RESEARCH INSTITUTE FOR
SYMBOLIC COMPUTATION

JOHANNES KEPLER
UNIVERSITY LINZ

Building a Logical Agent with LangChain ... and Quite Some Vibe Coding

Wolfgang Schreiner

March 2026

RISC Report Series No. 26-02

ISSN: 2791-4267 (online)

Available at `https://doi.org/10.35011/risc.26-02`

JOHANNES KEPLER
UNIVERSITY LINZ
Altenberger Str. 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Building a Logical Agent with LangChain
. . . and Quite Some Vibe Coding

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University Linz, Austria
Wolfgang.Schreiner@risc.jku.at

March 19, 2026

Abstract

This document reports on our experience of building an “agentic AI” (Artificial Intelligence) that helps a human to answer logical questions in a trustworthy way. This agent combines a Large Language Model (LLM) (which interacts with the human in natural language) with a logical software (which automatically proves formal theorems). The LLM engages in a dialogue with the human in order to translate their logical question from natural language to a formal proof problem. Once the human is satisfied with the formalization, the LLM invokes the prover to automatically solve the problem and thus answer the question; then the LLM also offers the user the possibility to inspect the successful proof or the unsuccessful proof attempt by calling the prover in an interactive mode. Furthermore, we describe how much of the source code (which is based on on the agent construction framework LangChain) has been “vibe coded”, i.e., itself generated with the help of an LLM.

Contents

1 Introduction

More and more, Large Language Models (LLMs) [2] are applied as core components in software systems that interact with humans in natural language and solve for them complex problems by orchestrating the execution of various tools. A recent example of this "agentic AI" is Aristotle [3, 1], a mathematical system that has achieved gold-medal-equivalent performance on problems of the International Mathematical Olympiad by the combination of an LLM and the proof assistant Lean 4 [9, 7]. The LLM translates mathematical problems expressed in natural language into formal proof problems in the language of Lean and generates proof attempts in the form of Lean scripts. It then employs Lean to formally check the correctness of the proofs; if a check fails, the LLM revises its proof attempt and repeats the check.

In this document, we report on our own experience to develop a "logical agent" that helps a human to answer logical questions in a trustworthy way, i.e., with a strong guarantee about the correctness of the answer. This agent employs some state-of-the-art LLMs (we use the commercial LLMs OpenAI GPT 5.2, Google Gemini 2.5 Pro, and Anthropic Claude Sonnet 4) and our own logical software, the RISCTP theorem proving interface [11, 13]; we have previously developed RISCTP to complement the model checking capabilities of the formal specification and verification system RISCAL [10, 12] with theorem proving capabilities. Our approach differs from that of Aristotle in that we apply the LLM only to generate the formal problem and then let RISCTP automatically generate the corresponding proof (while Aristotle lets the LLM also generate the proof and applies Lean only to check the proof).

However, our goal was not the development of a powerful agentic AI competitive with Aristotle. Rather we wanted to investigate the fundamental concepts and technologies required to couple an LLM with an external logical tool with minimal effort. For this purpose, we applied the ReAct principle [16] that is based on and generalizes the prompting pattern "Chain of Thought" [14]. Specifically, ReAct extends the capabilities of LLMs for reasoning[1] by action-taking: the output of an LLM may be the request to call some external tool whose output is in turn added to the conversation history for further processing by the LLM.

More concretely, we applied the implementation of ReAct by LangChain [8, 4], a Python-based software framework that helps to facilitate the integration of LLMs in applications. We deliberately stuck to the ReAct workflow pre-built in LangChain, not considering any more complex execution or prompting patterns such as "Tree of Thoughts" [15]. As a side quest, we also investigated the use of LLMs (specifically Google Gemini 3) for generating the code of our logical agent; in fact, most of the software was "vibe coded"[2] in this way.

The rest of this document is structured as follows: Section 2 sketches the basics of the ReAct pattern and its practical use in LangChain. Section 3 describes the architecture of the logical agent and the elements it consists of. Section 4 demonstrates the actual use of the software. Section 5 concludes with an outline of open problems and future work.

The bulk of this document actually consists of its appendices: Appendix A shows sample sessions with the logical agent using the different LLMs. Appendix B gives the prompts that were used to generate various source files or source templates as the basis for manual adaptation.

---

[1] Here the term "reasoning" means the language-based capabilities of LLM to mimic human reasoning; this is quite different from the reasoning of logical software which is based on provably sound formal calculi.

[2] https://en.wikipedia.org/wiki/Vibe_coding

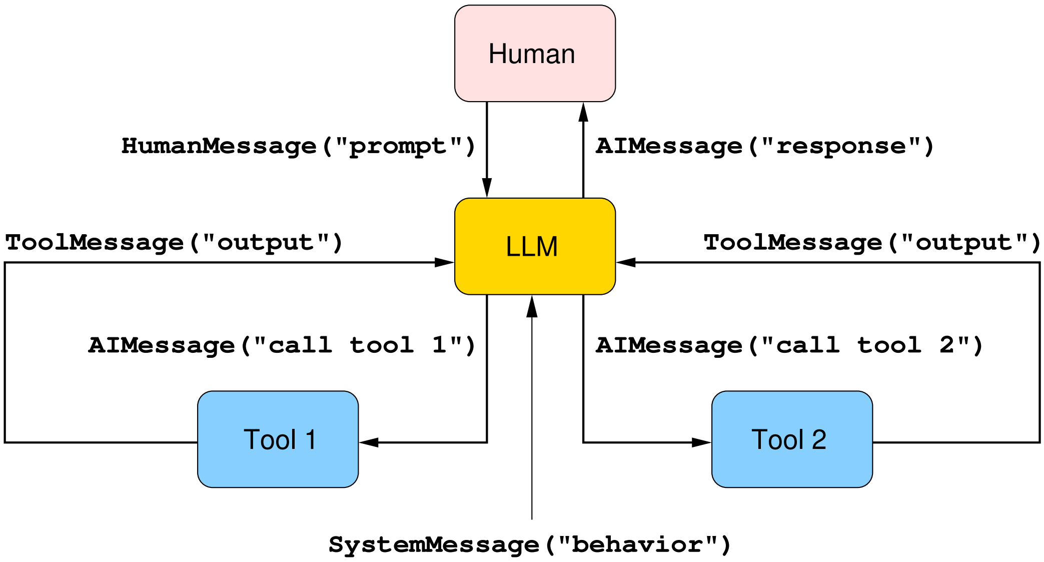


Figure 1: The ReAct Paradigm

Appendix C lists the complete source code of the agent which can be also downloaded from:

`https://www.risc.jku.at/people/schreine/agent.tgz`

2 ReAct and LangChain

On the fundamental level, every LLM is an engine that continues text (actually a sequence of "tokens" where each token can be interpreted as a word or a part of a word). Given a text from a human as input ("the prompt"), the LLM produces a continuation of the text (the "response"). A dialogue between human and LLM emerges, if the resulting text is extended by another prompt of the human which in turn is extended by another response of the LLM. Each input of the LLM thus contains the full history of the conversation so far, but truncated or summarized to some maximum size, the "context length" of the LLM; thus the LLM has only a limited "memory capacity" and may also "forget" some of the conversation. The LLM has been previously trained with a huge number of conversations from which it has learned to interact with humans in a way that we may judge as "intelligent".

On a more abstract level, this behavior can be described as the recurring interaction of two agents, the human and the LLM, exchanging messages (see the pink and the golden node in Figure 1). The human sends a `HumanMessage(“`*`prompt`*`”)` to the LLM which sends back an `AIMessage(“`*`response`*`”)`; this cycle of message exchanges repeats until the human ends the conversation. However, this description is just a convenient abstraction of the underlying reality: actually, the conversation between human and LLM is driven by a software, the "agent framework" that embeds the LLM, keeps track of the history of messages, and feeds it appropriately into the LLM. This framework also sets up the context of the conversation by sending an initial

`SystemMessage(“behavior”)` to the LLM that describes its expected behavior, i.e., how the LLM should interact with the human.

The ReAct principle [16] extends this picture by giving the LLM access to a set of “tools” (the blue nodes in Figure 1). These tools are programs that may perform some algorithmic computations, access external information resources (e.g., databases, calendars, or the world wide web), or also produce effects in the real world (e.g., send emails, book services, or make purchases). Given a message from the human, the LLM needs not immediately respond, if it “feels” not capable to do so, due to a lack of information or a lack of “confidence” in its own abilities. In such a situation, the LLM may rather generate one or more “tool calls”, i.e., requests to invoke some tools with specific inputs. The agent framework interprets such a call, invokes the corresponding tool with the arguments, and encapsulates its answer into a `ToolMessage(“output”)` that it forwards to the LLM (i.e., that it appends to the conversation history fed into the LLM). This behavior may be repeated until the LLM is confident enough to send a response to the human.

The development of such an agent framework is substantially simplified by utilizing libraries that have been especially developed for this purpose. A prominent example is the Python library LangChain [8, 4] which we have employed for building our “logical agent”. The following example demonstrates a simple agent built on top of LangChain:

```
from langchain_core.messages import SystemMessage, HumanMessage
from langchain_core.tools import tool
from langchain_openai import ChatOpenAI
from langchain.agents import create_agent

# to use this agent, set the following environment variable:
# export OPENAI_API_KEY=’your-key-here’

@tool
def add(a:str, b:str) -> str:
    """
    Adds two integer numbers provided as strings and returns the sum as a string.
    """
    try:
        result = str(int(a) + int(b))
        print(’add(’ + a + ’,’ + b + ’) = ’ + result)
        return result
    except ValueError:
        return "Error: Please provide valid integer strings."

llm = ChatOpenAI(model=’gpt-5.2’, temperature=0)
tools = [add]
agent = create_agent(llm, tools)

behavior = ’perform calculations only using external tools’
prompt = input(’User: ’).strip()
result = agent.invoke({"messages": [ SystemMessage(behavior), HumanMessage(prompt) ]})
output = result["messages"][-1].content
print(’Agent: ’ + output)
```

This Python program `addagent.py` implements an agent that combines the LLM GPT-5.2 of OpenAI with a tool (in the form of a Python function) that is able to (reliably) add two integer numbers (the tool prints out a message in order to demonstrate that the calculation was really performed by the tool, not by the LLM itself). The program then reads a prompt from the standard input and invokes the agent with two messages: first a system message that tells the LLM its expected behavior and then a human message that includes the prompt. The result of the agent is a complete history of messages (including the messages exchanged between LLM and tool); the program extracts the content of the last message (the response of the LLM) and prints it to the standard output.

By calling the program as `python3 addagent.py` we may have the following dialogue:

```
User: What is 235711 plus 314159?
add(235711,314159) = 549870
Agent: 235711 + 314159 = **549870**
```

Here the first line shows the input of the human user, the second line represents the output of the tool, and the last line gives the response of the LLM.

The numerical inputs and outputs may be even provided in natural language, as demonstrated by the following dialogue:

```
User: What is one hundred twelve and seventy one? (give the result in words)
add(112,71) = 183
Agent: One hundred eighty-three.
```

This agent only performs a single prompt-response cycle; therefore it copes without any memory of the previous conversation. However, it might also be easily equipped with such a memory:

```
from langgraph.checkpoint.memory import MemorySaver
agent = create_agent(llm, tools, checkpointer=MemorySaver())
```

With this extension, we might continue the execution of the program with an arbitrary number of invocations of the agent, using prompts that refer to the conversation history (we will make use of this in our implementation of the logical agent).

We have constructed the agent by applying the LangChain function `create_agent` which already implements the ReAct pattern as explained above. However, LangChain is much more powerful: it allows to describe complex workflows of interacting nodes some of which may be LLMs. For more details, see, e.g., [2, 4].

3 The Agent Architecture

Our basic idea for the logical agent has been as follows (see also Figure 2):

1. We define a formal language “FOL-PRE” (“first-order logic in prefix-notation”) for describing proof problems. We describe FOL-PRE in a document `FOL-PRE.txt`, including examples that explain how logical problems in natural language can be translated to FOL-PRE syntax and make this document accessible in the web.

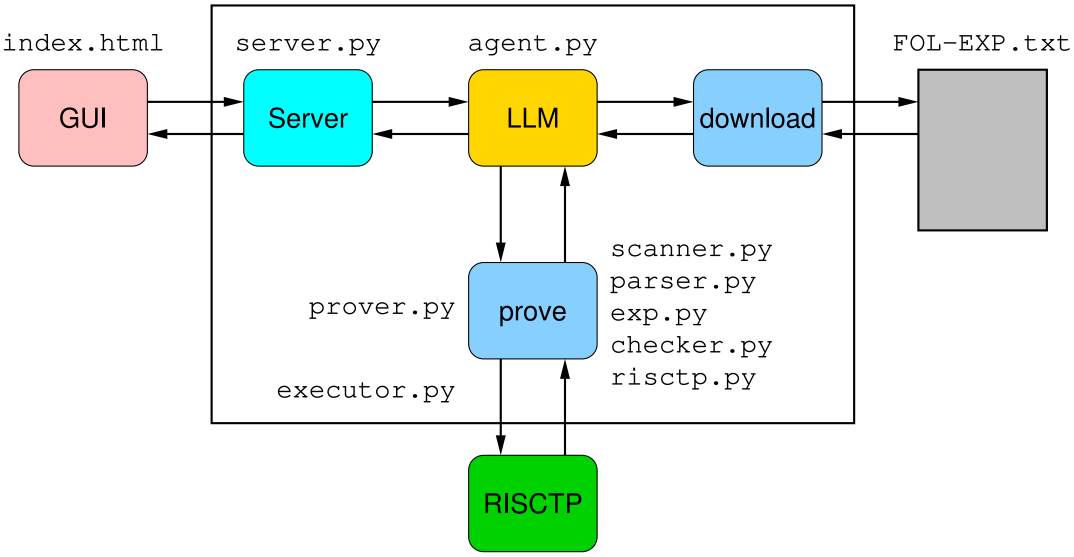


Figure 2: The Logical Agent

2. We write a tool “prove” (file `prover.py`) that accepts a proof problem in the FOL-PRE format, checks its syntactic correctness (`scanner.py`, `parser.py`, `exp.py`) and semantic consistency (`checker.py`), translates it into the language of the RISCTP theorem proving interface (`risctp.py`), and sends this translation to an external process that executes the RISCTP software (`executor.py`). The tool returns as a result the output of RISCTP, i.e., the confirmation of the construction of a successful proof or the report of the failure to find such a proof, respectively. The tool may be also called in an interactive mode where RISCTP opens a GUI in which the proof (attempt) may be inspected.
3. We tell the agent (`agent.py`) to respond to a logical question of the human by downloading (via an auxiliary “download” tool that we also provide) the file `FOL-PRE.txt` from the web and study its content to translate the question of the human to a formal proof problems in FOL-PRE notation. The agent then presents the translation to the human who may ask for corrections or additions. Once the human is satisfied with the formal description, the agent forwards it to the “prove” tool in its automatic mode in order to solve the problem; based on the outcome of the tool, the agent produces its answer to the question. The agent then also offers to invoke the prover in its interactive mode in order to allow the human to inspect the successful proof or the failed proof attempt, respectively.

The description of FOL-PRE is available at:

https://www.risc.jku.at/people/schreine/FOL-PRE.txt

`FOL-PRE.txt` is a document of about 1,100 lines that starts as follows:

```
The Language of Logic in the Notation FOL-PRE:
From Natural Language to Formal Expressions
```

```
==============================================

We describe the language of first-order logic (also called "first-order
predicate logic" or just "predicate logic"). This language allows to denote
values and to express statements about values. We describe this language both
formally and informally, namely by giving formal expressions which we
subsequently interpret in natural language. This also demonstrates how we can
conversely translate statements in natural language to formal expressions.

We give the formal expressions in a variant of "prefix notation" that we call
"FOL-PRE". Whenever we refer in the following to "formal expressions", we mean
"FOL-PRE" expressions.

As an example, the formal expression

  forall["x","A",imp[fun["p",var["x"]],exists["y","B",fun["q",var["x"],var["y"]]]]]

can be read as "for every value x of type A, if x satisfies property p, then
there exists some value y of type B, such that x is in relation q to y".
Likewise, the natural language statement "everybody loves somebody" can be
written as the formal expression

  forall["x","Person",exists["y","Person",fun["loves",var["x"],var["y"]]]]

which can be read in more detail as "for every person x, there exists some
person y, such that x loves y".
...
```

We deliberately used a new notation for describing proof problems, rather than using an existing notation such as SMT-LIB [5] or the language of RISCTP itself, because we wanted to investigate how well the LLM interprets descriptions of novel languages that it has not been previously trained with (although clearly the LLM has seen many examples of first-order logic in numerous similar, but not identical, notations).

