Embodied Declarative Logic: Learning Logica via Orchestrating Robot Behavior

Evgeny Skvortsov[1], Ojaswa Garg[1], Yilin Xia[2], Shawn Bowers[3] and Bertram Ludäscher[2]

[1]Google LLC, USA
[2]University of Illinois, Urbana-Champaign, USA
[3]Gonzaga University, USA

Abstract

Learning to program is challenging regardless of paradigm. Logic programming offers a particularly natural approach: the programmer describes *what* should hold rather than *how* to compute it. But like any skill, it requires practice with concrete feedback. This paper presents the *Logical Robots* platform [9] as a tool for teaching Logica, an open-source logic programming language that compiles to SQL. The platform comes with a combined view of the logic program and the visual simulation environment where robots act. Users can start, pause, and restart simulations as well as drag robots within the environment to create debugging scenarios interactively. In Logical Robots, autonomous agents in this simulated world are controlled by logical predicates that determine their immediate reaction to sensory data. By starting the simulation, the student can immediately see how logical rules drive the robot's behavior. Learning to program through this feedback loop of writing a rule and seeing the robot act, is yet another way an instructor can make learning logic programming more engaging.

Keywords

Logic Programming, Multi-Agent Systems, Declarative Programming, Robot Simulation

1. Introduction

Learning to program is hard. Every paradigm—imperative, functional, object-oriented—presents its own conceptual hurdles, and every generation of educators searches for better ways to make programming accessible [7].

Logic programming has always offered a compelling answer: describe *what* should hold, not *how* to compute it. From Prolog to Datalog and beyond, the logic programming tradition shows that complex problems—graph analysis, data transformation, constraint satisfaction—can be expressed as concise logical rules. Logica [11, 10] continues this tradition: an open-source language that extends Datalog with aggregation, functions, and modules, compiling to SQL for execution on standard database engines. In logic programming, a few well-chosen rules can replace pages of imperative code.

But even in this natural paradigm, beginners need practice with concrete, immediate feedback. Papert's Constructionism [5] showed that learning is most effective when students construct a tangible artifact and observe the consequences of their choices. Karel the Robot [6] applied this insight to imperative programming—students commanded a robot on a grid and immediately saw the effect of each instruction. While a number of systems use logic for programming robots [3, 2, 1], a system that would aim to teach logic programming via robotics has been missing.

The Logical Robots platform [9] fills this gap. In this environment, every Logica rule controls an autonomous agent in a simulated world. A `WeightedAverage` aggregation steers the robot away from walls; a `Min=` aggregation finds the shortest path home; a flawed rule makes it crash. The feedback loop—write a rule, see the robot act, fix the logic—is immediate and physical. Instead of commanding the robot step by step, the student defines its world through predicates and rules, and the robot's behavior *emerges from the logic*.

This design also carries a deeper consequence. In Logical Robots, each simulation tick runs a non-recursive Logica program that compiles to a single SQL query—placing it in NC (polylogarithmic

PEG 2026: Prolog Education Workshop



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-11-03

CEUR-WS.org/Vol-4311/PEG_regular_12.pdf

parallel time). Students write efficiently parallelizable code as a natural byproduct of writing declarative rules, without ever hearing the term.

This paper presents the platform with a complete working program, showing how Logica's abstractions—predicates, aggregation, and functional composition—translate directly into robot behavior, and describes the progression of scenarios through which students learn the language.

2. Logica

Logica [11, 10] is an open-source logic programming language developed at Google. Logica programs compile to SQL and run on standard database engines—DuckDB, SQLite, PostgreSQL, and BigQuery [8]—giving logic programs access to the full computational infrastructure of modern data platforms.

A Logica program is a collection of *facts* and *rules*. A fact states that a predicate holds for given values. A rule derives new facts from existing ones using the syntax *head* `:-` *body*, read as "the head holds whenever the body is satisfied." For example:

```
Parent("Adam", "Cain");
Parent("Cain", "Enoch");
Grandparent(x, z) :- Parent(x, y), Parent(y, z);
```

Here `Parent` is defined by two facts, and `Grandparent` is derived by a rule that chains two `Parent` relationships. Logica extends this Datalog core with several features that improve expressiveness and readability.

