to represent logic programs, the transformer could be prompted to reproduce correct sources by providing it with some samples. It would be *de-compiling* logic programs from the sub-symbolic representation. In each case, we aim to extract a logic program whose inherent structure and rules represent meaningful, correct, and independently verifiable knowledge about the domain, irrespective of the neural network's specific input-output mappings.

3.2. Neural Networks and Logic

Let us complete this review of basic principles by looking at the relationship between neural networks and logic programs. It is no longer as simple as it will be presented below, because the technology of the connectionist systems, as they are also called, has evolved. Transformers[2], in particular, operates in a very different manner. But the essence of a neural network as an array of weights still holds.

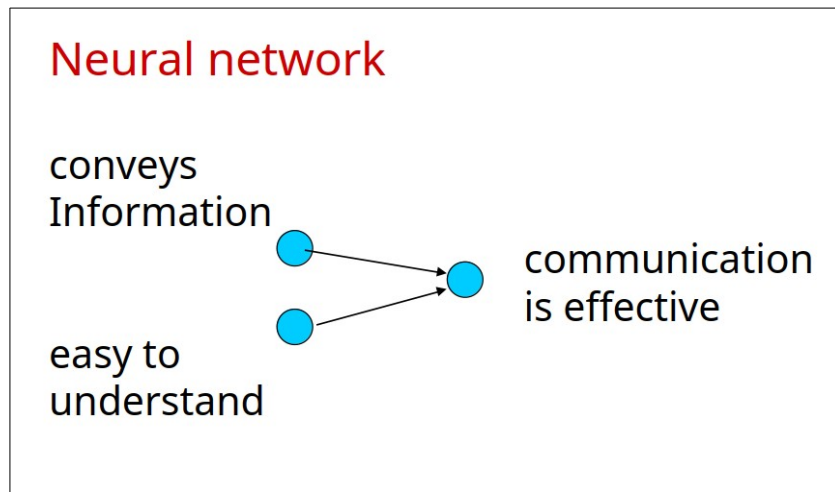


Figure 4: A very simple neural network with just two layers (no hidden layer), two inputs and one output. (From Kowalski [16]).

The neural network shown in figure 4 is very simple and has a direct correspondence with a rule in logic:

*A communicative act is effective with a strength W
if the act conveys information with a strength $W1$
and the act is easy to understand with a strength $W2$
and $W = f(W1, W2)$.*

Which can be rendered in even simpler human, natural language, as:

A communicative act is more effective, the more it conveys information and the easier it is to understand.

This goes to show that neural networks can have some human meaning associated with them. That connection is, of course, harder to elicit as the network grows. Moreover, in the Transformer

Architecture [1], that connection is mediated by a process computing a prediction for the next token in a sequence. One can say, however, that the success of this process logically depends on how well the connection between that token and its context to the previous ones is enshrined in the network or its input.

To first test that idea, consider the following collection of examples in table 1:

Table 1

Examples to train a neural network (From [16])

Author	Thread	Length	Location	Action
known	new	long	home	skip
unknown	new	short	work	read
unknown	follow-up	long	work	skip
known	new	short	home	read

This small set of examples could be used to train a network like the one shown in figure 5:

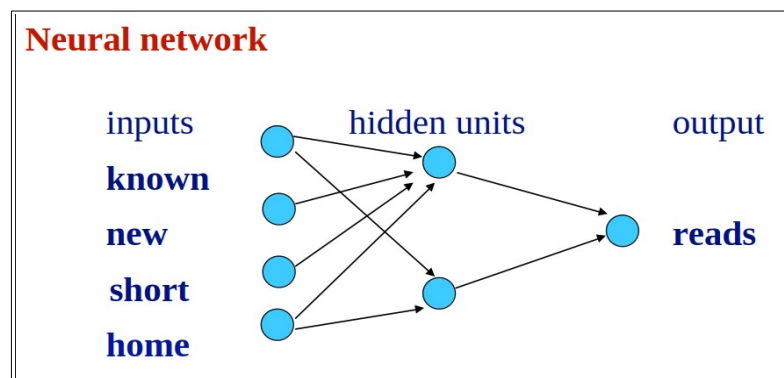


Figure 5: A less simple neural network with just three layers (no hidden layer), to decide whether to read or skip (From Kowalski [16]).