The “download” tool is specified by the following docstring:

```
@tool
def download(url: str) -> str:
    """Download resource from the denoted URL and return its content."""
...
```

The “prove” tool is more comprehensively specified as follows:

```
def prove(fol_text: str, interactive: bool=False) -> tuple[bool,str]:
    """
    This function applies a prover to a proof problem, either interactively or
    automatically (depending on the value of the second parameter). The first
    parameter is the text of the proof problem in the FOL-PRE syntax
    described at https://www.risc.jku.at/people/schreine/FOL-PRE.txt.
    In the automatic mode, the function returns ’True’ if and if the the proof
    could be completed successfully, together with the output produced by the
```

```
    prover. In the interactive mode, the function always returns False and
    the empty string (the function blocks until the session has terminated).
    """
...
```

The overall behavior of the agent is controlled by this initial system prompt

```
You are a helpful assistant that answers logical questions in the following way
(inform the user before every step what you are going to do):
0. You download the description of the FOL-PRE syntax from
   https://www.risc.jku.at/people/schreine/FOL-PRE.txt and read it carefully.
1. You translate the question into a proof problem in the FOL-PRE syntax
   (make sure it is really in this syntax, do not just guess).
2. You ask the user whether this translation is correct.
3. If the translation is not correct, you follow the suggestions of the user
   to come up with a correct translation; then go back to step 2.
4. Once the translation is correct, you call an automatic prover with the proof
   problem and present its result (state whether the proof attempt was successful
   and show the full output produced by the prover).
5. Finally you also offer to to call an interactive prover on the problem
   for investigating the successful proof or the unsuccessful proof attempt.
   Call the interactive prover with the same translation that you used
   for the automatic prover.
Do not try to answer a logical question in any other way.
```

This formulation has been refined over several iterations to ensure that the LLM

- does not try to answer the logical question on its own,
- indeed performs the translation of the question into a formal proof problem according to the definition of the FOL-PRE syntax,
- approves of the translation by the user before invoking the prover,
- invokes the prover in the interactive mode with the same formalization that it also applied in the automatic mode.

Even with those caveats, we experienced variously deviations of the requested behavior; in particular in some instances the LLM ignored the command “go back to step 2” (i.e., to ask the prover for an approval after a modification of the translation).

We can use the agent from the command line or, more conveniently, via a web-based graphical user interface (`index.html`). For this, we have to start a server process (`server.py`) which embeds the agent as a web service; we will demonstrate this in the following section.

4 Using the Agent

The logical agent embeds one of the commercial LLMs OpenAI GPT 5.2, Google Gemini 2.5 Pro, or Anthropic Claude Sonnet 4 (other LLMs would work as well). To use one of these LLMs, the user has to require a corresponding API key from the vendor and set one of these environment variables:

```
export OPENAI_API_KEY='your-key-here'
export GOOGLE_API_KEY='your-key-here'
export ANTHROPIC_API_KEY='your-key-here'
```

The agent is chosen from a command line argument (see `agent.py`) or the value of a configuration variable (see `server.py`); the default is OpenAI GPT 5.2.

The agent can be called from the command line upon which the user may repeatedly enter queries (as single lines) to which the LLM responds until an empty line is entered:

```
> python3 agent.py
--- Agent Active (Model: gpt) ---
Enter your query. Enter an empty line to exit. Type 'reset' to clear history.
Does A imply A?
download https://www.risc.jku.at/people/schreine/FOL-PRE.txt
Assistant: I will now translate your question (“Does A imply A?”) into a
**proof problem in FOL-PRE syntax**.
...
```

However, it is more convenient start a server process that embeds the agent into a web service:

```
> python3 server.py
 * Serving Flask app 'server'
...
```

This service may be accessed by loading the file `index.html` into a web browser:

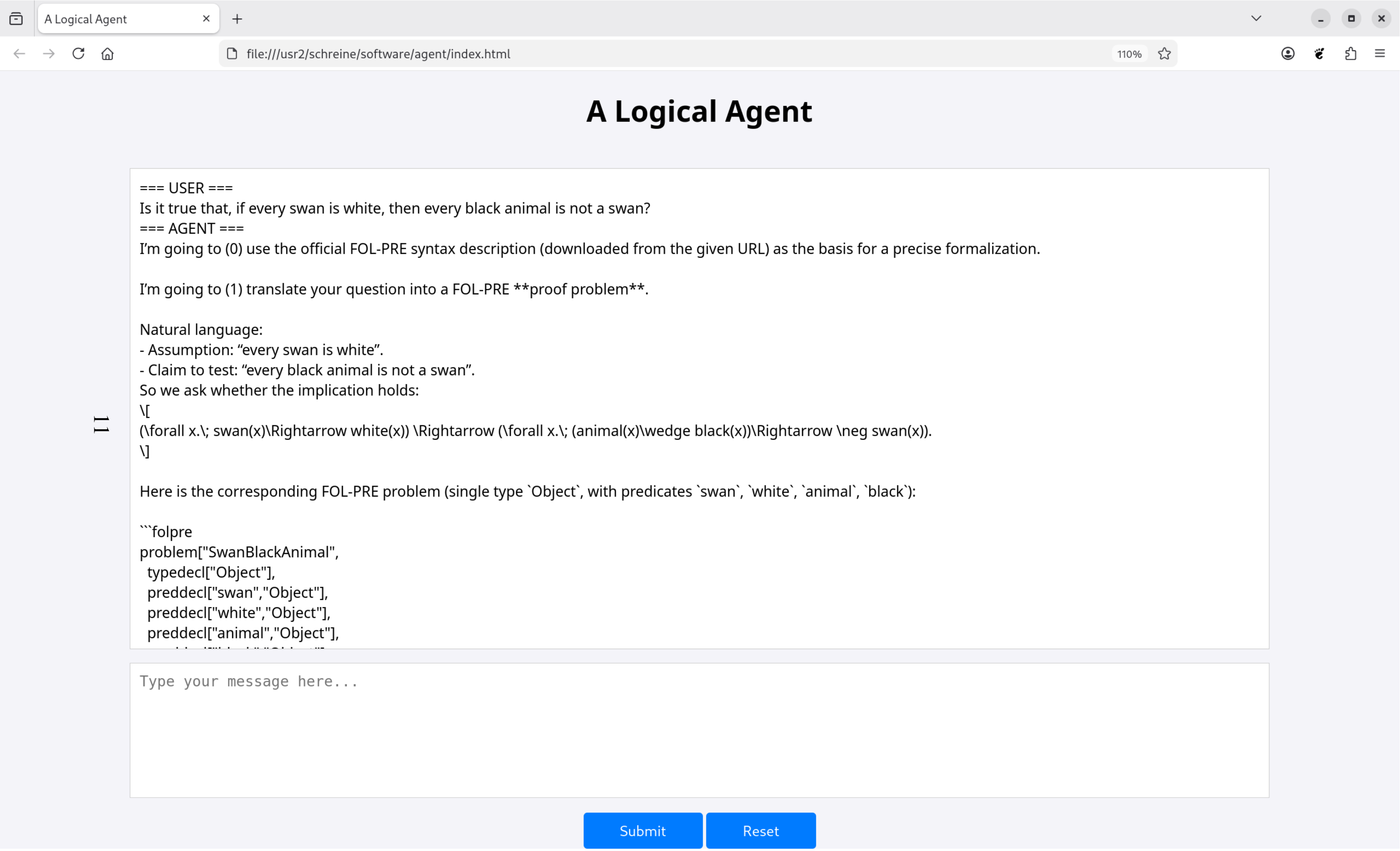
```
> firefox index.html
```

This brings up the graphical user interface depicted in Figure 3. Here the user may enter in the input area at the bottom multi-line input prompts to which, after pressing the button “Submit” the agent responds. The upper output area shows both user inputs and agent responses, each prefixed by the strings `=== USER ===` and `=== AGENT ===`, respectively. The agent remembers the full conversation history until the button “Reset” is pressed which resets the agent and its GUI to the initial state.

By interacting in a dialogue, user and agent may collaboratively engage in the formalization of a logical question which may be finally submitted to the prover in order to answer the question.

An example dialogue of this kind is given below:

```
=== USER ===
Is it true that, if every swan is white, then every black animal is not a swan?
=== AGENT ===
I'm going to (0) use the official FOL-PRE syntax description, then (1) translate
your question into a FOL-PRE proof problem, and (2) ask you to confirm the
translation.

## (1) Translation to a FOL-PRE proof problem

Interpretation:
```



A Logical Agent

~~~
=== USER ===
Is it true that, if every swan is white, then every black animal is not a swan?
=== AGENT ===
I’m going to (0) use the official FOL-PRE syntax description (downloaded from the given URL) as the basis for a precise formalization.

I’m going to (1) translate your question into a FOL-PRE **proof problem**.

Natural language:
- Assumption: “every swan is white”.
- Claim to test: “every black animal is not a swan”.
So we ask whether the implication holds:
\[
(\forall x.\; swan(x)\Rightarrow white(x)) \Rightarrow (\forall x.\; (animal(x)\wedge black(x))\Rightarrow \neg swan(x)).
\]

Here is the corresponding FOL-PRE problem (single type `Object`, with predicates `swan`, `white`, `animal`, `black`):

```folpre
problem["SwanBlackAnimal",
  typedecl["Object"],
  preddecl["swan","Object"],
  preddecl["white","Object"],
  preddecl["animal","Object"],
~~~

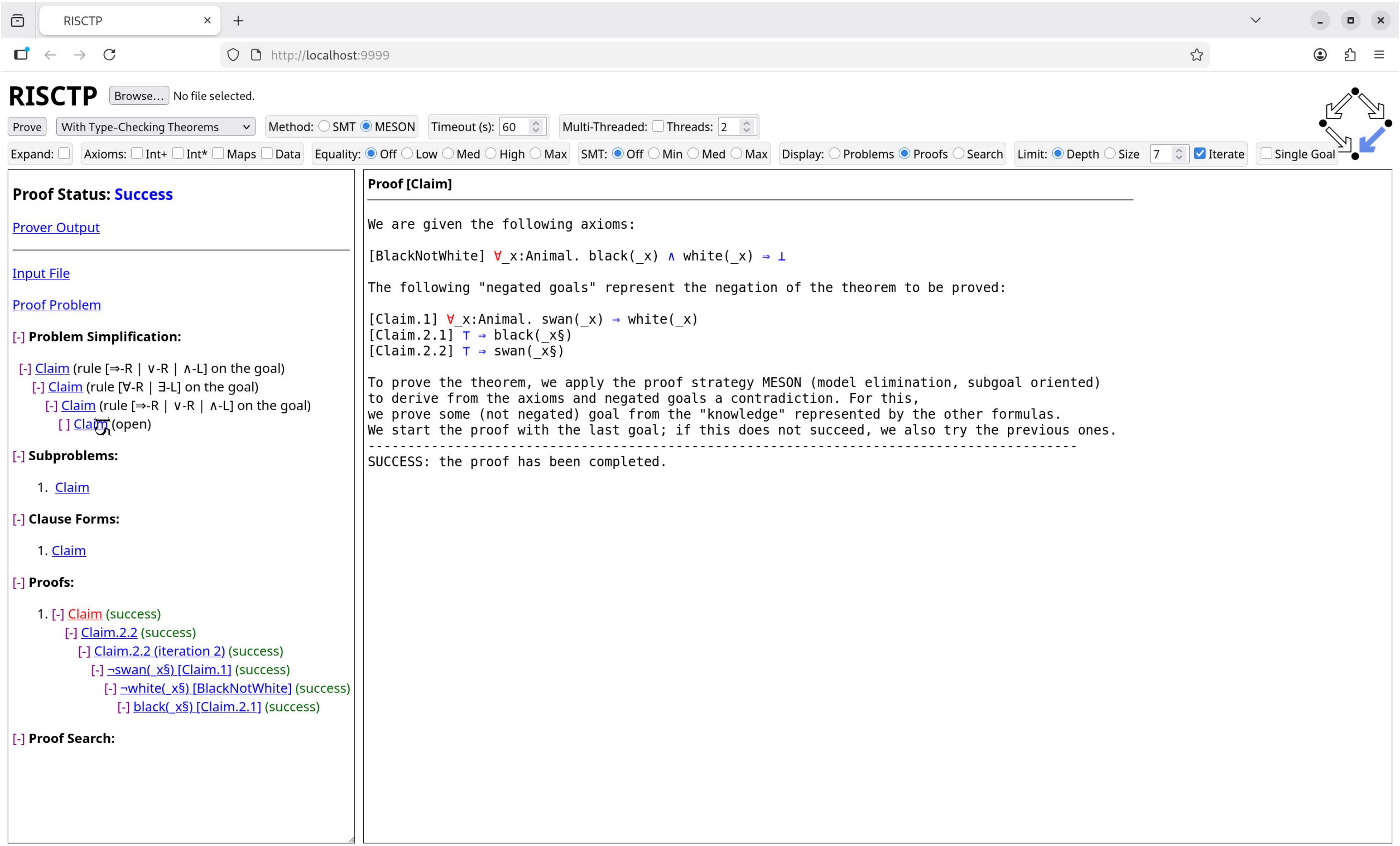
Figure 3: The Logical Agent GUI

```
- Domain/type: ‘Animal‘
- Predicates:
  - ‘swan(x)‘ := “x is a swan”
  - ‘white(x)‘ := “x is white”
  - ‘black(x)‘ := “x is black”
- Assumption: “every swan is white”  →  ∀x:Animal. swan(x) ⇒ white(x)
- Claim: “if every swan is white, then every black animal is not a swan”
  → (∀x swan(x)⇒white(x)) ⇒ (∀x black(x)⇒¬swan(x))

Here is the FOL-PRE problem:

‘‘‘folpre
problem["SwanBlack",
  typedecl["Animal"],
  preddecl["swan","Animal"],
  preddecl["white","Animal"],
  preddecl["black","Animal"],
  goal["Claim",
    imp[
      forall["x","Animal",
        imp[pred["swan",var["x"]],pred["white",var["x"]]]
      ],
      forall["x","Animal",
        imp[pred["black",var["x"]],not[pred["swan",var["x"]]]]
      ]
    ]
  ]
]
‘‘‘

## (2) Is this translation correct?

Two quick checks about intended meaning:
1. Is ‘black(x)‘ meant as “black (in color)”? (not “evil”, “illegal”, etc.)
2. Do you intend **no extra background assumption** like “nothing can be both
black and white”?

If the above matches your intended reading, tell me “yes, correct”. If not,
tell me what to change (e.g., add an axiom like ‘forall x. imp[pred["black",x],
not[pred["white",x]]]‘).
=== USER ===
Please show the translation in traditional syntax (use Unicode symbols,
not LaTeX).
=== AGENT ===
I’m going to (0) keep the same FOL-PRE-based formalization choices as before,
but (1) display the **translation in traditional first-order logic syntax**
using Unicode symbols, and then (2) ask you to confirm it.

## (1) Traditional (Unicode) syntax translation
```

```
Type/domain: ‘Animal‘
Predicates: ‘swan(x)‘, ‘white(x)‘, ‘black(x)‘

Goal formula:

(∀x ∈ Animal. (swan(x) → white(x))) → (∀x ∈ Animal. (black(x) → ¬swan(x)))

## (2) Is this translation correct?

Confirm that:
1) “black” just means ‘black(x)‘ (a unary predicate),
2) there is **no extra assumption** like ∀x(black(x) → ¬white(x)) unless
you want to add it.

Is the formula above the intended formalization of your English sentence?
=== USER ===
Please add the extra assumption.
=== AGENT ===
I’m going to (1) update the formalization by **adding the extra assumption**
“every black animal is not white”, (2) show you the updated translation
in traditional Unicode syntax and in FOL-PRE, (3) ask you to confirm it,
and then (4) report the automatic prover result (with full output).

## (1) Updated translation (traditional Unicode syntax)

Assumption (extra):
- ∀x ∈ Animal. (black(x) → ¬white(x))

Goal (same as before):
- (∀x ∈ Animal. (swan(x) → white(x))) → (∀x ∈ Animal. (black(x) → ¬swan(x)))

So the full problem context is:
- Axiom: ∀x ∈ Animal. (black(x) → ¬white(x))
- Prove: (∀x ∈ Animal. (swan(x) → white(x))) → (∀x ∈ Animal. (black(x) → ¬swan(x)))

## (2) Updated translation (FOL-PRE proof problem)

‘‘‘folpre
problem["SwanBlack",
  typedecl["Animal"],
  preddecl["swan","Animal"],
  preddecl["white","Animal"],
  preddecl["black","Animal"],
  axiom["BlackNotWhite",
    forall["x","Animal",
      imp[pred["black",var["x"]],not[pred["white",var["x"]]]]
    ]
  ],
```

```
  goal["Claim",
    imp[
      forall["x","Animal",
        imp[pred["swan",var["x"]],pred["white",var["x"]]]
      ],
      forall["x","Animal",
        imp[pred["black",var["x"]],not[pred["swan",var["x"]]]]
      ]
    ]
  ]
]
‘‘‘

## (3) Is this updated translation correct?

In particular, is the added axiom exactly what you intended:
“black implies not white” for all animals?

## (4) Automatic prover result (full output)

Proof attempt was **successful**.