Named Arguments. When predicates have many columns, positional arguments become hard to read. Logica supports *named arguments*, so that only the relevant columns need to be mentioned:

```
Human(name: "Socrates", iq: 250, yr: -470, loc: "Athens");
AthenianPhilosopher(name:) :-
  Human(name:, loc: "Athens", iq:), iq > 200;
```

The query for `AthenianPhilosopher` references only the columns it needs; the remaining columns (`yr`) are left unconstrained. When a variable name matches the argument name, Logica allows the shorthand `name:` in place of `name: name`.

Functional Notation. In Logica, functions are predicates with an extra column called `logica_value`. The syntax `F(x) = y` is shorthand for a predicate where `x` is an argument and `y` is the value. Because functions are predicates, they can naturally be multi-valued (returning several results for the same input) or return no value at all, in which case they act as a filter. Functions compose just like in mathematics:

```
Square(x) = x * x;
Double(x) = 2 * x;
Result(Double(Square(x))) :- Input(x);
```

Because functions are predicates, a function may return multiple values or no values at all:

```
Parent(x) = p :- Father(x, p) | Mother(x, p);
Neighbor(x) = y :- Edge(x, y);
```

Here `Parent("Alice")` returns both parents, and `Neighbor(x)` returns all adjacent nodes—behavior that is natural for predicates but impossible for conventional functions. For an isolated vertex with no edges, `Neighbor` returns nothing, acting as a filter.

Aggregation. Aggregation is central to data analysis, and Logica integrates it directly into the language. Predicate-level aggregation uses the `distinct` keyword together with aggregation operators like `+=` (sum/count), `Max=`, and `Min=`:

```
AuthorStats(author:,
        book_count? += 1,
        latest_year? Max= year) distinct :-
  Book(author:, year:);
```

This computes, for each author, the number of books and the most recent publication year. Columns marked with `?` are aggregated; the others serve as grouping keys.

Logica also supports *inline* aggregation, which allows an aggregate to be used as an expression anywhere in a rule. The syntax resembles a multiset builder: the programmer specifies which elements to collect and which aggregation function to apply to collapse the multiset into a single value. For example, the total salary budget for a department can be written as:

```
DeptBudget(dept) = Sum{salary :- Employee(dept:, salary:)};
```

The expression `Sum{salary :- ...}` collects all matching salaries into a multiset and sums them. Replacing `Sum` with `List` would keep the full multiset; replacing it with `Max` would select the largest value. The choice of aggregation function determines the semantics of the collapse, not the volume of data collected. This inline style plays a central role in the Logical Robots scenarios described in Section 3.

Recursion and Time. Logica supports recursive predicates, evaluated iteratively until a fixpoint is reached. However, Logical Robots does not use Logica's built-in recursion: the robot's program must execute in real time at every simulation tick, and iterating to a fixpoint would be too slow.

Instead, recursion is *unrolled across time*. The robot maintains a persistent `memory` between ticks. At each tick, a non-recursive Logica program reads the previous state from memory, performs one step of computation, and writes the updated state back. Because each tick's program is non-recursive, it compiles to a single SQL query, evaluable in NC (polylogarithmic parallel time). We illustrate this concretely in Section 3.2, where robots use memory to implement a distributed shortest-path algorithm.

3. Logical Robots

In Logical Robots [9], the student writes a Logica program that controls autonomous agents in a simulated two-dimensional world. The simulation runs in real time: at every tick, the platform provides each robot with sensor data, and the student's program processes this data to produce a steering command. The interface is a single predicate:

```
Robot(robot_name:, desire:, memory:) :-
  Sensor(robot_name:, sensor:), ...
```

The `Sensor` record contains the robot's `radar`—an array of rays, each with a `distance`, `angle`, and `object` type—as well as `forward_distance`, a `clock`, and other readings. The program computes a `desire` record with `left_engine` and `right_engine` fields that control differential steering, and optionally a `memory` string that persists to the next tick. This is the entire contract: everything the robot perceives comes through `Sensor`; everything it does flows from `Robot`.