We posed the following prompt to Claude Sonnet 4:

Consider this set of examples that correspond, respectively, to Author, Thread, Length, Location, Action:
 known, new, long, home, skip
 unknown, new, short, work, read
 unknown, follow-up, long, work, skip
 known, new, short, home, read
 Write rules to decide when to read and when to skip. Write the rules as logical clauses like skip if known and new. Explain the rules.

and obtained this answer:

<Start Content enhancement by Claude Sonnet 4>¹

Looking at the examples, I can derive these rules:

¹Labeled as suggested here: <https://ceur-ws.org/GenAI/Taxonomy.html>

Rules:

1. **Skip if long** (regardless of other factors)
2. **Read if short** (regardless of other factors)

Explanation:

The key pattern in the data is that **length is the decisive factor**:

- All long emails are skipped (examples 1 and 3)
- All short emails are read (examples 2 and 4)

The other attributes (Author, Thread, Location) don't appear to influence the decision. For instance:

- Both known and unknown authors can be read or skipped
- Both new threads and follow-ups can be read or skipped
- Both home and work locations can be read or skipped

The decision rule is elegantly simple: **length determines action**. This suggests a practical email management strategy where you prioritize shorter emails (quick to process) over longer ones (time-consuming), regardless of who sent them, whether they're new conversations, or where you're reading them.

This approach maximizes email processing efficiency by tackling quick wins first, which is a common productivity strategy.

<End Content enhancement by Claude Sonnet 4>

This exercise illustrates how an LLM can take an input with a set of data and some instructions about how to produce an output and, not only carrying on the instructions regarding the data, but producing useful rules to model it. It even goes on to also hypothesize an explanation. At this point, one could test the rules to check on their validity independently from the LLM. We end up with a hybrid system that is very effective at parsing natural language and can be fully accounted for while reasoning.

3.3. Small Language Models

Examples as the one in the previous section are, of course, encouraging but tend to drive the hype around Generative AI. Users develop a form of trust not just on the parsing abilities of LLMs but on their reasoning capabilities, which can be justified by instances such as that in which they appear to be reasoning correctly.

To help us recover perspective, we have developed a very simple example of a language model on one of software platform used for the large ones [17]. The code is shown in Appendix A. We have trained this little model with the following three sequences of proofs in Prolog notation, to indicate how those elementary rules and facts could be used to deduce another fact in a very straightforward exercise of forward reasoning.

```
patterns = [  
    "a.b:-a.b.",  
    "b.c:-b.c.",  
    "a.b.d:-a,b.d."  
]
```

To test the model, we provide strings of characters and expect to recover the following character on one of the training patterns. These are the outputs:

Input: 'a.' -> Predicted: '<eos>' (Expected: 'b')

```

Input: 'a.b' -> Predicted: '.' (Expected: ':')
Input: 'a.b:' -> Predicted: '-' (Expected: '-:')
Input: 'a.b:-' -> Predicted: 'a' (Expected: 'a:')
Input: 'a.b:-a' -> Predicted: '<eos>' (Expected: ':')
Input: 'a.b:-a.' -> Predicted: '<eos>' (Expected: 'b')
Input: 'a.b:-a.b' -> Predicted: '.' (Expected: ':')
Input: 'b.c:-b.' -> Predicted: '<eos>' (Expected: 'c')
Input: 'a.b.d:-a,b.' -> Predicted: 'd' (Expected: 'd')

```

This tiny model correctly predicts a **correct** continuation of 5 out of 9 cases (as marked in **yellow** and **green** highlighting). **One** of them (last line in **red**) could be (mis)interpreted as the correct application of forward reasoning. That output lists what comes from the model (the prediction) and what one can expect given the inputs. For example, in the last line, we expected *d* because of the input pattern: "*a.b.d:-a,b.d*". And the model correctly predicts that *d* follows *a.b.d:-a,b*. That prediction also coincides with the results of inferring *d* from *a* and *b* through the rule *d:-a,b*, a coincidence that could have people believing that the model does *modus ponens*.

This experiment, of course, has no statistical significance. But it does suggest that Transformers' language models, while simply predicting the next token, could be seen as doing some form of reasoning. That they can produce outputs as elaborated as the one shown in the previous section, just by scaling, is, of course, extraordinary and requires further investigations.

4. De-compiling an LLM into Logic Programs

We have performed some more practical experiments to generate logic programs from LLMs.