‘‘‘
RISC Theorem Proving Interface 1.8.5 (February 20, 2026)
https://www.risc.jku.at/research/formal/software/RISCTP
(C) 2022-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCTP -h" to see the available command line options.
------------------------------------------------------------------
Reading file /tmp/tmp160fjow1...
=== proof method ’meson’: model elimination, subgoal-oriented
Goal 3:[Claim.2] ⊤ ∧ ¬swan(_x§)
Iteration 1 (proof depth 1)... 2 clause applications.
Iteration 2 (proof depth 2)... 3 clause applications.
SUCCESS: the proof problem has been solved (3 clause applications, 12 ms).
===
SUCCESS termination (25 ms).
‘‘‘

## (5) Interactive prover

If you want, I can now run the **interactive prover** on exactly
the same FOL-PRE problem to inspect the proof steps.
=== USER ===
Please do so.
```

It should be noted that after the addition of the missing axiom the agent actually started the prover without asking for a confirmation by the user (which contradicts the initial system prompt and also the explanation given by the LLM in its response).



Proof Status: Success

Prover Output

Input File

Proof Problem

[-] **Problem Simplification:**

- [-] Claim (rule [⇒-R | ∨-R | ∧-L] on the goal)
  - [-] Claim (rule [∀-R | ∃-L] on the goal)
    - [-] Claim (rule [⇒-R | ∨-R | ∧-L] on the goal)
      - [ ] Claim (open)

[-] **Subproblems:**

1. Claim

[-] **Clause Forms:**

1. Claim

[-] **Proofs:**

1. [-] Claim (success)
   - [-] Claim.2.2 (success)
     - [-] Claim.2.2 (iteration 2) (success)
       - [-] ¬swan(_x§) [Claim.1] (success)
         - [-] ¬white(_x§) [BlackNotWhite] (success)
           - [-] black(_x§) [Claim.2.1] (success)

[-] **Proof Search:**

Proof [Claim]

```
We are given the following axioms:

[BlackNotWhite] ∀_x:Animal. black(_x) ∧ white(_x) ⇒ ⊥

The following "negated goals" represent the negation of the theorem to be proved:

[Claim.1] ∀_x:Animal. swan(_x) ⇒ white(_x)
[Claim.2.1] ⊤ ⇒ black(_x§)
[Claim.2.2] ⊤ ⇒ swan(_x§)

To prove the theorem, we apply the proof strategy MESON (model elimination, subgoal oriented)
to derive from the axioms and negated goals a contradiction. For this,
we prove some (not negated) goal from the "knowledge" represented by the other formulas.
We start the proof with the last goal; if this does not succeed, we also try the previous ones.
-----------------------------------------------------------------------------------------
SUCCESS: the proof has been completed.
```

Figure 4: The RISCTP GUI

Finally, the agent has offered an invocation of the prover in interactive mode. After a confirmation of the user, it starts the RISCTP user interface depicted in Figure 4. By pressing the “Prove” button, the proof is elaborated and may be inspected.

It is beyond the scope of this paper to discuss the pragmatics of the formulation of proof problems. However, Appendix A lists sample sessions for two different problems, the simple one shown above, and a more complicated one, with the three commercial LLMs listed above. From these conversations, one gets quickly a rough feeling of the different “personalities” of the LLMs that are exhibited by their interaction with the human.

A good part of this software was “vibe coded” with the help the LLM Google Gemini, see Appendix B. The actual software itself is listed in Appendix C.

5 Conclusions

We have reported on an experiment to develop with minimal efforts a “logical agent” that combines an LLM and a theorem prover and profits from their respective strengths: The LLM is competent in processing natural language, both for understanding and following instructions given by the human, and for translating natural language descriptions into formal proof problems. The theorem prover is competent in solving these problems, by constructing formal proofs that are open for independent inspection and checking. This combination is directed by natural language texts that both describe the formal language into which the problem is to be translated and (via system prompts and tool descriptions) the behavior expected from the agent in its interaction with external software. This approach also represents a blueprint for the integration of other symbolic (algebraic/logic) software with LLMs via natural language descriptions.

However, the experiment has been very limited in its scope. Further work is needed to address many open questions. How can the basic ReAct workflow be more refined, e.g., by separating the phases of syntax-checking, type-checking, and proving into separate stages? How can the translation be scaled to more complex problems that exceed the capabilities of the “few-shot” prompting technique exhibited by the examples given in the FOL-EXP document, e.g., by applying the “Three-of-Thought” paradigm? How do different LLMs compare to each other and are free LLMs competitive with commercial ones? How can we overcome the limitations of the context window of LLMs, e.g., by letting them manipulate externally managed stateful objects of discourse (“artifacts”) [6]. There is a wide range of possibilities that await further exploration.

References

[1] The Harmonic Team: Tudor Achim, Alex Best, Alberto Bietti, Kevin Der, Mathïs Fédérico, Sergei Gukov, Daniel Halpern-Leistner, Kirsten Henningsgard, Yury Kudryashov, Alexander Meiburg, Martin Michelsen, Riley Patterson, Eric Rodriguez, Laura Scharff, Vikram Shanker, Vladmir Sicca, Hari Sowrirajan, Aidan Swope, Matyas Tamas, Vlad Tenev, Jonathan Thomm, Harold Williams, and Lawrence Wu. Aristotle: IMO-level Automated Theorem Proving. *arXiv*, 2025. doi:10.48550/arXiv.2510.01346.

[2] Jay Alammar and Maarten Grootendorst. *Hands-On Large Language Models: Language*

Understanding and Generation. O'Reilly Media, 2022. https://www.oreilly.com/library/view/hands-on-large-language/9781098150952.

[3] Aristotle API. Harmonic, Palo Alto, CA, USA, 2026. https://aristotle.harmonic.fun.

[4] Ben Auffarth and Leonid Kuligin. *Generative AI with LangChain: Build production-ready LLM applications and advanced agents using Python, LangChain, and LangGraph*. Packt Publishing, 2025. https://www.packtpub.com/en-us/product/generative-ai-with-langchain-9781837022007.

[5] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, The SMT-LIB Initiative, 2021. http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2021-05-12.pdf.

[6] John Berrymand and Albert Ziegler. *Prompt Engineering for LLMs: The Art and Science of Building Large Language Model-Based Applications*. O'Reilly Media, 2025. https://www.oreilly.com/library/view/prompt-engineering-for/9781098156145.

[7] Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In André Platzer, editor, *Automated Deduction – CADE 28, 28th International Conference on Automated Deduction, Virtual Event, July 12–15*, pages 625–635, Cham, Switzerland, 2021. Springer International Publishing. doi:10.1007/978-3-030-79876-5_37.

[8] LangChain: Observe, Evaluate, and Deploy Reliable AI Agents. LangChain, Inc., San Francisco, CA, USA, 2026. https://www.langchain.com.

[9] Lean Programming Language. Lean FRO, Cambridge, MA, USA, 2026. https://lean-lang.org.

[10] The RISC Algorithm Language (RISCAL). Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, 2019. https://www.risc.jku.at/research/formal/software/RISCAL.

[11] The RISCTP Theorem Proving Interface). Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, 2024. https://www.risc.jku.at/research/formal/software/RISCTP.

[12] Wolfgang Schreiner. The RISC Algorithm Language (RISCAL) — Tutorial and Reference Manual. Technical report, RISC, Johannes Kepler University, Linz, Austria, March 2017. Available at [10].

[13] Wolfgang Schreiner. The RISCTP Theorem Proving Interface — Tutorial and Reference Manual. Technical report, RISC, Johannes Kepler University, Linz, Austria, June 2022. Available at [11].

[14] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *arXiv*, 2023. doi:https://doi.org/10.48550/arXiv.2201.11903.

[15] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. *arXiv*, 2023. doi:10.48550/arXiv.2305.10601.

[16] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *ICLR 2023: 11th International Conference on Learning Representations*, Kigali, Rwanda, May 1–5, 2023. doi:10.48550/arXiv.2210.03629.

A The Agent Sessions

We are now going to show various sessions of the logical agent that solve two logical problems:

The Swan Problem

```
Is it true that, if every swan is white,
then every black animal is not a swan?
```

The Superman Problem

```
Consider the following assumptions:

1. If superman were able and willing to prevent evil, he would do so.
2. If superman were unable to prevent evil, he would be impotent.
3. If superman were unwilling to prevent evil, he would be malevolent.
4. Superman does not prevent evil.
5. If Superman exists, he is neither impotent nor malevolent.

Is it true, that under these assumptions superman does not exist?

Use in your formulation only predicates, not constants or functions.
```

We are going to use for this the LLMS OpenAI GPT 5.2, Google Gemini 2.5 Pro, and Claude Sonnet 4.6. The sessions were all run on March 10, 2026.

A.1 OpenAI GPT 5.2

The Swan Problem

```
=== USER ===
Is it true that, if every swan is white, then every black animal is not a swan?
=== AGENT ===
I’m going to (0) use the official FOL-PRE syntax description, then (1) translate
your question into a FOL-PRE proof problem, and (2) ask you to confirm the
translation.

## (1) Translation into a FOL-PRE proof problem

Interpretation (tell me if you mean it differently):
- We talk about a single type of objects, **Animal**.
- Predicates:
  - ‘Swan(x)‘ means “x is a swan”
  - ‘White(x)‘ means “x is white”
  - ‘Black(x)‘ means “x is black”

Your statement: “If every swan is white, then every black animal is not a swan”
becomes:
```

~~~~
- Assumption: ∀x. Swan(x) ⇒ White(x)
- Conclusion: ∀x. Black(x) ⇒ ¬Swan(x)
- Whole: (∀x. Swan(x) ⇒ White(x)) ⇒ (∀x. Black(x) ⇒ ¬Swan(x))

Here is the corresponding **FOL-PRE** problem:

‘‘‘folpre
problem["SwanBlack",
  typedecl["Animal"],
  preddecl["Swan","Animal"],
  preddecl["White","Animal"],
  preddecl["Black","Animal"],
  goal["stmt",
    imp[
      forall["x","Animal",imp[pred["Swan",var["x"]],pred["White",var["x"]]]],
      forall["x","Animal",imp[pred["Black",var["x"]],not[pred["Swan",var["x"]]]]]
    ]
  ]
]
‘‘‘

## (2) Is this translation correct?

Two quick checks for meaning:
1. Do you want “black animal” to mean ‘Black(x)‘ (a predicate), as above?
2. Do you assume anything like “black implies not white” (i.e., ∀x. Black(x) ⇒
¬White(x))? I did **not** assume that.

Tell me “yes, correct” or specify changes, and I’ll proceed to (4) run an
automatic prover on the confirmed formalization and show its full output.
=== USER ===
Please add the assumption and then show me the proof problem again.
=== AGENT ===
I’m going to (1) extend the translation by adding the extra assumption you
requested (that black implies not white), and then (2) show you the updated
FOL-PRE proof problem and ask you to confirm it.

## (1) Added assumption

Added axiom:
- ∀x:Animal. Black(x) ⇒ ¬White(x)

## Updated FOL-PRE proof problem

‘‘‘folpre
problem["SwanBlack",
  typedecl["Animal"],
  preddecl["Swan","Animal"],
  preddecl["White","Animal"],
  preddecl["Black","Animal"],

  axiom["BlackNotWhite",
    forall["x","Animal",
~~~~

```
   imp[pred["Black",var["x"]],not[pred["White",var["x"]]]]
  ]
 ],

 goal["stmt",
  imp[
   forall["x","Animal",
    imp[pred["Swan",var["x"]],pred["White",var["x"]]]
   ],
   forall["x","Animal",
    imp[pred["Black",var["x"]],not[pred["Swan",var["x"]]]]
   ]
  ]
 ]
]
‘‘‘

## (2) Is this updated translation correct?

If yes, I will (4) run the automatic prover on exactly this problem and show you
the full prover output.
=== USER ===
Please show the problem in traditional syntax.
=== AGENT ===
I’m going to (1) restate the *same* proof problem you confirmed, but in
traditional first-order logic notation (∀, ⇒, ¬), and then (2) ask you to
confirm that this is indeed what you mean.

## (1) Traditional syntax version

**Type/domain:** \(Animal\)

**Predicates:** \(Swan(x), White(x), Black(x)\)

**Axiom (assumption):**
\[
\forall x\in Animal.\; Black(x) \Rightarrow \neg White(x)
\]

**Goal:**
\[
\Big(\forall x\in Animal.\; Swan(x)\Rightarrow White(x)\Big)
\Rightarrow
\Big(\forall x\in Animal.\; Black(x)\Rightarrow \neg Swan(x)\Big)
\]

## (2) Is this the intended meaning?

If yes, I will proceed to (4) run the automatic prover on the previously shown
FOL-PRE problem (the machine-checkable version) and show the full output.
=== USER ===
Please use unicode symbols, not LaTeX.
=== AGENT ===
I’m going to (1) restate the same axiom and goal using plain Unicode logical
```

~~~~
symbols (no LaTeX), and then (2) ask you to confirm it.

## (1) Traditional syntax with Unicode symbols

Domain/type: Animal
Predicates: Swan(x), White(x), Black(x)

Axiom (assumption):
- ∀x∈Animal. (Black(x) ⇒ ¬White(x))

Goal:
- (∀x∈Animal. (Swan(x) ⇒ White(x))) ⇒ (∀x∈Animal. (Black(x) ⇒ ¬Swan(x)))

## (2) Is this correct?

If yes, I will (4) run the automatic prover on the corresponding FOL-PRE problem
and show you the full prover output.
=== USER ===
Okay, start the proof.
=== AGENT ===
I’m going to (4) call an automatic prover on the confirmed FOL-PRE proof problem
and then present the full prover output.

## (4) Automatic prover result

Proof attempt was **successful**.

Full output produced by the prover:

```
RISC Theorem Proving Interface 1.8.5 (February 20, 2026)
https://www.risc.jku.at/research/formal/software/RISCTP
(C) 2022-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCTP -h" to see the available command line options.
-----------------------------------------------------------------
Reading file /tmp/tmp_qfeh9jz...
=== proof method ’meson’: model elimination, subgoal-oriented
Goal 3:[stmt.2] ⊤ ∧ ¬Swan(_x§)
Iteration 1 (proof depth 1)... 2 clause applications.
Iteration 2 (proof depth 2)... 3 clause applications.
SUCCESS: the proof problem has been solved (3 clause applications, 13 ms).
===
SUCCESS termination (28 ms).
```

I’m going to (5) offer to run an interactive prover session on the *same*
problem so you can inspect the proof steps. Do you want me to start the
interactive prover?
=== USER ===
Yes, please.
=== AGENT ===
I’m going to (5) start an interactive prover session on the same (confirmed)
FOL-PRE problem.
~~~~

(5) Interactive prover

I called the interactive prover with this problem:

```folpre
problem["SwanBlack",
  typedecl["Animal"],
  preddecl["Swan","Animal"],
  preddecl["White","Animal"],
  preddecl["Black","Animal"],

  axiom["BlackNotWhite",
    forall["x","Animal",
      imp[pred["Black",var["x"]],not[pred["White",var["x"]]]]
    ]
  ],

  goal["stmt",
    imp[
      forall["x","Animal",
        imp[pred["Swan",var["x"]],pred["White",var["x"]]]
      ],
      forall["x","Animal",
        imp[pred["Black",var["x"]],not[pred["Swan",var["x"]]]]
      ]
    ]
  ]
]
```

Result returned by the interface:

```
proof session terminated (successfully or unsuccessfully)
```

If you tell me what interface you’re using (web UI vs. terminal) and what you want to inspect (e.g., the derived clauses / proof tree / natural-deduction style steps), I can guide you through reproducing and examining the successful proof interactively.

The Superman Problem

=== USER ===
Consider the following assumptions:

1. If superman were able and willing to prevent evil, he would do so.
2. If superman were unable to prevent evil, he would be impotent.
3. If superman were unwilling to prevent evil, he would be malevolent.
4. Superman does not prevent evil.
5. If Superman exists, he is neither impotent nor malevolent.

Is it true, that under these assumptions superman does not exist?

```
Use in your formulation only predicates, not constants or functions.

=== AGENT ===
I am going to (0) use the official FOL-PRE syntax description, (1) translate
your assumptions and the question into a FOL-PRE proof problem **using only
predicates (no constants/functions)**, and then (2) ask you to confirm that the
translation matches your intent.

## (1) Translation into a FOL-PRE proof problem (predicates only)

Since we may not use a constant for “Superman”, we represent “Superman” by **a
unary predicate** ‘S(x)‘ meaning “x is Superman”. We use one type ‘E‘
(“entities”).

Predicates (all unary, on ‘E‘):
- ‘S(x)‘ : x is Superman
- ‘Able(x)‘ : x is able to prevent evil
- ‘Willing(x)‘ : x is willing to prevent evil
- ‘Prevent(x)‘ : x prevents evil
- ‘Impotent(x)‘ : x is impotent
- ‘Malevolent(x)‘ : x is malevolent
- ‘Exists(x)‘ : x exists  (so we don’t use existence of the domain element itself)

Now the assumptions become:

1. “If superman were able and willing to prevent evil, he would do so.”
   - For all x: if x is Superman and able and willing, then x prevents evil.
2. “If superman were unable to prevent evil, he would be impotent.”
   - For all x: if x is Superman and not able, then x is impotent.
3. “If superman were unwilling to prevent evil, he would be malevolent.”
   - For all x: if x is Superman and not willing, then x is malevolent.
4. “Superman does not prevent evil.”
   - With no constant, this must mean: **no Superman prevents evil**.
5. “If Superman exists, he is neither impotent nor malevolent.”
   - For all x: if x is Superman and exists, then not impotent and not malevolent.

Goal: “Under these assumptions superman does not exist.”
- With no constant: **no Superman exists**.

Here is the FOL-PRE problem:

‘‘‘fol
problem["Superman",
  typedecl["E"],
  preddecl["S","E"],
  preddecl["Able","E"],
  preddecl["Willing","E"],
  preddecl["Prevent","E"],
  preddecl["Impotent","E"],
  preddecl["Malevolent","E"],
  preddecl["Exists","E"],

  axiom["A1",
```

```
    forall["x","E",
      imp[
        and[pred["S",var["x"]],
            and[pred["Able",var["x"]],pred["Willing",var["x"]]]],
        pred["Prevent",var["x"]]
      ]
    ]
  ],

  axiom["A2",
    forall["x","E",
      imp[
        and[pred["S",var["x"]],not[pred["Able",var["x"]]]],
        pred["Impotent",var["x"]]
      ]
    ]
  ],

  axiom["A3",
    forall["x","E",
      imp[
        and[pred["S",var["x"]],not[pred["Willing",var["x"]]]],
        pred["Malevolent",var["x"]]
      ]
    ]
  ],

  axiom["A4",
    forall["x","E",
      imp[pred["S",var["x"]],not[pred["Prevent",var["x"]]]]
    ]
  ],

  axiom["A5",
    forall["x","E",
      imp[
        and[pred["S",var["x"]],pred["Exists",var["x"]]],
        and[not[pred["Impotent",var["x"]]],not[pred["Malevolent",var["x"]]]]
      ]
    ]
  ],

  goal["NoSupermanExists",
    not[exists["x","E",and[pred["S",var["x"]],pred["Exists",var["x"]]]]]
  ]
]
‘‘‘

## (2) Please confirm/correct the translation

Two key interpretation choices (forced by “no constants/functions”):

1. Assumption (4) “Superman does not prevent evil” was formalized as
   **∀x (S(x) → ¬Prevent(x))**, i.e. *no Superman prevents evil*.
```

```
   Is that what you mean, or do you intend “Superman (the unique individual)
   does not prevent evil”, which would require either a constant or an explicit
   uniqueness axiom?