3.1. A Complete Program: Obstacle Avoidance

The simplest scenario places a single robot in an open space with irregular walls (Figure 1). The goal: avoid collisions and keep moving. Here is the complete program:

```
WeightedAverage(k -> v) = Sum(k * v) / Sum(k);

FreedomMotion(radar) = WeightedAverage{
  x.distance -> x.angle :- x in radar
};

Robot(robot_name:, desire:, memory: "I am") :-
  Sensor(robot_name:, sensor:),
  freedom = FreedomMotion(sensor.radar),
  speed = 0.5,
  desire = {
    left_engine: speed - freedom + 0.1,
```

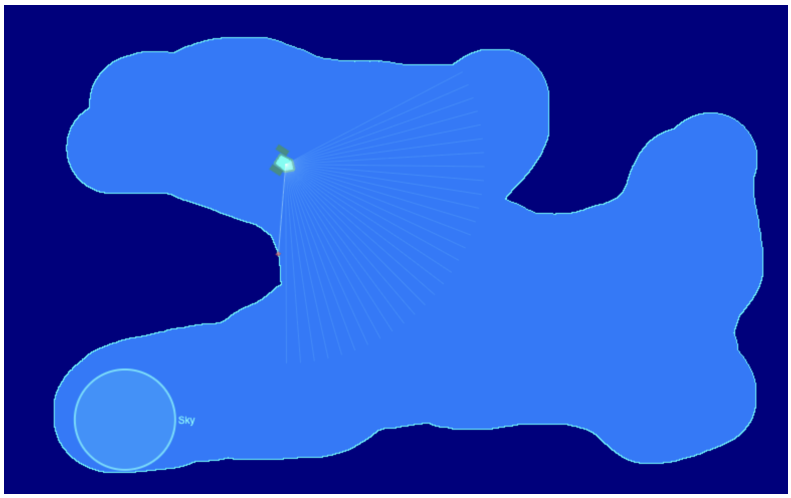


Figure 1: The obstacle avoidance scenario. Radar rays fan out from the robot; longer rays indicate open space. The `FreedomMotion` function computes a weighted average of ray angles, steering the robot toward the direction with the most room.

```
    right_engine: speed + freedom
  };
```

The function `WeightedAverage` is defined using inline aggregation (Section 2): it takes weighted votes and returns their average. `FreedomMotion` applies it to the robot's radar: each ray votes for its angle, weighted by how far it reaches before hitting a wall. Rays into open space carry more weight, so the result is an angle pointing toward freedom.

The `Robot` rule reads sensor data, computes the freedom angle, and outputs a steering command. If `freedom` is positive (open space to the right), the right engine gets more power and the robot turns right. If the student accidentally flips a sign, the robot steers *into* walls—an error that is visible within seconds. This immediate, tangible feedback [4] makes the connection between a logical mistake and its physical consequence unmistakable.

The `memory` field is unused here (the robot simply remembers `I am`), but it becomes essential in advanced scenarios where the robot must carry information between ticks.

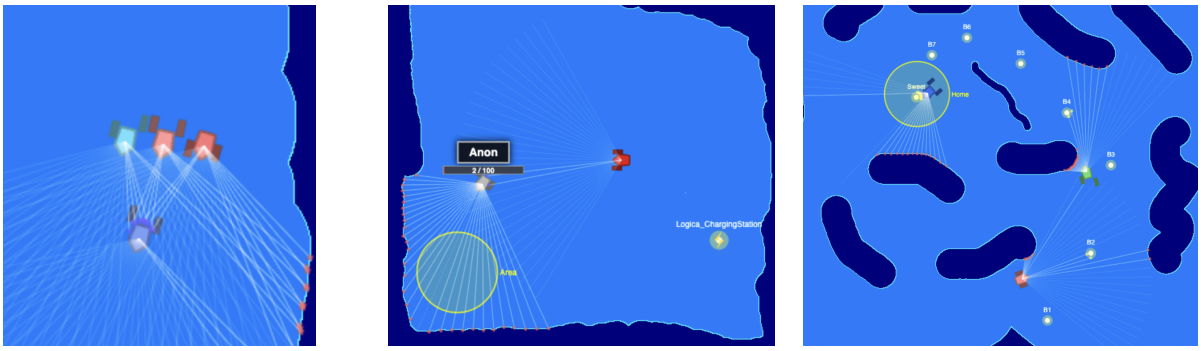
3.2. A Progression of Scenarios

The platform currently includes twelve levels. They explore many aspects of declarative robot programming: defining elegant behavior, organizing multi-robot coordination, assigning different roles to robots that share the same code, and teaching robots to build models of their environment. Here we present four levels that illustrate this progression. The game is available at logica.dev/robots, and complete solutions for all levels discussed below can be found at github.com/EvgSkv/logica/tree/main/examples/peg2026-robots. We highlight key code fragments in the text; the reader is encouraged to explore and play all twelve levels.