In a first experiment, we [18] prompted Google's LLM, Gemini (1.5), with a set of instructions to list the most common ways to ask the same question and then map the corresponding alternatives as new entries to a Prolog dictionary² (.ibid). That dictionary is used by the control natural language Logical English, LE. We ended with a series of descriptions of relations in natural language, called *templates* in LE, that can be used to query a document processed by the regular LE engine in Prolog.

In a recent experiment, we prompted an LLM (DeepSeek V3) to produce alternative texts to extend and improved a logic program written in Logical English. We started with the prompt:

This is a set of rules, an scenario for the rules and a list of queries:

and then included the content of a Logical English document describing an Australian Tax Law³, but without including templates. Then, we asked:

Can you produce some interesting scenarios to test those rules

and it produced a list of 6 scenarios to test the given rules, described in plain English but not in Logical English. Following the same chat thread, we asked:

Can you restate those scenarios using these relations:

and included all the templates that accompany that document. The answer is a partial but computable list of examples that could be added to the document in question, such as this one:

<Start Content enhancement by DeepSeek V3>

²https://github.com/LogicalContracts/LogicalEnglish/blob/main/kb/4_affiliates_3.pl

https://github.com/LogicalContracts/LogicalEnglish/blob/8ee7ff2740cb1e7231d115572a5a3dad7c673535/le_input.pl#L2478 (where the particular entries start)

³https://github.com/LogicalContracts/LogicalEnglish/blob/main/moreExamples/1_cgt_assets_and_exemptions_3.le

Scenario 1: The Crypto Collector

Facts:

- Alice is a taxpayer.
- asset bitcoin belongs to Alice.
- asset bitcoin was acquired at 2020-01-01.
- asset bitcoin is used for the purchase of items for personal consumption.
- asset nft_rare belongs to Alice.
- asset nft_rare costed 15000 to acquire.
- asset nft_gift belongs to Alice.
- asset nft_gift was gifted through a will to a deductible gift recipient beneficiary.

Queries:

1. Is asset bitcoin a CGT exempt asset?
2. Is asset nft_rare a CGT exempt asset?
3. Is asset nft_gift a CGT exempt asset?

<End Content enhancement by DeepSeek V3>

Some extra processing is still required, but the content is already in the correct syntax so that it could be added to a runnable document in Logical English, with the following content⁴

```
scenario Crypto is:
  Alice is a taxpayer.
  asset bitcoin is a cryptocurrency.
  asset bitcoin belongs to Alice.
  asset bitcoin was acquired at 2020-01-01.
  asset bitcoin is used for the purchase of items for personal consumption.
  asset nft_rare belongs to Alice.
  asset nft_rare costed 15000 to acquire.
  asset nft_gift is a gift.
  asset nft_gift belongs to Alice.
  asset nft_gift was gifted through a will to a deductible gift recipient
  beneficiary.
```

The produced facts approximately follow the logic in the document and cannot be trusted to cover it all. Some adjustments and extra information are still required to make it work. But there seems no need to use a particular syntax, like Prolog's, and the generated outputs are very useful to extend the codification of testing cases. It can be argued that the difficult part is to generate the rules. Some preliminary results show that it is also feasible with few shots prompting [19].

5. Conclusion

We have made an argument for the use of LLM to generate logic programs. We have shown a few experiments in which proper prompting of an LLM produces fragments of logic programs and queries with immediate utility. The read/skip example produces a small logic program, a previous paper shows how we could generate the queries (in Prolog) from English and the last example is about producing facts and queries for scenarios that test the rules in a given document. This has led us to believe that we could exploit the extraordinary capacity of Transformers for natural language parsing, without having to rely on them for reasoning. By de-compiling logic programs from LLMs, we could scale the production of systems integrating generative AI + Logical AI, which could provide a flexible interaction with humans in natural language while offering well-verified

⁴https://github.com/LogicalContracts/LogicalEnglish/blob/main/moreExamples/cgt_assets.le

capabilities for reasoning. We aim to extract knowledge from LLMs into logic programs and queries and, therefore, add LLMs to the **materials and tools** used to teach logic programming.