2. “Superman exists” was modeled as a predicate ‘Exists(x)‘ and the goal as
   “no x that is Superman also Exists”.
   Is that acceptable, or do you instead want “existence” to be just ‘∃x S(x)‘
   (which would make assumption (5) read differently)?

Tell me what to change (if anything), and I will revise the FOL-PRE and ask
again. Once you confirm, I will run the automatic prover and show its full
output.
=== USER ===
Please show the problem in traditional syntax (use Unicode symbols).
=== AGENT ===
I am going to (1) restate the *same* translation I proposed, but written in
traditional first-order logic with Unicode symbols (∀, ∃, ¬, ∧, →).

### (1) Traditional (Unicode) formulation of the proof problem (predicates only)

Domain/type: \(E\) (entities)

Unary predicates on \(E\):
- \(S(x)\): “x is Superman”
- \(Able(x)\), \(Willing(x)\), \(Prevent(x)\), \(Impotent(x)\), \(Malevolent(x)\), \(Exists(x)\)

**Assumptions (axioms):**

1. \[
\forall x\in E.\ \big(S(x)\land Able(x)\land Willing(x)\big)\to Prevent(x)
\]

2. \[
\forall x\in E.\ \big(S(x)\land \neg Able(x)\big)\to Impotent(x)
\]

3. \[
\forall x\in E.\ \big(S(x)\land \neg Willing(x)\big)\to Malevolent(x)
\]

4. \[
\forall x\in E.\ S(x)\to \neg Prevent(x)
\]

5. \[
\forall x\in E.\ \big(S(x)\land Exists(x)\big)\to \big(\neg Impotent(x)\land \neg Malevolent(x)\big)
\]

**Goal:**

\[
\neg \exists x\in E.\ \big(S(x)\land Exists(x)\big)
\]
```

```
### (2) Please confirm/correct

Is this exactly the reading you want for (4) and for “exists” (as the predicate
\(Exists(x)\))? If you confirm, I will (4) run the automatic prover on the
corresponding FOL-PRE encoding and show the complete output.
=== USER ===
Please use Unicode symbols, not LaTeX.
=== AGENT ===
I am going to (1) restate the same proof problem again, but **purely in Unicode
logical symbols and plain text** (no LaTeX).

(1) Traditional Unicode formulation (predicates only)

Type/domain: E

Unary predicates on E:
S(x), Able(x), Willing(x), Prevent(x), Impotent(x), Malevolent(x), Exists(x)

Assumptions:

1. ∀x∈E. (S(x) ∧ Able(x) ∧ Willing(x)) → Prevent(x)

2. ∀x∈E. (S(x) ∧ ¬Able(x)) → Impotent(x)

3. ∀x∈E. (S(x) ∧ ¬Willing(x)) → Malevolent(x)

4. ∀x∈E. S(x) → ¬Prevent(x)

5. ∀x∈E. (S(x) ∧ Exists(x)) → (¬Impotent(x) ∧ ¬Malevolent(x))

Goal:

¬∃x∈E. (S(x) ∧ Exists(x))

(2) Please confirm/correct

Is this translation what you intend (especially that statement 4 means “no
Superman prevents evil”, and that “exists” is represented by the predicate
Exists(x))? If you confirm, I will (4) run the automatic prover on the
corresponding FOL-PRE encoding and show the full output.
=== USER ===
Please use for "exists" a quantifier, not a predicate, and then show me the
problem again.
=== AGENT ===
I am going to (1) change the formalization so that “exists” is expressed with
the quantifier ∃ (not a predicate), and then (2) show you the resulting
assumptions+goal again in pure Unicode syntax.

## (1) Revised translation using ∃ instead of an Exists predicate

We keep only these unary predicates on E:
S(x), Able(x), Willing(x), Prevent(x), Impotent(x), Malevolent(x)

Assumptions 1-4 stay the same. Assumption 5 “If Superman exists, he is neither
```

impotent nor malevolent” becomes:

(∃x∈E. S(x)) → (∀x∈E. S(x) → (¬Impotent(x) ∧ ¬Malevolent(x)))

This matches the idea: if there is at least one Superman, then every Superman is neither impotent nor malevolent.

The goal “Superman does not exist” becomes simply:

¬∃x∈E. S(x)

(1) Problem shown again (Unicode, no LaTeX)

Type/domain: E

Unary predicates on E:
S(x), Able(x), Willing(x), Prevent(x), Impotent(x), Malevolent(x)

Assumptions:

1. ∀x∈E. (S(x) ∧ Able(x) ∧ Willing(x)) → Prevent(x)

2. ∀x∈E. (S(x) ∧ ¬Able(x)) → Impotent(x)

3. ∀x∈E. (S(x) ∧ ¬Willing(x)) → Malevolent(x)

4. ∀x∈E. S(x) → ¬Prevent(x)

5. (∃x∈E. S(x)) → (∀x∈E. S(x) → (¬Impotent(x) ∧ ¬Malevolent(x)))

Goal:

¬∃x∈E. S(x)

(2) Please confirm/correct

Is this now the intended reading of assumption (5)? If you confirm, I will (4) run the automatic prover on the corresponding FOL-PRE encoding and show the full output.
=== USER ===
Please show me the problem in FOL-PRE syntax.
=== AGENT ===
I am going to (1) show the proof problem in **FOL-PRE syntax** using the version where “exists” is a quantifier (∃), not a predicate, and then (2) (since the translation is now fixed) report the result of an automatic prover run on exactly this FOL-PRE text, including its full output.

(1) Proof problem in FOL-PRE syntax

```fol
problem["Superman",
  typedecl["E"],
  preddecl["S","E"],
  preddecl["Able","E"],
```

```
preddecl["Willing","E"],
preddecl["Prevent","E"],
preddecl["Impotent","E"],
preddecl["Malevolent","E"],

axiom["A1",
  forall["x","E",
    imp[
      and[pred["S",var["x"]],
          and[pred["Able",var["x"]],pred["Willing",var["x"]]]],
      pred["Prevent",var["x"]]
    ]
  ]
],

axiom["A2",
  forall["x","E",
    imp[
      and[pred["S",var["x"]],not[pred["Able",var["x"]]]],
      pred["Impotent",var["x"]]
    ]
  ]
],

axiom["A3",
  forall["x","E",
    imp[
      and[pred["S",var["x"]],not[pred["Willing",var["x"]]]],
      pred["Malevolent",var["x"]]
    ]
  ]
],

axiom["A4",
  forall["x","E",
    imp[pred["S",var["x"]],not[pred["Prevent",var["x"]]]]
  ]
],

axiom["A5",
  imp[
    exists["x","E",pred["S",var["x"]]],
    forall["x","E",
      imp[
        pred["S",var["x"]],
        and[not[pred["Impotent",var["x"]]],not[pred["Malevolent",var["x"]]]]
      ]
    ]
  ]
],

goal["NoSuperman",
  not[exists["x","E",pred["S",var["x"]]]]
]
```

```
]
‘‘‘

## (4) Automatic prover result (full output)

The proof attempt was successful:

‘‘‘
RISC Theorem Proving Interface 1.8.5 (February 20, 2026)
https://www.risc.jku.at/research/formal/software/RISCTP
(C) 2022-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCTP -h" to see the available command line options.
-----------------------------------------------------------------
Reading file /tmp/tmp1183udwn...
=== proof method ’meson’: model elimination, subgoal-oriented
Goal 1:[NoSuperman] ⊤ ∧ ¬S(_x§)
Iteration 1 (proof depth 1)... 24 clause applications.
Iteration 2 (proof depth 2)... 60 clause applications.
Iteration 3 (proof depth 3)... 39 clause applications.
SUCCESS: the proof problem has been solved (39 clause applications, 15 ms).
===
SUCCESS termination (31 ms).
‘‘‘

## (5) Interactive prover?

I can also run the interactive prover on the **same** FOL-PRE problem text
above, so you can inspect the proof steps (or explore variants if you change any
assumption).
=== USER ===
Yes, please do so.
=== AGENT ===
I am going to (5) call the interactive prover on **exactly the same FOL-PRE
problem** as used for the automatic prover.

## (5) Interactive prover call

Interactive prover was started on:

‘‘‘fol
problem["Superman",
  typedecl["E"],
  preddecl["S","E"],
  preddecl["Able","E"],
  preddecl["Willing","E"],
  preddecl["Prevent","E"],
  preddecl["Impotent","E"],
  preddecl["Malevolent","E"],

  axiom["A1",
    forall["x","E",
      imp[
        and[pred["S",var["x"]],
```

```
      and[pred["Able",var["x"]],pred["Willing",var["x"]]]],
    pred["Prevent",var["x"]]
   ]
  ]
 ],

 axiom["A2",
  forall["x","E",
   imp[
    and[pred["S",var["x"]],not[pred["Able",var["x"]]]],
    pred["Impotent",var["x"]]
   ]
  ]
 ],

 axiom["A3",
  forall["x","E",
   imp[
    and[pred["S",var["x"]],not[pred["Willing",var["x"]]]],
    pred["Malevolent",var["x"]]
   ]
  ]
 ],

 axiom["A4",
  forall["x","E",
   imp[pred["S",var["x"]],not[pred["Prevent",var["x"]]]]
  ]
 ],

 axiom["A5",
  imp[
   exists["x","E",pred["S",var["x"]]],
   forall["x","E",
    imp[
     pred["S",var["x"]],
     and[not[pred["Impotent",var["x"]]],not[pred["Malevolent",var["x"]]]]
    ]
   ]
  ]
 ],

 goal["NoSuperman",
  not[exists["x","E",pred["S",var["x"]]]]
 ]
]
‘‘‘

Result returned by the interface after the session ended:

‘‘‘
proof session terminated (successfully or unsuccessfully)
‘‘‘
```

```
If you tell me what you did/what you saw in the interactive session (e.g.,
whether it displayed a finished proof, where it got stuck, or what command
sequence you used), I can help interpret the state and suggest the next
interactive steps.
```

A.2 Google Gemini 2.5 Pro

The Swan Problem

```
=== USER ===
Is it true that, if every swan is white, then every black animal is not a swan?
=== AGENT ===
I have read the description of the FOL-PRE syntax. Now, I will translate your
question into a proof problem in this syntax.

The question is: "Is it true that, if every swan is white, then every black
animal is not a swan?"

This can be broken down as follows:
*   We are talking about entities of type ‘Animal‘.
*   We have the properties ‘swan‘, ‘white‘, and ‘black‘, which can be represented as predicates.
*   The statement has the form of an implication: "if P then Q".
*   The premise P is "every swan is white".
*   The conclusion Q is "every black animal is not a swan".
*   There is an implicit assumption that an animal cannot be both black and white.
    I will add this as an axiom.

Here is the translation into the FOL-PRE syntax:‘‘‘
problem["BlackSwans",
  typedecl["Animal"],
  preddecl["swan", "Animal"],
  preddecl["white", "Animal"],
  preddecl["black", "Animal"],
  axiom["BlackNotWhite",
    forall["x", "Animal",
      imp[pred["black", var["x"]], not[pred["white", var["x"]]]]]],
  goal["MainGoal",
    imp[
      forall["x", "Animal",
        imp[pred["swan", var["x"]], pred["white", var["x"]]]],
      forall["x", "Animal",
        imp[pred["black", var["x"]], not[pred["swan", var["x"]]]]]
    ]
  ]
]
‘‘‘

**Explanation of the translation:**
*   ‘typedecl["Animal"]‘: Declares a type ‘Animal‘.
* ‘preddecl[...]‘: Declares the predicates ‘swan‘, ‘white‘, and ‘black‘ for
animals.
* ‘axiom["BlackNotWhite", ...]‘ states our background assumption: for any animal
‘x‘, if ‘x‘ is black, then ‘x‘ is not white.
* ‘goal["MainGoal", ...]‘ represents the statement we want to prove. It is an
```

```
implication (‘imp[...]‘) where:
* The premise is ‘forall["x", "Animal", imp[pred["swan", var["x"]],
pred["white", var["x"]]]]‘ which means "for every animal x, if x is a swan, then
x is white".
* The conclusion is ‘forall["x", "Animal", imp[pred["black", var["x"]],
not[pred["swan", var["x"]]]]]‘ which means "for every animal x, if x is black,
then x is not a swan".

Is this translation correct?
=== USER ===
Please show the problem in traditional syntax.
=== AGENT ===
Of course. Here is the same proof problem written in a more traditional, human-readable syntax:

**Types:**
*   ‘Animal‘

**Predicates:**
*   ‘swan(Animal)‘
*   ‘white(Animal)‘
*   ‘black(Animal)‘

**Axiom "BlackNotWhite":**
*   ‘∀x:Animal. (black(x) ⇒ ¬white(x))‘
    *   This reads: "For every animal x, if x is black, then x is not white."

**Goal "MainGoal":**
*   ‘(∀x:Animal. (swan(x) ⇒ white(x))) ⇒ (∀x:Animal. (black(x) ⇒ ¬swan(x)))‘
    * This reads: "If (for every animal x, if x is a swan, then x is white),
    then (for every animal x, if x is black, then x is not a swan)."

Does the original FOL-PRE translation correctly represent your question?
=== USER ===
Yes, please start the proof.
=== AGENT ===
I will now call the automatic prover with the translated problem.
The proof attempt was successful. Here is the full output from the prover:‘‘‘
RISC Theorem Proving Interface 1.8.5 (February 20, 2026)
https://www.risc.jku.at/research/formal/software/RISCTP
(C) 2022-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCTP -h" to see the available command line options.
-----------------------------------------------------------------
Reading file /tmp/tmpuyydvigc...
=== proof method ’meson’: model elimination, subgoal-oriented
Goal 3:[MainGoal.2] ⊤ ∧ ¬swan(_x§)
Iteration 1 (proof depth 1)... 2 clause applications.
Iteration 2 (proof depth 2)... 3 clause applications.
SUCCESS: the proof problem has been solved (3 clause applications, 11 ms).
===
SUCCESS termination (25 ms).
‘‘‘
The prover was able to find a proof, which means the statement is logically true
based on the given axiom.
```

```
Would you like to start an interactive session with the prover to investigate
the proof in more detail?
=== USER ===
Yes, please.
=== AGENT ===
The interactive prover session has been started. You can now interact with the
prover in the terminal. The session will remain active until you decide to
terminate it.
```

The Superman Problem

```
=== USER ===
Consider the following assumptions:

1. If superman were able and willing to prevent evil, he would do so.
2. If superman were unable to prevent evil, he would be impotent.
3. If superman were unwilling to prevent evil, he would be malevolent.
4. Superman does not prevent evil.
5. If Superman exists, he is neither impotent nor malevolent.

Is it true, that under these assumptions superman does not exist?

Use in your formulation only predicates, not constants or functions.

=== AGENT ===
I will start by downloading the description of the FOL-PRE syntax to understand
the format required for the proof problem.

Now, I will translate your question into a proof problem in the FOL-PRE syntax.

Here is the translation. I have used the following predicates to represent the
properties of a being ‘x‘:
*   ‘S(x)‘: ‘x‘ is Superman
*   ‘A(x)‘: ‘x‘ is able to prevent evil
*   ‘W(x)‘: ‘x‘ is willing to prevent evil
*   ‘P(x)‘: ‘x‘ prevents evil
*   ‘I(x)‘: ‘x‘ is impotent
*   ‘M(x)‘: ‘x‘ is malevolent

The proof problem is formulated as follows:‘‘‘
problem["superman",
  typedecl["Being"],
  preddecl["S", "Being"],
  preddecl["A", "Being"],
  preddecl["W", "Being"],
  preddecl["P", "Being"],
  preddecl["I", "Being"],
  preddecl["M", "Being"],
  axiom["a1", forall["x", "Being", imp[pred["S", var["x"]], imp[and[pred["A",
    var["x"]], pred["W", var["x"]]], pred["P", var["x"]]]]]],
  axiom["a2", forall["x", "Being", imp[pred["S", var["x"]], imp[not[pred["A",
    var["x"]]], pred["I", var["x"]]]]]],
  axiom["a3", forall["x", "Being", imp[pred["S", var["x"]], imp[not[pred["W",
```

```
    var["x"]]], pred["M", var["x"]]]]]],
  axiom["a4", forall["x", "Being", imp[pred["S", var["x"]], not[pred["P",
    var["x"]]]]]],
  axiom["a5", forall["x", "Being", imp[pred["S", var["x"]], and[not[pred["I",
    var["x"]]], not[pred["M", var["x"]]]]]]],
  goal["g", not[exists["x", "Being", pred["S", var["x"]]]]]
]
‘‘‘

Is this translation correct?
=== USER ===
Please show the problem in traditional syntax.
=== AGENT ===
Certainly. Here is the problem in traditional logical syntax.