Every level takes place in a world with walls, beacons (named landmarks visible on radar), and designated areas. The level author specifies a victory condition—typically, certain robots must reach a designated area. In the levels below, robots achieve victory by navigating toward beacons or by coordinated wandering.

Formation (Level 8). The victory zone is large enough to find by wandering, but all robots must be inside it simultaneously—so they need to stay in tight formation. Each robot sees the others on its radar and knows their names. The solution: every robot identifies the leader—the one with the lexicographically smallest name—using `ArgMin`, and steers toward it:

```
leader_angle = Coalesce(
 ArgMin{ x.angle -> x.distance :-
```



(a) Formation (Level 8) (b) Charger (Level 12) (c) Beacon Navigation (Level 6)

Figure 2: Three scenarios: robots in formation behind a leader (left), a robot with low energy near a charging station (center), and navigating a beacon trail to "Sweet Home" (right).

```
  x in sensor.radar,
  x.object = "robot",
  IsLeader(x.label) == true }, 0);
```

Each robot has its own preferred angle relative to the leader, determined by its index in the group: robot 1 steers slightly to the left, robot 2 slightly to the right, and so on. This prevents them from piling up and produces the spread formation visible in Figure 2a. The same program runs independently in each robot, yet coordinated group behavior emerges from the individual rules.

Charger (Level 12). A cargo robot does not have enough energy to reach its destination. A charger robot must shuttle between the charging station (itself a beacon) and the cargo robot, delivering fuel. The key insight is that the `FreedomMotion` function from the previous section can be parameterized with a target:

```
Destination(sensor) = (
  if sensor.self.can_charge &&
    sensor.self.battery_level > 50 then "robot"
  else if sensor.self.can_charge then "beacon"
  else "happiness");

FreedomMotion(radar, destination) = WeightedAverage{
  x.distance * desire_factor -> x.angle :-
  x in radar,
  desire_factor == (if x.object == destination
              then 10.0 else 1.0)};
```

Three lines of conditional logic turn undirected wandering into purposeful behavior. When the charger's battery is full, it seeks the cargo robot; when low, it returns to the station. The `memory` field records the current goal—`"Looking for robot"` or `"Looking for beacon"`—making the robot's decision visible.

Beacon Navigation (Level 6). A simple labyrinth with a trail of beacons leading to "Sweet Home." The beacons are arranged so that following the lexicographically largest visible beacon always leads closer to the goal. The modification to `FreedomMotion` is minimal—find the best beacon and weight it heavily:

```
FreedomMotion(radar, best_beacon:) = WeightedAverage{
  x.distance * desire_factor -> x.angle :-
  x in radar,
  desire_factor == (if x.label == best_beacon
              then 20.0 else 1.0)
} :-
  best_beacon = Max{b.label :-
    b in radar, b.object == "beacon"};
```

No memory is needed—the robot navigates using only what it currently sees. This level and the previous one show how a small modification to the wandering program—adding a single aggregation—creates directed navigation.

Labyrinth (Level 10). The most challenging level we have developed (Figure 3). A complex maze with dozens of beacons, but unlike the previous level, there is no predetermined path—the beacons are not arranged in any helpful order. Robots must collaboratively build a shared map of the maze and compute shortest paths home.