6. Future work

We envisage a road map to develop these ideas, bridging LLMs and symbolic logic together, and emphasizing the goal of extracting *independently meaningful and correct* logic. First, to establish a robust communication interface between LLMs and Prolog. Second, to define and implement methods to evaluate the *semantic correctness* of extracted logic, independent of the LLM's original output, which will probably bring us back to the discussion about the semantics of LP and will include developing an iterative agentic toolchain where the LLM proposes and refines Prolog rules based on feedback from the Prolog interpreter, and create benchmarks for evaluating this deep logic extraction. And, third, to explore the scalability of the approach and identify real-world applications for independently correct logic extraction. Some related research questions that we could discuss at the workshop are: **What constitutes "independently meaningful and correct logic"?** (A precise definition is key); **What to do if the LLM "hallucinates" plausible-looking but incorrect logic?** (The role of the Prolog interpreter as a strict validator could be crucial); **What are the theoretical limits of extracting symbolic logic from sub-symbolic models?**; **What specific architectures for LLMs would best support logic extraction or LP de-compilation?** (e.g., models explicitly trained on symbolic reasoning tasks, or those with more transparent internal representations); and, **How can we scale this beyond toy examples to real-world complexity?** (Modularization, hierarchical reasoning, combining with other AI paradigms).

Acknowledgements

We are grateful to LodgeIT for their encouragement and support to encode Australian Tax Law in Logical English.

Declaration on Generative AI

During the preparation of this work, the author used Claude Sonnet 4 for *Content enhancement* to produce the example of guided-rule generation, in section 3.2. We also used DeepSeek V3 for *Content enhancement* to improve Logical English documents, as described in section 4. After using these tools/services, the author reviewed and edited the content as needed and takes full responsibility for the publication's content.

References

- [1] Aarohi Srivastava and more than 400 contributors⁵. "Beyond the imitation game: Quantifying and extrapolating the capabilities of language models." *arXiv preprint arXiv:2206.04615* (2022).
- [2] Vaswani, Ashish, *et al.* "Attention is all you need." *Advances in neural information processing systems* 30 (2017).
- [3] Emily M. Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. 2021. On the Dangers of Stochastic Parrots: Can Language Models Be Too Big? 🦜. In Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency (FAccT '21). Association for Computing Machinery, New York, NY, USA, 610–623. <https://doi.org/10.1145/3442188.3445922>
- [4] Mohammadamin Barektain, Anant Nawalgaria, Daniel J. Mankowitz, Majd Al Merey, Yaniv Leviathan, Massimo Mascarò, Matan Kalman, Elena Buchatskaya, Aliaksei Severyn, Irina

⁵<https://github.com/google/BIG-bench/graphs/contributors> <https://arxiv.org/abs/2206.04615>

- Sigler, and Antonio Gulli. Foundational Large Language Models & Text Generation. Google (2025) <https://www.kaggle.com/whitepaper-foundational-llm-and-text-generation>
- [5] Xu, Ziwei, Sanjay Jain, and Mohan Kankanhalli. "Hallucination is inevitable: An innate limitation of large language models." *arXiv preprint arXiv:2401.11817* (2024).
 - [6] Xiaocheng Yang, Bingsen Chen, and Yik-Cheung Tam. 2024. Arithmetic Reasoning with LLM: Prolog Generation & Permutation. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 2: Short Papers)*, pages 699–710, Mexico City, Mexico. Association for Computational Linguistics.
 - [7] Borazjanizadeh, N., & Piantadosi, S. T. (2024). Reliable reasoning beyond natural language. *arXiv preprint arXiv:2407.11373*.
 - [8] Tan, X., Deng, Y., Qiu, X., Xu, W., Qu, C., Chu, W., Xu, Y & Qi, Y. (2024). THOUGHT-LIKE-PRO: Enhancing Reasoning of Large Language Models through Self-Driven Prolog-based Chain-of-Thought. *arXiv preprint arXiv:2407.14562*.
 - [9] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., & Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35, 24824-24837.
 - [10] Tarau, P. (2025, January). Leveraging LLM Reasoning with Dual Horn Programs. In *International Symposium on Practical Aspects of Declarative Languages* (pp. 163-178). Cham: Springer Nature Switzerland.
 - [11] Tarau, P. (2025). On LLM-generated Logic Programs and their Inference Execution Methods. *arXiv preprint arXiv:2502.09209*.
 - [12] Thuy, P. T. T., & Yamamoto, A. (2024). Implementing Derivations of Definite Logic Programs with Self-Attention Networks. *arXiv preprint arXiv:2410.11396*.
 - [13] Liu, H., Fu, Z., Ding, M., Ning, R., Zhang, C., Liu, X., & Zhang, Y. (2025). Logical Reasoning in Large Language Models: A Survey. *arXiv preprint arXiv:2502.09100*.
 - [14] Shojaei, P., Mirzadeh, I., Alizadeh, K., Horton, M., Bengio, S., & Farajtabar, M. (2025). The illusion of thinking: Understanding the strengths and limitations of reasoning models via the lens of problem complexity. *arXiv preprint arXiv:2506.06941*.
 - [15] Aho, Alfred V., Lam, Monica S., Sethi, Ravi, & Ullman, Jeffrey E. (2007) *Compilers: Principles, Techniques and Tools*. Second Edition.
 - [16] Kowalski, B. (2011). *Computational Logic for Human Communication*. Imperial College. London, UK.
 - [17] Dávila, J. (2025). Decompiling LM into LP. Kaggle Notebook. <https://www.kaggle.com/code/jacintodavila/decompiling-lm-into-lp>
 - [18] Dávila, J. Controlled Natural Language Models. Prolog Education Workshop 2024, Workshop Proceedings of the 40th International Conference on Logic Programming (ICLP-WS 2024). <https://ceur-ws.org/Vol-3799/paper10PEG2.0.pdf>
 - [19] Zin, May Myo; Borges, Georg; Satoh, Ken; FUNGWACHARAKORN, Wachara (2025). Towards Machine-Readable Traffic Laws: Formalizing Traffic Rules into PROLOG Using LLMs. The 20th International Conference on Artificial Intelligence and Law, ICAIL 2025. Chicago. June 16 to 20, 2025