**Predicates:**

*   ‘S(x)‘: ‘x‘ is Superman
*   ‘A(x)‘: ‘x‘ is able to prevent evil
*   ‘W(x)‘: ‘x‘ is willing to prevent evil
*   ‘P(x)‘: ‘x‘ prevents evil
*   ‘I(x)‘: ‘x‘ is impotent
*   ‘M(x)‘: ‘x‘ is malevolent

**Axioms (Assumptions):**

1.  ‘∀x: Being . (S(x) → ((A(x) ∧ W(x)) → P(x)))‘
    (For any being x, if x is Superman, then if x is able and willing to prevent
    evil, x does prevent evil.)
2.  ‘∀x: Being . (S(x) → (¬A(x) → I(x)))‘
    (For any being x, if x is Superman, then if x is unable to prevent evil, x
    is impotent.)
3.  ‘∀x: Being . (S(x) → (¬W(x) → M(x)))‘
    (For any being x, if x is Superman, then if x is unwilling to prevent evil,
    x is malevolent.)
4.  ‘∀x: Being . (S(x) → ¬P(x))‘
    (For any being x, if x is Superman, then x does not prevent evil.)
5.  ‘∀x: Being . (S(x) → (¬I(x) ∧ ¬M(x)))‘
    (For any being x, if x is Superman, then x is neither impotent nor malevolent.)

**Goal (Conclusion):**

‘¬(∃x: Being . S(x))‘
(There does not exist any being x such that x is Superman.)

Is this translation correct?
=== USER ===
Yes, please start the proof.
=== AGENT ===
I will now call the automatic prover with the translated proof problem.

The proof attempt was successful. Here is the full output from the prover:‘‘‘
RISC Theorem Proving Interface 1.8.5 (February 20, 2026)
https://www.risc.jku.at/research/formal/software/RISCTP
```

```
(C) 2022-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCTP -h" to see the available command line options.
-----------------------------------------------------------------
Reading file /tmp/tmpoxddao0y...
=== proof method ’meson’: model elimination, subgoal-oriented
Goal 1:[g] ⊤ ∧ ¬S(_x§)
Iteration 1 (proof depth 1)... 12 clause applications.
Iteration 2 (proof depth 2)... 30 clause applications.
Iteration 3 (proof depth 3)... 61 clause applications.
SUCCESS: the proof problem has been solved (61 clause applications, 14 ms).
===
SUCCESS termination (28 ms).
‘‘‘
The prover was able to prove that under the given assumptions, Superman does not
exist.

Would you like me to call an interactive prover on the problem to investigate
the proof?
=== USER ===
Yes, please.
=== AGENT ===
I will now call the interactive prover. You can now interact with the prover to
investigate the proof. When you are finished, please let me know.
```

A.3 Anthropic Claude Sonnet 4.6

The Swan Problem

```
=== USER ===
Is it true that, if every swan is white, every black animal is not a swan?
=== AGENT ===
I’ve carefully read the FOL-PRE syntax. Now let me proceed to **Step 1:
Translate the question into a proof problem in FOL-PRE syntax**.

The question is: *"If every swan is white, then every black animal is not a swan."*

- We need a type ‘Animal‘.
- We need unary predicates: ‘swan‘, ‘white‘, ‘black‘.
- The **axiom** is: every swan is white → ‘∀x:Animal. swan(x) ⇒ white(x)‘
- The **goal** is: every black animal is not a swan → ‘∀x:Animal. black(x) ⇒ ¬swan(x)‘

Here is the FOL-PRE translation:

‘‘‘
problem["SwanProblem",
  typedecl["Animal"],
  preddecl["swan","Animal"],
  preddecl["white","Animal"],
  preddecl["black","Animal"],
  axiom["everySwan IsWhite",
    forall["x","Animal",imp[pred["swan",var["x"]],pred["white",var["x"]]]]],
  goal["everyBlackIsNotSwan",
    forall["x","Animal",imp[pred["black",var["x"]],not[pred["swan",var["x"]]]]]]
```

~~~~
]
‘‘‘

**Step 2: Is this translation correct?**

In plain English, this reads:
- **Axiom**: For every animal x, if x is a swan, then x is white.
- **Goal**: For every animal x, if x is black, then x is not a swan.

Does this correctly capture your question? Please let me know if you’d like any
adjustments before I run the prover!
=== USER ===
Please show the problem in traditional syntax.
=== AGENT ===
Sure! Here is the proof problem written in traditional logic syntax:

**Type:** Animal

**Predicates:** swan, white, black (all unary, on Animal)

**Axiom** *(everySwan IsWhite)*:
$$\forall x \in \text{Animal}. \ \text{swan}(x) \Rightarrow \text{white}(x)$$

**Goal** *(everyBlackIsNotSwan)*:
$$\forall x \in \text{Animal}. \ \text{black}(x) \Rightarrow \neg\text{swan}(x)$$

In words:
- **Axiom**: Every animal that is a swan is white.
- **Goal**: Every animal that is black is not a swan.

Does this correctly reflect your intended question? If so, I’ll proceed to run
the automatic prover!
=== USER ===
Please use Unicode symbols, not LaTeX.
=== AGENT ===
Sure! Here is the proof problem in traditional logic syntax with Unicode symbols:

**Type:** Animal

**Predicates:** swan, white, black (all unary, on Animal)

**Axiom** *(everySwanIsWhite)*:
∀x ∈ Animal. swan(x) ⇒ white(x)

**Goal** *(everyBlackIsNotSwan)*:
∀x ∈ Animal. black(x) ⇒ ¬swan(x)

In words:
- **Axiom**: Every animal that is a swan is white.
- **Goal**: Every animal that is black is not a swan.

Does this correctly reflect your intended question? If so, I’ll proceed to run the automatic prover!
=== USER ===
Yes, please go ahead.
~~~~

```
=== AGENT ===
The automatic prover was **not successful**. This is actually a very interesting
and meaningful result! Let me explain why:

The proof **fails** because the goal is not logically derivable from the axiom
alone. The axiom only tells us:

> ∀x ∈ Animal. swan(x) ⇒ white(x)

But to conclude that every black animal is not a swan, we would also need to
know that **no animal can be both black and white** - i.e., we need an
additional axiom such as:

> ∀x ∈ Animal. black(x) ⇒ ¬white(x)

Without this assumption, it is logically possible for an animal to be both black
and white at the same time, in which case a black swan could also be white, and
the goal would not hold.

So the answer to your original question is: **No, it is not necessarily true**
that "if every swan is white, every black animal is not a swan" - unless we also
assume that black animals cannot be white.

Would you like me to:
1. Add the missing axiom and try the prover again?
2. Open an interactive prover session to investigate the failed proof attempt
further?
=== USER ===
Please add the missing axiom and show me the proof problem again.
=== AGENT ===
Sure! Here is the updated proof problem in traditional logic syntax with Unicode
symbols:

**Type:** Animal

**Predicates:** swan, white, black (all unary, on Animal)

**Axiom** *(everySwanIsWhite)*:
∀x ∈ Animal. swan(x) ⇒ white(x)

**Axiom** *(everyBlackIsNotWhite)*:
∀x ∈ Animal. black(x) ⇒ ¬white(x)

**Goal** *(everyBlackIsNotSwan)*:
∀x ∈ Animal. black(x) ⇒ ¬swan(x)

In words:
- **Axiom 1**: Every animal that is a swan is white.
- **Axiom 2**: Every animal that is black is not white.
- **Goal**: Every animal that is black is not a swan.

Does this correctly reflect your intended question? If so, I’ll proceed to run
the automatic prover!
=== USER ===
```

```
Okay, go ahead.
=== AGENT ===
The automatic prover was **successful** this time!

Here is a summary of the full output:
- The prover used the **meson** method (model elimination, subgoal-oriented).
- It solved the proof in just **2 iterations** with **3 clause applications** in **11 ms**.

In plain English, the reasoning is straightforward:
1. Take any black animal x.
2. By **Axiom 2**, x is not white.
3. But by **Axiom 1**, if x were a swan, it would have to be white.
4. Therefore, x cannot be a swan. []

So the answer to your original question is: **Yes, it is true** - but only under
the additional assumption that no black animal can be white. Without that
assumption, the statement cannot be proved, as we saw earlier.

Would you like me to open an interactive prover session to investigate the proof
in more detail?
=== USER ===
Please show me the output of the prover.
=== AGENT ===
Sure! Here is the full output produced by the prover:

‘‘‘
RISC Theorem Proving Interface 1.8.5 (February 20, 2026)
https://www.risc.jku.at/research/formal/software/RISCTP
(C) 2022-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCTP -h" to see the available command line options.
-----------------------------------------------------------------
Reading file /tmp/tmp88i0cinz...
=== proof method 'meson': model elimination, subgoal-oriented
Goal 2:[everyBlackIsNotSwan.1] ⊤ ∧ ¬swan(_x§)
Iteration 1 (proof depth 1)... 2 clause applications.
Iteration 2 (proof depth 2)... 3 clause applications.
SUCCESS: the proof problem has been solved (3 clause applications, 11 ms).
===
SUCCESS termination (24 ms).
‘‘‘

Would you like me to open an interactive prover session to investigate the proof
in more detail?
=== USER ===
Yes, please.
=== AGENT ===
The interactive prover session has terminated. I hope you were able to
investigate the proof to your satisfaction! Let me know if you have any further
questions or if you'd like to explore another logical problem.
```

The Superman Problem

```
=== USER ===
```

```
Consider the following assumptions:

1. If superman were able and willing to prevent evil, he would do so.
2. If superman were unable to prevent evil, he would be impotent.
3. If superman were unwilling to prevent evil, he would be malevolent.
4. Superman does not prevent evil.
5. If Superman exists, he is neither impotent nor malevolent.

Is it true, that under these assumptions superman does not exist?

Use in your formulation only predicates, not constants or functions.

=== AGENT ===
I’ve carefully read the FOL-PRE syntax. Now let me proceed to **Step 1:
Translate the problem into FOL-PRE syntax**.

The key idea is to use **only predicates** (no constants or functions), so
Superman is represented by a universally quantified variable over a type (e.g.,
"Being"), and we use unary predicates like ‘Able‘, ‘Willing‘, ‘Prevents‘,
‘Impotent‘, ‘Malevolent‘, ‘Exists‘ - all applied to that variable.

Here is my proposed translation:

‘‘‘
problem["Superman",
  typedecl["Being"],
  preddecl["Able", "Being"],
  preddecl["Willing", "Being"],
  preddecl["Prevents", "Being"],
  preddecl["Impotent", "Being"],
  preddecl["Malevolent", "Being"],
  preddecl["Superman", "Being"],
  axiom["A1",
    forall["x","Being",
      imp[pred["Superman",var["x"]],
        imp[and[pred["Able",var["x"]],pred["Willing",var["x"]]],
            pred["Prevents",var["x"]]]]]],
  axiom["A2",
    forall["x","Being",
      imp[pred["Superman",var["x"]],
        imp[not[pred["Able",var["x"]]],
            pred["Impotent",var["x"]]]]]],
  axiom["A3",
    forall["x","Being",
      imp[pred["Superman",var["x"]],
        imp[not[pred["Willing",var["x"]]],
            pred["Malevolent",var["x"]]]]]],
  axiom["A4",
    forall["x","Being",
      imp[pred["Superman",var["x"]],
          not[pred["Prevents",var["x"]]]]]],
  axiom["A5",
    forall["x","Being",
      imp[pred["Superman",var["x"]],
```

~~~
        and[not[pred["Impotent",var["x"]]],
            not[pred["Malevolent",var["x"]]]]]]],
  goal["SupermanNotExists",
    not[exists["x","Being",pred["Superman",var["x"]]]]]
]
‘‘‘

**Step 2: Checking the translation with you.**

Here is how each assumption maps to the formal axioms:

| # | Natural Language | Formal Axiom |
|---|---|---|
| 1 | If Superman were able **and** willing to prevent evil, he would do so. |
      For all x: if x is Superman, then if x is Able and Willing, then x Prevents evil. |
| 2 | If Superman were unable to prevent evil, he would be impotent. |
      For all x: if x is Superman, then if x is not Able, then x is Impotent. |
| 3 | If Superman were unwilling to prevent evil, he would be malevolent. |
      For all x: if x is Superman, then if x is not Willing, then x is Malevolent. |
| 4 | Superman does not prevent evil. |
      For all x: if x is Superman, then x does not Prevent evil. |
| 5 | If Superman exists, he is neither impotent nor malevolent. |
      For all x: if x is Superman, then x is not Impotent and not Malevolent. |
| Goal | Superman does not exist. | There is no x such that x is Superman. |

Note: "Superman exists" is captured by the predicate ‘Superman(x)‘ being
satisfiable - i.e., there exists some being x that is Superman. The goal says no
such being exists.

**Does this translation look correct to you?** If you’d like any adjustments,
please let me know!
=== USER ===
Please show the problem in FOL-PRE syntax.
=== AGENT ===
Here is the full proof problem in FOL-PRE syntax:

‘‘‘
problem["Superman",
  typedecl["Being"],
  preddecl["Able", "Being"],
  preddecl["Willing", "Being"],
  preddecl["Prevents", "Being"],
  preddecl["Impotent", "Being"],
  preddecl["Malevolent", "Being"],
  preddecl["Superman", "Being"],
  axiom["A1",
    forall["x","Being",
      imp[pred["Superman",var["x"]],
        imp[and[pred["Able",var["x"]],pred["Willing",var["x"]]],
            pred["Prevents",var["x"]]]]]],
  axiom["A2",
    forall["x","Being",
      imp[pred["Superman",var["x"]],
        imp[not[pred["Able",var["x"]]],
~~~

```
      pred["Impotent",var["x"]]]]]],
 axiom["A3",
  forall["x","Being",
   imp[pred["Superman",var["x"]],
    imp[not[pred["Willing",var["x"]]],
      pred["Malevolent",var["x"]]]]]],
 axiom["A4",
  forall["x","Being",
   imp[pred["Superman",var["x"]],
     not[pred["Prevents",var["x"]]]]]],
 axiom["A5",
  forall["x","Being",
   imp[pred["Superman",var["x"]],
    and[not[pred["Impotent",var["x"]]],
      not[pred["Malevolent",var["x"]]]]]]]],
 goal["SupermanNotExists",
  not[exists["x","Being",pred["Superman",var["x"]]]]]
]
‘‘‘

Does this translation look correct to you? If you’d like any adjustments, please
let me know!
=== USER ===
Please show it in traditional syntax, using Unicode symbols.
=== AGENT ===
Here is the proof problem expressed in traditional logic syntax with Unicode
symbols:

**Type:** Being

**Predicates:**
- ‘Able(x)‘ - x is able to prevent evil
- ‘Willing(x)‘ - x is willing to prevent evil
- ‘Prevents(x)‘ - x prevents evil
- ‘Impotent(x)‘ - x is impotent
- ‘Malevolent(x)‘ - x is malevolent
- ‘Superman(x)‘ - x is Superman

**Axioms:**

| | |
|---|---|
| A1 | ∀x∈Being. Superman(x) ⇒ (Able(x) ∧ Willing(x) ⇒ Prevents(x)) |
| A2 | ∀x∈Being. Superman(x) ⇒ (¬Able(x) ⇒ Impotent(x)) |
| A3 | ∀x∈Being. Superman(x) ⇒ (¬Willing(x) ⇒ Malevolent(x)) |
| A4 | ∀x∈Being. Superman(x) ⇒ ¬Prevents(x) |
| A5 | ∀x∈Being. Superman(x) ⇒ (¬Impotent(x) ∧ ¬Malevolent(x)) |

**Goal:**

| | |
|---|---|
| SupermanNotExists | ¬∃x∈Being. Superman(x) |
```

```
Does this look correct to you? If you’d like any adjustments, please let me know!
=== USER ===
Okay, go ahead.
=== AGENT ===
The proof attempt was **successful**!

Here is the full output produced by the prover:

‘‘‘
RISC Theorem Proving Interface 1.8.5 (February 20, 2026)
https://www.risc.jku.at/research/formal/software/RISCTP
(C) 2022-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
-----------------------------------------------------------------
Reading file /tmp/tmpki_ztw8q...
=== proof method ’meson’: model elimination, subgoal-oriented
Goal 1:[SupermanNotExists] ⊤ ∧ ¬Superman(_x§)
Iteration 1 (proof depth 1)... 12 clause applications.
Iteration 2 (proof depth 2)... 30 clause applications.
Iteration 3 (proof depth 3)... 61 clause applications.
SUCCESS: the proof problem has been solved (61 clause applications, 14 ms).
===
SUCCESS termination (29 ms).
‘‘‘

**Summary:** Under the five given assumptions, it is indeed provable that
**Superman does not exist**. The prover used the *model elimination* method and
found the proof at depth 3 with 61 clause applications in just 14 ms. The
reasoning essentially is:

- By A4, Superman does not prevent evil.
- By A1, he must therefore be either unable or unwilling.
- By A2, if unable, he is impotent; by A3, if unwilling, he is malevolent.
- But A5 says Superman is neither impotent nor malevolent - a contradiction.
- Therefore, no such being can exist.

---

**Step 5:** Would you like me to call the **interactive prover** on the same
problem so you can explore the proof in more detail?
=== USER ===
Yes, please.
=== AGENT ===
The interactive proof session has terminated. I hope you were able to explore
the proof interactively to your satisfaction!

