Logic Agents

Intelligent Systems II

Mário Antunes

2026

Practice Guide: Building Logic Agents with Prolog and PySwip

In this guide, you will transition from basic logical programming in Prolog to building hybrid AI systems where Python handles the environment (game states, physics, UI) and Prolog acts as the "brain" of your agent making logical decisions.

The architecture for the later exercises is split into two parts:

1. **Backend/Environment (Python):** Runs game logic, UI, or socket servers.
2. **Agent Logic (Prolog via PySwip):** Evaluates the environment state and infers the best action.

Part 1: Prolog Warm-Up Exercises

Before building autonomous agents, ensure you are comfortable with Prolog's unification, recursion, and search. Create a file named `warmup.pl` and write Prolog facts and rules to solve the following 10 exercises.

Facts and Rules

1. **Family Facts:** Define a family tree using the `parent(Parent, Child)` fact. Include at least 4 generations (e.g., great-grandparent down to child).
2. **Basic Rules:** Write rules for `sibling(X, Y)` and `grandparent(X, Y)` using your `parent` facts.
3. **Recursive Rules:** Write an `ancestor(X, Y)` rule that finds if `X` is an ancestor of `Y` (this requires a base case and a recursive step).

4. **Arithmetic:** Write a rule `factorial(N, Result)` that calculates the factorial of a given number N.

List Processing Prolog uses the `[Head|Tail]` syntax heavily for state processing.

5. **Length:** Write a rule `list_length(List, Length)` that computes the number of elements in a list.
6. **Membership:** Write a rule `is_member(Element, List)` that succeeds if `Element` is inside `List`.
7. **Concatenation:** Write a rule `concat_lists(List1, List2, Result)` that joins two lists together.
8. **Reverse:** Write a rule `reverse_list(List, Reversed)` that reverses the order of a list.
9. **Maximum:** Write a rule `max_in_list(List, Max)` that finds the largest number in a list of integers.

Search and Graphs

10. **Pathfinding:** Given a set of facts representing directed edges in a graph (e.g., `edge(a, b). edge(b, c).`), write a rule `path(Start, End)` that determines if there is a valid path between two nodes.

Part 2: A Simple Agent - The Tower of Hanoi

The Tower of Hanoi is a classic planning problem. An agent must move n disks from a source peg to a target peg using an auxiliary peg, without ever placing a larger disk on a smaller one.

1. First-Order Logic Derivation

Before writing the code, we define the problem in First-Order Logic (FOL). Let $Move(n, S, T, A)$ represent the action of moving n disks from Source (S) to Target (T) using Auxiliary (A).

Base Case: Moving 1 disk is a direct action.

$$\forall S, T, A \, (Move(1, S, T, A) \implies \text{Action: Move top disk from } S \text{ to } T)$$

Recursive Step: To move n disks, the agent must move $n - 1$ disks out of the

way, move the largest disk, and then move the $n-1$ disks back on top.

$$\begin{aligned}\forall n > 1, S, T, A\,(Move(n, S, T, A) \implies \\ Move(n-1, S, A, T) \wedge \\ Move(1, S, T, A) \wedge \\ Move(n-1, A, T, S)\,)\end{aligned}$$

2. Prolog Implementation

Your Task: Translate the FOL derivation directly into a Prolog file named `hanoi.pl`. Write a rule `move(N, Source, Target, Auxiliary)` that prints the sequence of moves required to solve the puzzle. *Test* your code with: `?- move(3, left, right, center).`

Part 3: Python/Prolog Integration (Tic-Tac-Toe)

Now we introduce **PySwip**. Python will manage the Tic-Tac-Toe board, and Prolog will decide the next move.

1. Setup PySwip

Ensure you have SWI-Prolog installed on your system, then install the Python library:

```
pip install pyswip
```

2. Python Backend (`play_ttt.py`)

Here is the Python skeleton. It sets up the board and queries the Prolog agent for the best move.

```
from pyswip import Prolog

prolog = Prolog()
prolog.consult("tictactoe.pl")

# Represent board as indices 0-8. 'e' is empty.
board = ['x', 'e', 'o',
         'e', 'x', 'e',
```

```
        'o', 'e', 'e']

prolog_board = "[" + ",".join(board) + "]"
query = f"best_move(x, {prolog_board}, Move)"
result = list(prolog.query(query))

if result:
    print(f"The Prolog agent chooses index: {result[0]['Move']}")
else:
    print("No moves available.")
```

3. Prolog Agent Logic (`tictactoe.pl`)

Your Task: Create `tictactoe.pl`.

1. Define the winning lines (rows, columns, diagonals) using `line(Idx1, Idx2, Idx3).`
2. Write a rule `win_move(Player, Board, Index)` that finds an empty spot (`e`) to complete a line of 3 for the given player.
3. Write a fallback rule `first_empty(Board, Index)` that just finds the first available spot if no winning move is possible.
4. Combine them in `best_move(Player, Board, Move).` (Hint: Use the cut operator `!` to stop searching once a valid move is found).

Part 4: The Flappy Bird Logic Agent

Instead of Neuro-evolution, you will build a **Reflex Logic Agent** for Flappy Bird. Python runs the game and sends the continuous state via websockets; Prolog evaluates the state and decides if the bird should jump.

1. Python Communication Loop

Integrate PySwip into the game's websocket listener:

```
import json
from pyswip import Prolog

prolog = Prolog()
prolog.consult("flappy_agent.pl")
```

```python
def process_state(state_json):
    state = json.loads(state_json)
    bird_y = state['bird_y']
    pipe_y = state['next_pipe_bottom_y']

    # Assert current state
    list(prolog.query(f"update_state({bird_y}, {pipe_y})"))

    # Query agent action
    result = list(prolog.query("action(A)"))
    return result[0]['A'] if result else 'stay'
```

2. Prolog Logic (`flappy_agent.pl`)

Your Task:

1. Declare dynamic predicates for `bird_y/1` and `next_pipe_bottom_y/1`.
2. Write an `update_state(BirdY, PipeY)` rule that retracts the old knowledge and asserts the new coordinates.
3. Write `action(jump)` and `action(stay)` rules. The agent should `jump` if the bird's Y coordinate is dangerously close to the bottom pipe.

3. Experiments

- **Experiment A:** Does a hardcoded threshold (e.g., jumping when `BirdY > PipeY + 40`) work for all pipe speeds?
- **Experiment B:** Modify the Python script to send the bird's `velocity`. Update your Prolog agent to use this new dynamic fact. Can you write a rule that prevents the agent from jumping if its upward velocity is already high enough to clear the pipe?