A. Appendix

```
import torch
import torch.nn as nn
import torch.optim as optim
from collections import defaultdict

# Special tokens
PAD_ID = 0
```

```

BOS_ID = 1
EOS_ID = 2

class PatternTransformer(nn.Module):
    def __init__(self, vocab_size, embed_size=16, nhead=1):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, embed_size)
        self.attention = nn.MultiheadAttention(embed_size, nhead, batch_first=True)
        self.out = nn.Linear(embed_size, vocab_size)

    def forward(self, x):
        x = self.embed(x)
        seq_len = x.size(1)
        attn_mask = torch.triu(torch.ones(seq_len, seq_len), diagonal=1).bool()
        x, _ = self.attention(x, x, x, attn_mask=attn_mask.to(x.device))
        return self.out(x)

# Create vocabulary from multiple patterns
patterns = [
    "a.b:-a.b.",
    "b.c:-b.c.",
    "a.b.d:-a,b.d."
]

# Build vocabulary
vocab = defaultdict(lambda: len(vocab))
vocab['<pad>'] = PAD_ID
vocab['<bos>'] = BOS_ID
vocab['<eos>'] = EOS_ID
#print(vocab)

# Pre-process patterns
def preprocess(text):
    return [vocab[ch] for ch in text] + [EOS_ID]

pattern_ids = [preprocess(p) for p in patterns]
#print(pattern_ids)
# Create training samples
def create_samples():
    samples = []
    for seq in pattern_ids:
        for i in range(2, len(seq)): # Need at least 2 tokens for prediction
            input_seq = [BOS_ID] + seq[:i]
            target = seq[i]
            samples.append((torch.tensor(input_seq), torch.tensor(target)))
    return samples

# Initialize model
model = PatternTransformer(len(vocab))
opt = optim.Adam(model.parameters(), lr=0.01)
criterion = nn.CrossEntropyLoss()

# Training
samples = create_samples()
print(samples)
for epoch in range(500):
    total_loss = 0
    for input_seq, target in samples:
        opt.zero_grad()
        logits = model(input_seq.unsqueeze(0))
        loss = criterion(logits[0,-1:], target)
        loss.backward()
        opt.step()
        total_loss += loss.item()
    if epoch % 50 == 0:
        print(f"Epoch {epoch}, Loss: {total_loss/len(samples):.4f}")

# Testing
test_cases = [
    ("a.", "b"),
    ("a.b", ";"),
    ("a.b:", "-"),
    ("a.b:-", "a"),
    ("a.b:-a", "."),
    ("a.b:-a.", "b"),
    ("a.b:-a.b", "c"),
    ("b.c:-b.", "c"),
    ("a.b.d:-a,b.", "d")
]

print("\nTesting:")
for test_input, expected in test_cases:
    input_ids = torch.tensor([BOS_ID] + [vocab[ch] for ch in test_input])
    with torch.no_grad():
        logits = model(input_ids.unsqueeze(0))
        #print(logits)
        pred = logits.argmax(-1)[0,-1].item()
        print(f"Input: '{test_input}' -> Predicted: '{list(vocab.keys())[pred]}' (Expected: '{expected}')")

```