If you have any further questions - about this problem, the proof, or any other
logical question - feel free to ask!
```

B The Vibe Coding Prompts

A substantial part of the source code of the logical agent was generated by “vibe coding” with the help of the LLM Google Gemini 3, either for generating the actual code or for generating code templates that were manually adapted to the problem at hand.

In the following, we list the prompts that we applied (mostly in different chatbot sessions), in the order in which the software was actually developed. Appendix C then lists the actual source files (with a rough estimation about the fraction of automatically generated code).

B.1 The GUI

With the following prompt, a template for the file `server.py` was generated:

```
Give me a flask-based python program that implements a web service. This service
receives via a POST a string and returns that string as a result.
```

With the following prompt, a template for the file `index.html` was generated:

```
Create a HTML 5 page that provides an interface to a web service running on URL
http://localhost:5000/echo. This page has on top a horizontally centered output
field of 80% width and 60% height; this field is vertically scrollable. Below
that we have a horizontally centered input field of 80% width and 25% height;
also this field is vertically scrollable. Below that we have a horizontally
centered button labeled "Submit". When this button is pressed, it is locked and
the content of the output field is extended by multiple lines: the first line is
"=== QUERY ===", the next lines consist of the content of the input field. Then
the input field is cleared. A POST request is sent to the service in the form of
a JSON object with key "message" whose associated string is the value of the
input field (before clearing). The service returns as a result a JSON object
with key "result" to which a string is associated. Then the content of the
output field is extended by a line "=== ANSWER ===" followed by the returned
string. Finally the button is unlocked.
```

B.2 The Agent

With the following prompt, a template for the file `agent.py` was generated:

```
Create a Python program that is based on the newest version of LangChain. It
connects to a remote LLM and is extended by a tool. This tool is based on a
local function "add" that receives the string representation of two numbers and
returns the string representation of the sum of the two numbers.

The program provides a function "agent" that receives a string that represents a
user query and returns a string that returns an answer of the agent. The agent
is based on the ReAct pattern: it applies the LLM to answer the query, but,
whenever the query asks for the addition of two numbers, it applies the function
"add" to perform the computation.

The agent remembers the previous conversion, i.e., every call of "agent" depends
on the history of the previous communication. Also provide a function "reset"
that clears the history to start a new conversation.
```

```
The program can be called from the command line with a query from the standard
input to which it responds with the answer to the standard output and then waits
for another input. Each query consists of a single input line. If the user
enters an empty line, the program terminates.

The program understands a command line option "-model" with arguments "gpt" or
"gemini" or "claude" to apply as the LLM either OpenAI GPT 5.2 (the default) or
Google Gemini 2.5 Pro or Anthropic Claude Sonnet 4.6. Embed into the program
comments that clearly describe which environment variables have to be set with
API keys to be able to connect to the various LLMs.
```

B.3 FOL-Exp

With the following prompt, the file `scanner.py` was generated:

```
Create a Python function "scan" that takes as input a single string "text"
(consisting of a sequence of lines) and decomposes this text into a list of
strings that we call "tokens"; this list is returned as the result. A "token" is
either an opening square bracket (a single character ’[’) or a closing square
bracket (’]’) or a comma (’,’) or a non-empty string of ASCII letters (called a
"tag") or a string of characters that starts and ends with a double quote
(called an "identifier"); between the starting and ending double quote of an
identifier there may be arbitrary printable characters but not the double quote
itself. The function ignores whitespace (blanks, tabs, newlines) between tokens.
If the text does not match this description, an error message is printed that
clearly indicates the error (the number of the erroneous line in the text, the
line itself, the position of the erroneous character in the line, the character
itself); furthermore an exception is raised. Give typing hints for all
declarations according to the latest features of the "typing" package.
```

With the following prompt, the file `exp.py` was generated:

```
In the following, give typing hints for all declarations according to the latest
features of the "typing" package.

Create a Python class "Exp" for the abstract syntax tree of formal expressions.
Every object of this class has a field "tag" which can be one of the following
strings:

  var, fun, true, false, pred, not, and, or, imp, iff, forall, exists

Every object has a field "args" (the arguments of the object) which is a list of
values that can be either strings of characters (excluding double quotes) or
objects of class "Exp" itself. The class overrides the standard function
"__str__" to return the string representation of the object in the form ’tag[]’
or ’tag[args]’ (without the single quotes) which starts with the tag of the
object. If the object has arguments, a comma-separated list of their string
representations is included (here an argument of type string is represented by
the sequence of its characters embedded in opening and closing double quotes).
```

With the following prompt (by far the most extensive one that we used), the file `parse.py` was generated:

In the following, give typing hints for all declarations according to the latest features of the "typing" package.

Given above class "Exp", create a Python function "parseTerm" that takes as its argument a list of strings which we call "tokens" and constructs from this a particular object of type "Exp" which we call "term".

Each token can be either an opening square bracket (a single character '[') or a closing square bracket (']') or a comma (',') or a string of characters that starts and ends with a double quote (called an "identifier") or one of the following strings which we call "tags":

```
  var, fun
```

Valid token sequences that can be parsed into terms are:

```
  var [ "str" ]
  fun [ "str" ]
  fun [ "str" , term ]
  fun [ "str" , term , term ]
  ...
```

Here "str" stands for an arbitrary string of characters embedded in double quotes and "term" stands for a list of tokens that can be parsed into a term. The "..." indicates the general pattern that allows arbitrary many such sequences.

If the sequence is not valid, the function prints an error message that includes the sublist that starts with the erroneous token; then the function throws an exception.

The function shall be based on an auxiliary function that takes as input the token list and a starting position "p" and returns a pair (n+p,term) whose first component "n" is the length of the longest subsequence that starts at "p" and can be parsed into a term; the second component "term" is that term. If the function cannot parse such a term, it returns the pair (-1,None).

Based on this, also create a Python function "parseFormula" that also takes as its argument a list of token and parses this list into a particular object of type "Exp" which we call "formula"; this formula is returned as the result.

The tokens are defined as in the case of terms except that we have the following tags:

```
  true, false, pred, not, and, or, imp, iff, forall, exists
```

Valid token sequences that can be parsed into formulas are:

```
  true [ ]
  false [ ]
  pred [ "str" ]
  pred [ "str" , term ]
  pred [ "str", term , term ]
  ...
```

```
  not [ formula ]
  and [ formula , formula ]
  or [ formula , formula ]
  imp [ formula , formula ]
  iff [ formula , formula ]
  forall [ "str" , "str" , formula ]
  exists [ "str" , "str" , formula ]
```

Here "formula" stands for a token sequence that can be parsed into a formula.

If the sequence is not valid, the function prints an error message that includes the sublist that starts with the erroneous token; then the function throws an exception.

Use for this an auxiliary function analogous to the auxiliary function for terms described above; the definition of the new auxiliary function can be based on the original one.

Based on this, also create a Python function "parseProblem" that also takes as its argument a list of token and parses this list into a particular object of type "Exp" which we call "proof problem"; this problem is returned as the result.

The tokens are defined as in the case of terms except that we have the following tags:

```
  problem, typedecl, fundecl, preddecl, axiom, goal
```

Valid token sequences that can be parsed into proof problems are:

```
  problem [ "str" ]
  problem [ "str" , decl ]
  problem [ "str" , decl , decl ]
  ...
```

Here "decl" stands for a "declaration", i.e., any of the following kinds of token sequences:

```
  typedecl [ "id" ]
  fundecl [ "id" , "id" ]
  fundecl [ "id" , "id" , "id" ]
  ...
  preddecl [ "id" ]
  preddecl [ "id" , "id" ]
  ...
  axiom [ "id" , formula ]
  goal [ "id" , formula ]
```

Here "formula" stands for a token sequence that can be parsed into a formula.

If the sequence is not valid, the function prints an error message that includes the sublist that starts with the erroneous token; then the function throws an exception.

```
Use for this an auxiliary function analogous to the auxiliary function for
formulas described above; the definition of the new auxiliary function can be
based on the original one.
```

B.4 RISCTP

With the following prompt, a template for the file `prover.py` was generated:

```
Write a Python function "execute" that takes a string (or if more appropriate a
list of strings) as an argument and returns a string as a result. The input
describes the call of an external command with some arguments; the function
executes the command and returns the output of the command as the result.
```

With the following prompt, the file `executor.py` was generated:

```
Write a Python function "start" that takes a list of strings as an argument. The
input describes the call of an external command with some arguments; the
function starts this command as a background process and returns some information
that allows to terminate this process later. Also write a corresponding function
"stop" that takes this information as input and terminates the denoted process.
```

C The Source Code

In this section, we present the complete source code that implements the logical agent. The source code assumes that the following Python packages have been installed:

```
pip install flask flask_cors
pip install langchain langchain-openai langchain-google-genai langchain-anthropic
```

The following listing shows the individual files, their lengths, and the portion that was automatically generated by an LLM using the prompts listed in Appendix B.

1. **The GUI**

   `index.html`: the web page (140 lines, 95% vibe coded)

   `server.py`: the service handler (30 lines, 95% vibe coded)

2. **The Agent**

   `agent.py`: the LangChain agent (150 lines, 90% vibe coded)

3. **FOL-EXP**

   `exp.py`: the abstract syntax trees (50 lines, 100% vibe coded)

   `scanner.py`: the scanner (70 lines, 100% vibe coded)

   `parser.py`: the parser (180 lines, 100% vibe coded)

   `checker.py`: the type checker (230 lines, 0% vibe coded)

4. **RISCTP**

risctp.py: the translator (220 lines, 0% vibe coded)

prover.py: the prover interface (90 lines, 10% vibe coded)

executor.py: the process executor (80 lines, 100% vibe coded)

The individual files are listed below.

C.1 The GUI

index.html This file (140 lines, 95% vibe coded) represents in the form of a web page the user interface to the logical agent.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>A Logical Agent</title>
    <style>
        body, html {
            height: 100%;
            width: 100%;
            margin: 0;
            display: flex;
            flex-direction: column;
            align-items: center;
            font-family: sans-serif;
            background-color: #f4f4f9;
        }

        #output-field {
            width: 80%;
            height: 60%;
            margin-top: 20px;
            padding: 10px;
            border: 1px solid #ccc;
            background-color: #fff;
            overflow-y: scroll;
            white-space: pre-wrap; /* Preserves line breaks */
            word-wrap: break-word;
        }

        #input-field {
            width: 80%;
            height: 15%;
            font-size: 12pt;
            margin-top: 15px;
            padding: 10px;
            border: 1px solid #ccc;
            resize: none; /* Disables manual resizing to keep layout stable */
            overflow-y: scroll;
        }
```

```
        #submit-btn {
            margin-top: 15px;
            padding: 10px 40px;
            font-size: 16px;
            cursor: pointer;
            background-color: #007bff;
            color: white;
            border: none;
            border-radius: 4px;
        }

        #reset-btn {
            margin-top: 15px;
            padding: 10px 40px;
            font-size: 16px;
            cursor: pointer;
            background-color: #007bff;
            color: white;
            border: none;
            border-radius: 4px;
        }

        #submit-btn:disabled {
            background-color: #a0a0a0;
            cursor: not-allowed;
        }
    </style>
</head>
<body>
    <h1>A Logical Agent</h1>

    <div id="output-field"></div>

    <textarea id="input-field" placeholder="Type your message here..."></textarea>

    <span>
    <button id="submit-btn">Submit</button>
    <button id="reset-btn">Reset</button>
    </span>

    <script>
        const outputField = document.getElementById(’output-field’);
        const inputField = document.getElementById(’input-field’);
        const submitBtn = document.getElementById(’submit-btn’);
        const resetBtn = document.getElementById(’reset-btn’);

        submitBtn.addEventListener(’click’, async () => {
            const message = inputField.value;
            if (!message.trim()) return;

            // 1. Lock the button
            submitBtn.disabled = true;
            resetBtn.disabled = true;
```

```
            // 2. Update output with Query
            outputField.textContent += ‘=== USER ===\n${message}\n‘;

            // 3. Clear input field
            inputField.value = ’’;

            // 4. Auto-scroll output to bottom
            outputField.scrollTop = outputField.scrollHeight;

            try {
                // 5. POST request to Flask
                const response = await fetch(’http://localhost:5000/agent’, {
                    method: ’POST’,
                    headers: {
                        ’Content-Type’: ’application/json’
                    },
                    body: JSON.stringify({ message: message })
                });

                if (!response.ok) throw new Error(’Server error’);

                const data = await response.json();

                // 6. Update output with Answer
                outputField.textContent += ‘=== AGENT ===\n${data.result}\n‘;
            } catch (error) {
                outputField.textContent += ‘=== ERROR ===\n${error.message}\n‘;
            } finally {
                // 7. Unlock the button and scroll
                submitBtn.disabled = false;
                resetBtn.disabled = false;
                outputField.scrollTop = outputField.scrollHeight;
            }
        });

        resetBtn.addEventListener(’click’, async () => {
            try {
                const response = await fetch(’http://localhost:5000/reset’, {
                    method: ’POST’,
                });
                if (!response.ok) throw new Error(’Server error’);
                inputField.textContent = ’’;
                outputField.textContent = ’’;
            } catch (error) {
                outputField.textContent += ‘=== ERROR ===\n${error.message}\n‘;
            }
        });
    </script>
</body>
</html>
```

`server.py` This file (30 lines, 95% vibe coded) implements the service handler that invokes the logical agent.

```
from flask import Flask, request, jsonify
from flask_cors import CORS

app = Flask(__name__)
CORS(app)

from agent import MultiModelAgent
agent = MultiModelAgent('gpt')

@app.route('/agent', methods=['POST'])
def agent_service():
    # Retrieve the JSON data from the request
    data = request.get_json()

    # Check if 'input_string' exists in the sent JSON
    if not data or 'message' not in data:
        return jsonify({"error": "Please provide a 'message' key in your JSON body"}), 400

    # extract the message
    message = data['message']

    # compute the result
    result = agent.agent(message)

    # Return the string back as a JSON response
    return jsonify({"result": result})

@app.route('/reset', methods=['POST'])
def reset_service():
    agent.reset()
    return ''

if __name__ == '__main__':
    # Run the server on localhost:5000
    app.run(debug=True, port=5000)
```

C.2 The Agent

`agent.py` This file (150 lines, 90% vibe coded) implements the logical agent on top of the LangChain framework.

```
import sys
import argparse
import os
import requests

from typing import TypedDict, Annotated, Sequence

from langchain_core.messages import BaseMessage, HumanMessage, SystemMessage
from langchain_core.tools import tool
from langchain_openai import ChatOpenAI
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_anthropic import ChatAnthropic
```

```
from langchain.agents import create_agent
from langgraph.checkpoint.memory import MemorySaver

import prover

# --- ENVIRONMENT VARIABLES SETUP ---
# To use these LLMs, ensure the following keys are in your environment:
# OpenAI:  export OPENAI_API_KEY=’your-key-here’
# Google:  export GOOGLE_API_KEY=’your-key-here’
# Claude:  export ANTHROPIC_API_KEY=’your-key-here’

# 1. Define the Local Tools

@tool
def download(url: str) -> str:
    """Download resource from the denoted URL and return its content."""
    print("download", url)
    try:
        with requests.get(url) as r:
            return r.text
    except requests.exceptions.RequestException as e:
        return f"An error occurred: {e}"

@tool
def prove(fol_text: str, interactive: bool=False) -> tuple[bool,str]:
    """
    This function applies a prover to a proof problem, either interactively or
    automatically (depending on the value of the second parameter). The first
    parameter is the text of the proof problem in the FOL-PRE syntax
    described at https://www.risc.jku.at/people/schreine/FOL-PRE.txt.
    In the automatic mode, the function returns ’True’ if and if the the proof
    could be completed successfully, together with the output produced by the
    prover. In the interactive mode, the function always returns False and
    the empty string (the function blocks until the session has terminated).
    """
    print("prove", "interactive" if interactive else "automatic")
    return prover.prove(fol_text, interactive)

SYSTEM_PROMPT=\
’’’
You are a helpful assistant that answers logical questions in the following way
(inform the user before every step what you are going to do):
0. You download the description of the FOL-PRE syntax from
   https://www.risc.jku.at/people/schreine/FOL-PRE.txt and read it carefully.
1. You translate the question into a proof problem in the FOL-PRE syntax
   (make sure it is really in this syntax, do not just guess).
2. You ask the user whether this translation is correct.
3. If the translation is not correct, you follow the suggestions of the user
   to come up with a correct translation; then go back to step 2.
4. Once the translation is correct, you call an automatic prover with the proof
   problem and present its result (state whether the proof attempt was successful
   and show the full output produced by the prover).
5. Finally you also offer to to call an interactive prover on the problem
   for investigating the successful proof or the unsuccessful proof attempt.
```

```
   Call the interactive prover with the same translation that you used
   for the automatic prover.
Do not try to answer a logical question in any other way.
'''

class MultiModelAgent:
    def __init__(self, model_type: str):
        self.tools = [download, prove]
        self.memory = MemorySaver()
        self.config = {"configurable": {"thread_id": "default_user"}}
        self.counter = 0

        # 2. Initialize the Remote LLM based on user selection
        if model_type == "gpt":
            # GPT-5.2 is the flagship 2026 model for reasoning tasks
            self.llm = ChatOpenAI(model="gpt-5.2", temperature=0)
        elif model_type == "gemini":
            # Gemini 2.5 Pro released in mid-2025
            # self.llm = ChatGoogleGenerativeAI(model="gemini-2.5-pro", temperature=0)
            self.llm = ChatGoogleGenerativeAI(model="gemini-2.5-pro", temperature=0)
        elif model_type == "claude":
            # Claude 4.6 Sonnet released Feb 2026
            self.llm = ChatAnthropic(model="claude-sonnet-4-6", temperature=0)
        else:
            raise ValueError("Unsupported model type.")

        # 3. Create the ReAct Agent using LangGraph
        self.agent_executor = create_agent(
            self.llm,
            self.tools,
            checkpointer=self.memory
        )

    def agent(self, query: str) -> str:
        """Processes a query and maintains history."""
        self.counter = self.counter+1
        if self.counter == 1:
            messages = [SystemMessage(content=SYSTEM_PROMPT),HumanMessage(content=query)]
        else:
            messages = [HumanMessage(content=query)]
        input_message = {"messages": messages}
        response = self.agent_executor.invoke(input_message, config=self.config)
        # The last message in the sequence is the agent’s final response
        return response["messages"][-1].text

    def reset(self):
        """Clears the history by generating a new thread ID."""
        self.counter = 0
        import uuid
        self.config["configurable"]["thread_id"] = str(uuid.uuid4())
        print("(Conversation history has been reset.)")

def main():
    parser = argparse.ArgumentParser(description="LangChain 2026 ReAct Agent")
```

```
    parser.add_argument("-model", choices=["gpt", "gemini", "claude"], default="gpt",
                        help="Select the LLM (default: gpt-5.2)")
    args = parser.parse_args()

    app = MultiModelAgent(args.model)
    print(f"--- Agent Active (Model: {args.model}) ---")
    print("Enter your query. Enter an empty line to exit. Type 'reset' to clear history.")

    while True:
        try:
            user_input = sys.stdin.readline().strip()

            if not user_input:
                break

            if user_input.lower() == "reset":
                app.reset()
                continue

            answer = app.agent(user_input)
            print(f"Assistant: {answer}")

        except EOFError:
            break

if __name__ == "__main__":
    main()
```

C.3 FOL-EXP

`exp.py` This file (50 lines, 100% vibe coded) implements the type of abstract syntax trees of FOL-EXP expressions.

```
from typing import List, Literal, Union, Self, Final

# A type alias for the allowed argument types: Strings or nested Exp objects
ExpArg = Union[str, 'Exp']

class Exp:
    def __init__(
        self,
        tag: Literal[
            "var", "fun", "true", "false", "pred", "not",
            "and", "or", "imp", "iff", "forall", "exists"
        ],
        args: List[ExpArg] #| None = None
    ) -> None:
        """
        Initializes an Abstract Syntax Tree node for a formal expression.

        :param tag: The operator or quantifier type.
        :param args: A list of arguments (either strings or other Exp objects).
        """
```

```
        self.tag: Final[str] = tag
        self.args: Final[List[ExpArg]] = args if args is not None else []

    def __str__(self) -> str:
        """
        Returns the string representation in the format tag[] or tag["arg1",...]
        Strings are wrapped in double quotes; Exp objects are called recursively.
        """
        if not self.args:
            return f"{self.tag}[]"

        formatted_args: List[str] = []
        for arg in self.args:
            if isinstance(arg, str):
                formatted_args.append(f’"{arg}"’)
            else:
                formatted_args.append(str(arg))

        return f"{self.tag}[{’,’.join(formatted_args)}]"

# --- Example Usage ---
if __name__ == "__main__":
    # Representing: ∀x (P(x) → Q(x))
    p_x = Exp(tag="pred", args=["P", Exp(tag="var", args=["x"])])
    q_x = Exp(tag="pred", args=["Q", Exp(tag="var", args=["x"])])
    implication = Exp(tag="imp", args=[p_x, q_x])
    forall_stmt = Exp(tag="forall", args=["x", implication])

    print(forall_stmt)
    # Output: forall["x",imp[pred["P",var["x"]],pred["Q",var["x"]]]]
```

scanner.py This file (70 lines, 100% vibe coded) implements the lexical analyzer (“scanner”) for FOL-EXP.

```
import re
from typing import List, Final

def scan(text: str) -> List[str]:
    """
    Scans the input text and decomposes it into a list of tokens:
    ’[’, ’]’, ’,’, tags (ASCII letters), and identifiers ("quoted strings").
    """
    # Define regex patterns for each token type using named groups
    # Tag: one or more ASCII letters
    # Identifier: double quote, any non-quote chars, closing double quote
    token_specification: Final[List[tuple[str, str]]] = [
        (’BRACKET_OPEN’,  r’\[’),
        (’BRACKET_CLOSE’, r’\]’),
        (’COMMA’,         r’,’),
        (’TAG’,           r’[a-zA-Z]+’),
        (’IDENTIFIER’,    r’"[^"]*"’),
        (’SKIP’,          r’[ \t\n\r]+’),  # Whitespace to be ignored
        (’MISMATCH’,      r’.’),           # Any other character (error)
```

```
    ]

    # Combine patterns into a single regex
    tok_regex: str = '|'.join(f'(?P<{name}>{pattern})' for name, pattern in token_specification)
    line_num: int = 1
    line_start: int = 0
    tokens: List[str] = []

    for mo in re.finditer(tok_regex, text):
        kind: str | None = mo.lastgroup
        value: str = mo.group()
        column: int = mo.start() - line_start + 1

        if kind == 'SKIP':
            # Update line count if the skipped whitespace contains newlines
            if '\n' in value:
                line_num += value.count('\n')
                line_start = mo.end() - (value[::-1].find('\n'))
            continue
        elif kind == 'MISMATCH':
            # Find the full text of the current line for the error message
            lines = text.splitlines()
            current_line = lines[line_num - 1] if line_num <= len(lines) else ""

            error_msg = (
                f"\n[Error] Illegal character '{value}' at line {line_num}, column {column}\n"
                f"Line {line_num}: {current_line}\n"
                f"{' ' * (len(str(line_num)) + 8 + column - 1)}^"
            )
            print(error_msg)
            raise ValueError(f"Scanning error at line {line_num}: unexpected character {value!r}")
        else:
            # For all valid tokens, append the literal string found
            tokens.append(value)

            # Handle newlines that might be inside identifiers (if allowed by your spec)
            if '\n' in value:
                line_num += value.count('\n')
                line_start = mo.end() - (value[::-1].find('\n'))

    return tokens

# --- Example Usage ---
if __name__ == "__main__":
    sample_text = """
    forall["x","A",imp[pred["p",var["x"]],
      exists["y","B",pred["q",var["x"],var["y"]]]]]
    """
    try:
        result = scan(sample_text)
        print(f"Tokens found: {result}")
    except ValueError as e:
        pass
```

`parser.py` This file (180 lines, 100% vibe coded) implements the syntactic analyzer (“parser”) for FOL-EXP.

```
from typing import List, Tuple, Literal, Optional, Final, TypeAlias, Union

from exp import Exp
from scanner import scan

# Define the argument type for the Exp class
ExpArg: TypeAlias = Union[str, 'Exp']

# Define the return type for auxiliary functions: (new_position, expression)
ParseResult: TypeAlias = Tuple[int, Optional['Exp']]

def _report_error(tokens: List[str], p: int) -> None:
    """Prints the error sublist and raises an exception."""
    sublist: List[str] = tokens[p:]
    error_msg: str = f"Parsing error at position {p}. Remaining tokens: {sublist}"
    print(error_msg)
    raise ValueError(error_msg)

# --- Term Parser ---

def _aux_parse_term(tokens: List[str], p: int) -> ParseResult:
    if p >= len(tokens) or tokens[p] not in ("var", "fun"):
        return -1, None

    tag: str = tokens[p]
    # Expecting: tag [ "name" ... ]
    if p + 2 >= len(tokens) or tokens[p+1] != "[" or not tokens[p+2].startswith('"'):
        return -1, None

    name: str = tokens[p+2].strip('"')
    args: List[ExpArg] = [name]
    curr: int = p + 3

    while curr < len(tokens) and tokens[curr] == ",":
        curr += 1
        next_p, sub_term = _aux_parse_term(tokens, curr)
        if next_p == -1 or sub_term is None:
            return -1, None
        args.append(sub_term)
        curr = next_p

    if curr < len(tokens) and tokens[curr] == "]":
        return curr + 1, Exp(tag=tag, args=args) # type: ignore

    return -1, None

def parseTerm(tokens: List[str]) -> Exp:
    pos, res = _aux_parse_term(tokens, 0)
    if pos != len(tokens) or res is None:
        _report_error(tokens, max(0, pos))
    return res
```

```
# --- Formula Parser ---

def _aux_parse_formula(tokens: List[str], p: int) -> ParseResult:
    if p >= len(tokens):
        return -1, None

    tag: str = tokens[p]

    # 1. Atomic: true, false
    if tag in ("true", "false"):
        if p + 2 < len(tokens) and tokens[p+1] == "[" and tokens[p+2] == "]":
            return p + 3, Exp(tag=tag, args=[]) # type: ignore
        return -1, None

    # 2. Predicate: pred["str", term, ...]
    if tag == "pred":
        if p + 2 < len(tokens) and tokens[p+1] == "[" and tokens[p+2].startswith(’"’):
            args: List[ExpArg] = [tokens[p+2].strip(’"’)]
            curr = p + 3
            while curr < len(tokens) and tokens[curr] == ",":
                curr += 1
                next_p, term = _aux_parse_term(tokens, curr)
                if next_p == -1 or term is None: return -1, None
                args.append(term)
                curr = next_p
            if curr < len(tokens) and tokens[curr] == "]":
                return curr + 1, Exp(tag="pred", args=args) # type: ignore

    # 3. Unary: not[formula]
    if tag == "not":
        if p + 1 < len(tokens) and tokens[p+1] == "[":
            next_p, f = _aux_parse_formula(tokens, p + 2)
            if next_p != -1 and f and next_p < len(tokens) and tokens[next_p] == "]":
                return next_p + 1, Exp(tag="not", args=[f]) # type: ignore

    # 4. Binary: and, or, imp, iff [f1, f2]
    if tag in ("and", "or", "imp", "iff"):
        if p + 1 < len(tokens) and tokens[p+1] == "[":
            p1, f1 = _aux_parse_formula(tokens, p + 2)
            if p1 != -1 and f1 and p1 < len(tokens) and tokens[p1] == ",":
                p2, f2 = _aux_parse_formula(tokens, p1 + 1)
                if p2 != -1 and f2 and p2 < len(tokens) and tokens[p2] == "]":
                    return p2 + 1, Exp(tag=tag, args=[f1, f2]) # type: ignore

    # 5. Quantifiers: forall, exists ["str", "str", formula]
    if tag in ("forall", "exists"):
        try:
            if tokens[p+1] == "[" and tokens[p+2].startswith(’"’) and tokens[p+3] == "," \
               and tokens[p+4].startswith(’"’) and tokens[p+5] == ",":
                v1, v2 = tokens[p+2].strip(’"’), tokens[p+4].strip(’"’)
                next_p, f = _aux_parse_formula(tokens, p + 6)
                if next_p != -1 and f and next_p < len(tokens) and tokens[next_p] == "]":
                    return next_p + 1, Exp(tag=tag, args=[v1, v2, f]) # type: ignore
```

```
        except IndexError: pass

    return -1, None

def parseFormula(tokens: List[str]) -> Exp:
    pos, res = _aux_parse_formula(tokens, 0)
    if pos != len(tokens) or res is None:
        _report_error(tokens, max(0, pos))
    return res

# --- Problem Parser ---

def _aux_parse_decl(tokens: List[str], p: int) -> ParseResult:
    if p >= len(tokens): return -1, None
    tag: str = tokens[p]

    # typedecl, fundecl, preddecl (sequences of "id")
    if tag in ("typedecl", "fundecl", "preddecl"):
        if p + 2 < len(tokens) and tokens[p+1] == "[" and tokens[p+2].startswith(’"’):
            args: List[ExpArg] = [tokens[p+2].strip(’"’)]
            curr = p + 3
            while curr < len(tokens) and tokens[curr] == ",":
                if curr + 1 < len(tokens) and tokens[curr+1].startswith(’"’):
                    args.append(tokens[curr+1].strip(’"’))
                    curr += 2
                else: return -1, None
            if curr < len(tokens) and tokens[curr] == "]":
                return curr + 1, Exp(tag=tag, args=args) # type: ignore

    # axiom, goal ["id", formula]
    if tag in ("axiom", "goal"):
        if p + 3 < len(tokens) and tokens[p+1] == "[" and tokens[p+2].startswith(’"’) and tokens[p+3] == ","
            id_val: str = tokens[p+2].strip(’"’)
            next_p, f = _aux_parse_formula(tokens, p + 4)
            if next_p != -1 and f and next_p < len(tokens) and tokens[next_p] == "]":
                return next_p + 1, Exp(tag=tag, args=[id_val, f]) # type: ignore

    return -1, None

def _aux_parse_problem(tokens: List[str], p: int) -> ParseResult:
    if p >= len(tokens) or tokens[p] != "problem":
        return -1, None
    if p + 2 >= len(tokens) or tokens[p+1] != "[" or not tokens[p+2].startswith(’"’):
        return -1, None

    args: List[ExpArg] = [tokens[p+2].strip(’"’)]
    curr: int = p + 3

    while curr < len(tokens) and tokens[curr] == ",":
        curr += 1
        next_p, decl = _aux_parse_decl(tokens, curr)
        if next_p == -1 or decl is None: return -1, None
        args.append(decl)
        curr = next_p
```

```
    if curr < len(tokens) and tokens[curr] == "]":
        return curr + 1, Exp(tag="problem", args=args) # type: ignore
    return -1, None

def parseProblem(tokens: List[str]) -> Exp:
    pos, res = _aux_parse_problem(tokens, 0)
    if pos != len(tokens) or res is None:
        _report_error(tokens, max(0, pos))
    return res

# --- Example Usage ---
if __name__ == "__main__":
    text = """
problem["SupermanExistence",
  ... (omitted)
]
 """
    try:
        print(text)
        tokens = scan(text)
        formula = parseProblem(tokens)
        print(formula)
    except ValueError as e:
        pass
```

`checker.py` This file (230 lines, 0% vibe coded) implements the type checker for FOL-EXP.

```
from exp import Exp, ExpArg

# ----------------------------------------------------------------------------
# type-checking expressions
# ----------------------------------------------------------------------------

# a signature (function/predicate name + argument types)
Sig = tuple[str,...]

# an environment
tenv: set[str]       # set of declared types
fenv: dict[Sig, str] # mapping of constant/function signatures to result types
penv: set[Sig]       # set of predicate signatures
venv: list[tuple[str, str]] # stack of variable/type pairs
agenv: set[str]      # set of axiom/goal formulas

class CheckException(Exception):
   def __init__(self, msg:str):
       super(CheckException, self).__init__(msg)

def check(cond: bool, msg: str, exp: (ExpArg|None)=None)->None:
    if cond:
        return
    estr: str = ': ' + (str(exp) if exp is not None else '')
```

```
    raise CheckException('ERROR (' + msg + ')' + estr)

def check_problem(problem: Exp)->None:
    '''
    check a problem, returns normally if everything is fine,
    otherwise raises an exception
    '''
    global tenv, fenv, penv, venv, agenv
    tenv = set()
    fenv = { }
    penv = set()
    venv = [ ]
    agenv = set()
    args: list[ExpArg] = problem.args
    n: int = len(args)
    check(n >= 1, 'illformed problem', problem)
    name = getname(args[0])
    for decl in problem.args[1:]:
        check(isinstance(decl,Exp), 'illformed declaration', decl)
        match decl.tag:
            case 'typedecl':
                check_typedecl(decl)
            case 'fundecl':
                check_fundecl(decl)
            case 'preddecl':
                check_preddecl(decl)
            case 'axiom':
                check_axiomgoal(decl)
            case 'goal':
                check_axiomgoal(decl)
            case _:
                check(False, 'unknown kind of declaration', decl)

def check_typedecl(decl: Exp)->None:
    global tenv
    name: str = declgetname(decl)
    check(not istype(name), 'doubly declared type', decl)
    tenv.add(name)
def check_fundecl(decl: Exp)->None:
    global fenv
    sig, rtype = funsig(decl)
    check(not isfun(sig), 'doubly declared function', decl)
    fenv[sig] = rtype
def check_preddecl(decl: Exp)->None:
    global penv
    sig: Sig = predsig(decl)
    check(not ispred(sig), 'doubly declared predicate', decl)
    penv.add(sig)
def check_axiomgoal(decl: Exp)->None:
    global agenv
    name: str = declgetname(decl)
    check(not isaxiomgoal(name), 'doubly declared axiom/goal', decl)
    formula: Exp = axiomgoalformula(decl)
    check_formula(formula)
```

```
    agenv.add(name)

def declgetname(decl: Exp)->str:
    args: list[ExpArg] = decl.args
    n: int = len(args)
    check(n >= 1, 'illformed declaration', decl)
    return getname(args[0])
def getname(exp: ExpArg)->str:
    check(isinstance(exp, str), 'not an identifier', exp)
    return exp
def funsig(decl: Exp)->tuple[Sig,str]:
    args: list[ExpArg] = decl.args
    n: int = len(args)
    check(n >= 2, 'illformed function declaration', decl)
    for arg in args:
       check(isinstance(arg, str), 'illformed function declaration', decl)
    for arg in args[1:]:
       check(istype(arg), 'undeclared type', arg)
    return tuple(args[:-1]),args[-1]
def predsig(decl: Exp)->Sig:
    args: list[ExpArg] = decl.args
    n: int = len(args)
    check(n >= 1, 'illformed predicate declaration', decl)
    for arg in args:
       check(isinstance(arg, str), 'illformed predicate declaration', decl)
    for arg in args[1:]:
       check(istype(arg), 'undeclared type', arg)
    return tuple(args)
def axiomgoalformula(decl: Exp)->Exp:
    args: list[ExpArg] = decl.args
    n: int = len(args)
    check(n == 2, 'illformed axiom/goal declaration', decl)
    check(isinstance(args[1],Exp), 'illformed axiom/goal declaration', decl)
    return args[1]

def istype(name: str)->bool:
    global tenv
    return name in tenv
def isfun(sig: Sig)->bool:
    global fenv
    return sig in fenv
def ispred(sig: Sig)->bool:
    global penv
    return sig in penv
def isaxiomgoal(name: str)->bool:
    global agenv
    return name in agenv

def check_formula(formula: Exp)->None:
    '''
    check a formula, returns normally if everything is fine,
    otherwise raises an exception
    '''
    args: List[ExpArg] = formula.args
```

```
    n: int = len(args)
    match formula.tag:
        case 'true' | 'false':
            check(n == 0, "illformed truth literal", formula)
        case 'pred':
            check_atomic(formula)
        case 'not':
            check(n == 1, "invalid application of unary connective", formula)
            check_argformula(args[0])
        case 'and' | 'or' | 'imp' | 'iff':
            check(n == 2, "invalid application of binary connective", formula)
            check_argformula(args[0])
            check_argformula(args[1])
        case 'forall' | 'exists':
            check(n == 3, "invalid quantifier application", formula)
            vname: str = getname(args[0])
            tname: str = getname(args[1])
            check(istype(tname), "unknown type", args[1])
            venv.append((vname,tname))
            check_argformula(args[2])
            venv.pop()
        case _:
            check(false, "unknown kind of formula", formula)

def check_argformula(exp: ExpArg)->None:
    check(isinstance(exp, Exp), 'illformed formula', exp)
    check_formula(exp)

def check_atomic(formula: Exp)->None:
    args: List[ExpArg] = formula.args
    n: int = len(args)
    check(n >= 1, "illformed atomic formula", formula)
    pname: str = getname(args[0])
    atypes: list[str] = [ check_term(a) for a in args[1:] ]
    sig: Sig = tuple([ pname ] + atypes)
    check(ispred(sig), "unknown predicate or ill-typed application", formula)

def check_term(term: ExpArg)->str:
    check(isinstance(term, Exp), "illformed term", term)
    args: List[ExpArg] = term.args
    n: int = len(args)
    match term.tag:
        case 'var':
            check(n == 1, "illformed variable occurrence", term)
            vname: str = getname(args[0])
            tname = vtype(vname)
            check(tname is not None, "undeclared variable", term)
            return tname
        case 'fun':
            check(n >= 1, "illformed function application", term)
            fname: str = getname(args[0])
            atypes: list[str] = [ check_term(a) for a in args[1:] ]
            sig: Sig = tuple([ fname ] + atypes)
            check(isfun(sig), "unknown function or ill-typed application", term)
```

```
            rtype: str = ftype(sig)
            return rtype
        case _:
            check(false, "unknown kind of term", term)

def vtype(name: str)->(str|None):
    global venv
    for vname,tname in reversed(venv):
        if vname == name:
            return tname;
    return None

def ftype(sig: Sig)->str:
    global fenv
    return fenv[sig]

if __name__ == '__main__':
    try:
        exp0: Exp = Exp('var', [ 'x' ])
        exp1: Exp = Exp('fun', [ 'c' ])
        exp2: Exp = Exp('fun', [ 'f' , exp0, exp1])
        exp3: Exp = Exp('pred', [ 'p', exp2 ])
        exp4: Exp = Exp('pred', [ '>', Exp('fun', [ 'd' ]), Exp('fun', [ 'd' ]) ])
        exp5: Exp = Exp('and', [ exp3, exp4 ])
        exp6: Exp = Exp('forall', [ 'x', 'T', exp5 ])
        idecl: Exp = Exp('typedecl', [ 'ℤ' ])
        tdecl: Exp = Exp('typedecl', [ 'T' ])
        cdecl: Exp = Exp('fundecl', [ 'c', 'T' ])
        ddecl: Exp = Exp('fundecl', [ 'd', 'ℤ' ])
        fdecl: Exp = Exp('fundecl', [ 'f', 'T', 'T', 'T' ])
        pdecl: Exp = Exp('preddecl', [ 'p', 'T' ])
        gdecl: Exp = Exp('preddecl', [ '>', 'ℤ', 'ℤ' ])
        goal: Exp = Exp('goal', [ 'G', exp6 ])
        problem: Exp = Exp('problem', [ 'P', idecl, tdecl, cdecl, ddecl, \
        fdecl, pdecl, gdecl, goal ])
        check_problem(problem)
        print('SUCCESS')
    except CheckException as e:
        print(problem)
        print(e)
```

C.4 RISCTP

`risctp.py` This file (220 lines, 0% vibe coded) implements the translation of FOL-EXP expressions to the syntax of the RISCTP theorem proving interface.

```
from exp import Exp, ExpArg

# -----------------------------------------------------------------------------
# conversion of expressions to RISCTP syntax
# -----------------------------------------------------------------------------

BUILTIN_TYPES: list[str] = [ 'Int', 'ℤ', 'Nat', 'ℕ']
```

```
UNARY_FUNOPS: list[str] = [ ’-’ ]
BINARY_FUNOPS: list[str] = [ ’+’, ’-’, ’*’, ’/’ ]
UNARY_PREDOPS: list[str] = [ ]
BINARY_PREDOPS: list[str] = [ ’=’, ’≠’, ’<’, ’<=’, ’≤’, ’>’, ’>=’, ’≥’ ]

UNDERSCORE: int = ord(’_’)
DIGIT_0: int = ord(’0’)
DIGIT_9: int = ord(’9’)
LOWER_A: int = ord(’a’)
LOWER_Z: int = ord(’z’)
UPPER_A: int = ord(’A’)
UPPER_Z: int = ord(’Z’)

def isdigit(code: int)->bool:
    return (DIGIT_0 <= code and code <= DIGIT_9)
def isletter(code: int)->bool:
    return (LOWER_A <= code and code <= LOWER_Z) or \
           (UPPER_A <= code and code <= UPPER_Z)

def idstr(name: str)->str:
    if plain(name):
      return name;
    return "’" + "".join([ clean(ch) for ch in name ]) + "’"
def plain(name: str)->bool:
    assert len(name) >= 1
    first: int = ord(name[0])
    if first != UNDERSCORE and not isletter(first):
      return False
    for ch in name[1:]:
      code: int = ord(ch)
      if (not isletter(code)) and (not isdigit(code)):
        return False
    return True
def clean(ch: str)->str:
    assert len(ch) == 1
    if ch in [ "’", "\\", "§" ]:
      return ’"’ + str(ord(ch)) + ’"’
    return ch

def varstr(name: str)->str:
    return idstr(’_’ + name)
def parens(text: str)->str:
    return ’(’ + text + ’)’

def problemstr(exp: Exp)->str:
    ’’’
    Return string representation of proof problem in RISCTP syntax
    ’’’
    assert exp.tag == ’problem’
    args: list[ExpArg] = exp.args
    n: int = len(args)
    assert n >= 1
    result : str = ’// === problem ’ + args[0] + " ===\n"
    for decl in args[1:]:
```

```
      result += declstr(decl)
    result += "// === end of problem ===\n"
    return result

def declstr(exp: Exp)->str:
    '''
    Return string representation of declaration in RISCTP syntax
    '''
    args: list[ExpArg] = exp.args
    n: int = len(args)
    assert n >= 1
    name: str = args[0]
    match exp.tag:
        case 'typedecl':
            result: str = '// ' if name in BUILTIN_TYPES else ''
            result += 'type ' + idstr(name) + ';\n'
            return result
        case 'fundecl':
            assert n >= 2;
            result: str = '// ' \
                if name in UNARY_FUNOPS or name in BINARY_FUNOPS else ''
            if n == 2:
                result += 'const ' + idstr(name) + ':' + idstr(args[1]) + ';\n'
            else:
                result += 'fun ' + idstr(name) + '(' +  sigstr(args[1:]) \
                        + '):' + idstr(args[n-1]) + ';\n'
            return result
        case 'preddecl':
            result: str = '// ' \
                if name in UNARY_PREDOPS or name in BINARY_PREDOPS else ''
            if n == 1:
                result += 'pred ' + idstr(name) + ';\n'
            else:
                result += 'pred ' + idstr(name) + '(' + sigstr(args[1:]) + ');\n'
            return result
        case 'axiom':
            assert n == 2
            return 'axiom ' + idstr(name) + ' ⇔ ' + formulastr(args[1]) + ';\n';
        case 'goal':
            assert n == 2
            return 'theorem ' + idstr(name) + ' ⇔ ' + formulastr(args[1]) + ';\n';
        case _:
            assert False, "unknown kind of declaration"

def sigstr(args: list[ExpArg])->str:
    n: int = len(args)
    result: str = ''
    for i, arg in enumerate(args):
        astr: str = arg
        result += 'x' + str(i) + ':' + idstr(astr)
        if i+1 < n:
            result += ','
    return result
```

```
def formulastr(exp: Exp)->str:
    '''
    Return string representation of formula in RISCTP syntax
    '''
    args: list[ExpArg] = exp.args
    n: int = len(args)
    match exp.tag:
        case 'true':
            return 'true'
        case 'false':
            return 'false'
        case 'pred':
            return atomicstr(exp)
        case 'not':
            assert n == 1
            return parens('¬' + formulastr(args[0]))
        case 'and':
            assert n == 2
            return parens(formulastr(args[0]) + '∧' + formulastr(args[1]))
        case 'or':
            assert n == 2
            return parens(formulastr(args[0]) + '∨' + formulastr(args[1]))
        case 'imp':
            assert n == 2
            return parens(formulastr(args[0]) + '⇒' + formulastr(args[1]))
        case 'iff':
            assert n == 2
            return parens(formulastr(args[0]) + '⇔' + formulastr(args[1]))
        case 'forall':
            assert n == 3
            vstr: str = varstr(args[0])
            tstr: str = idstr(args[1])
            fstr: str = formulastr(args[2])
            return parens('∀' + vstr + ':' + tstr + '.' + fstr)
        case 'exists':
            assert n == 3
            vstr: str = varstr(args[0])
            tstr: str = idstr(args[1])
            fstr: str = formulastr(args[2])
            return parens('∃' + vstr + ':' + tstr + '.' + fstr)
        case _:
            assert False, 'unknown kind of formula'

def atomicstr(exp: Exp)->str:
    '''
    Return string representation of atomic formula in RISCTP syntax
    '''
    args: list[ExpArg] = exp.args
    n: int = len(args)
    assert n >= 1
    name: str = args[0]
    assert exp.tag == 'pred'
    if n == 1:
        return name
```

```
    if n == 2 and name in UNARY_PREDOPS:
        arg1: str = termstr(args[1])
        return parens(name + arg1)
    if n == 3 and name in BINARY_PREDOPS:
        arg1: str = termstr(args[1])
        arg2: str = termstr(args[2])
        return parens(arg1 + name + arg2)
    result: str = name + ’(’;
    for i, arg in enumerate(args[1:]):
      result += termstr(arg)
      if i+2 < n:
        result += ’,’
    result += ’)’;
    return result

def termstr(exp: Exp)->str:
    ’’’
    Return string representation of term in RISCTP syntax
    ’’’
    args: list[ExpArg] = exp.args
    n: int = len(args)
    assert n >= 1
    name: str = args[0]
    if exp.tag == ’var’:
      assert n == 1
      return varstr(name)
    assert exp.tag == ’fun’
    if n == 1:
        return idstr(name)
    if n == 2 and name in UNARY_FUNOPS:
        arg1: str = termstr(args[1])
        return parens(name + arg1)
    if n == 3 and name in BINARY_FUNOPS:
        arg1: str = termstr(args[1])
        arg2: str = termstr(args[2])
        return parens(arg1 + name + arg2)
    result: str = name + ’(’;
    for i, arg in enumerate(args[1:]):
      result += termstr(arg)
      if i+2 < n:
        result += ’,’
    result += ’)’;
    return result

if __name__ == ’__main__’:
    ... (omitted)
    problem: Exp = Exp(’problem’, [ ’P’, idecl, tdecl, cdecl, ddecl, \
    fdecl, pdecl, gdecl, goal ])
    print(problemstr(problem))
```

`prover.py` This file (90 lines, 10% vibe coded) implements the interface to the RISCTP software.

```
from exp import Exp
from scanner import scan
from parser import parseProblem
from checker import check_problem
from risctp import problemstr
from executor import execute, start, stop

import tempfile, os

RISCTP_PATH='/software/bin/RISCTP'
Z3_PATH='/software/RISCTP/etc/z3'
BROWSER_PATH='/software/bin/firefox'

BROWSER_CMD = [BROWSER_PATH, '-no-remote', '-P', 'RISCTP', 'localhost:9999']

INTERACTIVE_CMD = ['-web', '9999', '1']
RISCTP_SMT_CMD = [RISCTP_PATH, \
  '-method', 'smt', '-solver', 'z3', '-path', Z3_PATH]
RISCTP_MESON_CMD = [RISCTP_PATH, '-method', 'meson']

RISCTP_CMD = RISCTP_MESON_CMD

def prove(fol_text:str,interactive:bool=False)->tuple[bool,str]:
    """
    Given the text of a proof problem in FOL-PRE syntax, apply RISCTP
    to the proof problem, either interactively or automatically
    (depending on the second parameter). In the automatic mode, the
    function returns 'True' if the proof could be performed and 'False' otherwise,
    in both cases together with the output produced by the prover.
    In the interactive mode, the result is (False,'').
    """
    result:str = ''
    try:
        tokens: list(str) = scan(fol_text)
        problem: Exp = parseProblem(tokens)
        check_problem(problem)
        risctp_text = problemstr(problem)
        spec_path = create_persistent_temp_file(risctp_text)
        risctp_cmd = RISCTP_CMD.copy()
        if interactive:
            risctp_cmd += INTERACTIVE_CMD
        risctp_cmd += [ spec_path ]
        if interactive:
            risctp_proc = start(risctp_cmd)
            if risctp_proc == None:
                return -1, 'could not start RISCTP process'
            print
            code,stdout = execute(BROWSER_CMD)
            stop(risctp_proc)
            return -1, 'proof session terminated (successfully or unsuccessfully)'
        else:
            code, stdout = execute(risctp_cmd)
            os.remove(spec_path)
            return (code == 0), stdout
```

```
    except ValueError as e:
        result = 'error in scanning/parsing text: ' + str(e)
    except CheckException as e:
        result = 'error in type-checking problem: ' + str(e)
    except _ as e:
        result = 'internal error: ' + str(e)
    return False, result


def create_persistent_temp_file(content:str)->str:
    # delete=False ensures the file stays on disk after .close()
    # mode='w+' allows us to write text and read if needed
    with tempfile.NamedTemporaryFile(mode='w+', delete=False, encoding='utf-8') as temp:
        temp.write(content)
        temp_path = temp.name
    return temp_path

'''
# --- Usage Example ---
file_path = create_persistent_temp_file("This is some secret text for later.")

# The file is closed now, but it still exists!
# You can pass 'file_path' to your 'execute' function or other methods.
with open(file_path, 'r') as f:
    print(f"Re-opening file: {f.read()}")

# IMPORTANT: Since delete=False, you are responsible for cleaning it up
# os.remove(file_path)
'''

if __name__ == '__main__':
    fol_text = """
... (omitted)
"""
    code, stdout = prove(fol_text,True)
    print(code)
    print(stdout)
```

`executor.py` This file (80 lines, 100% vibe coded) implements the interface between the Python program and external processes.

```
import subprocess
import signal
import os

def execute(command_args):
    """
    Executes an external command and returns its output.

    :param command_args: A list of strings (e.g., ["ls", "-l", "/tmp"])
    :return: The return code of the command and its standard output as a string.
    """
    try:
```

```
        # check=True raises an exception if the command fails
        # capture_output=True grabs stdout and stderr
        # text=True ensures the result is a string, not bytes
        result = subprocess.run(
            command_args,
            capture_output=True,
            text=True,
            check=True
        )
        return result.returncode, result.stdout

    except subprocess.CalledProcessError as e:
        return e.returncode, e.stdout
    except FileNotFoundError:
        return -1, "Error: The specified command was not found."

# Example Usage:
# output = execute(["echo", "Hello World"])
# print(output)

def start(command_args):
    """
    Starts an external command in the background.

    :param command_args: List of strings, e.g., ["python", "script.py"]
    :return: A subprocess.Popen object (the handle for the process).
    """
    try:
        # We use Popen to start the process without blocking
        process = subprocess.Popen(
            command_args,
            stdout=subprocess.PIPE,  # Optional: capture output
            stderr=subprocess.PIPE,  # Optional: capture errors
            text=True
        )
        #print(f"Process started with PID: {process.pid}")
        return process
    except Exception as e:
        print(f"Failed to start process: {e}")
        return None

def stop(process):
    """
    Terminates the background process.

    :param process: The subprocess.Popen object returned by start().
    """
    if process is None:
        print("No process to stop.")
        return

    # Check if the process is still running
    if process.poll() is None:
        #print(f"Terminating process {process.pid}...")
```

```
    process.terminate()  # Sends SIGTERM (polite request to stop)

    try:
        # Wait up to 5 seconds for it to shut down gracefully
        process.wait(timeout=5)
        # print("Process stopped gracefully.")
    except subprocess.TimeoutExpired:
        # If it doesn’t stop, kill it forcefully
        print("Process did not stop; forcing kill...")
        process.kill()  # Sends SIGKILL (immediate termination)
else:
    print("Process had already finished.")
```