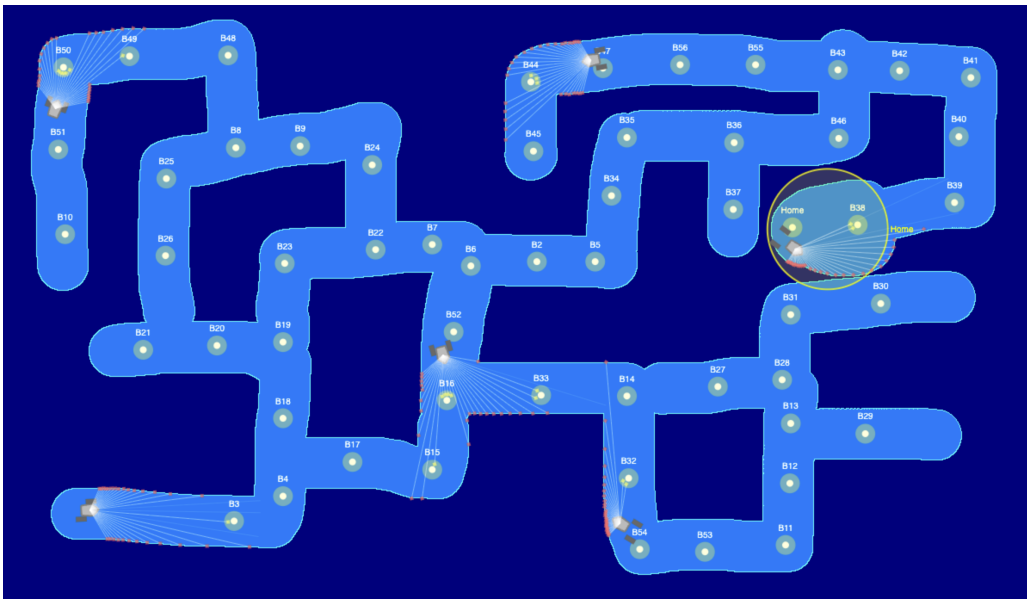


Figure 3: The full labyrinth: over fifty beacons, multiple robots collaboratively computing shortest paths home.

This is where `memory` becomes essential, and where we see recursion unrolled across time (Section 2). One robot serves as the leader, storing the shared map—a graph of beacon-to-beacon distances and a heatmap of distances to home—in its `memory`. Robots in Logical Robots are "telepaths": any robot can read any other's memory. The student decides whether to exploit this; at this level, it is essential, so that all robots contribute their observations to the leader's map.

At each tick, robots observe which beacons are simultaneously visible and estimate distances between them using the law of cosines. These observations are merged with the stored graph, and a single Bellman-Ford relaxation step updates the distance estimates:

```
UpdatedHeatmap(beacon) Min= d :-
  StoredHeatmap(beacon:, d:) |
  (beacon == "Home", d == 0) |
  (StoredHeatmap(beacon: neighbor, d: d_neighbor),
   Graph(neighbor, beacon) == edge_weight,
   d = d_neighbor + edge_weight);
```

Each tick performs one relaxation. Over hundreds of ticks, the heatmap converges to the true shortest distances—exactly as in the distributed Bellman-Ford algorithm, but with each iteration happening naturally as the robot moves through the world. Each robot then navigates toward the visible beacon closest to home:

```
LovelyBeacon(robot_name) ArgMin= beacon -> distance :-
  beacon in RobotSeesBeacons(robot_name),
  UpdatedHeatmap(beacon) == distance;
```

The full program is approximately 200 lines—substantial, but built entirely from concepts introduced in the earlier levels: `WeightedAverage` for steering, `ArgMin` for choosing targets, conditional logic for state management, and `Min=` for shortest-path computation.

4. Conclusion

Learning a declarative language like Logica requires building a new mental model for programming—one based on defining logical relationships rather than commanding sequential steps. Logical Robots [9] is a promising tool for building this model. By mapping Logica's abstract concepts to the tangible actions of autonomous agents, it provides a "body" for the disembodied logic of the code. The intuitive connection between a `WeightedAverage` and dodging a wall, or between a recursive predicate and finding the way home, makes the learning process concrete and engaging. Complemented by immediate visual feedback and a scaffolded, game-like progression, Logical Robots suggests that embodied simulation can be an effective way to teach abstract logic.

Perhaps most importantly, the game reveals something that tables and queries alone cannot: that discrete logical rules, when applied tick by tick, give rise to continuous behavior—smooth motion, gradual convergence, emergent navigation. For a student who has already learned to think in relations and aggregations, this is the moment when logic stops being about data and starts being about the world.

Declaration on Generative AI

During the preparation of this work, the authors used Gemini (Google), Claude (Anthropic), and ChatGPT (OpenAI) for grammar and spelling checks and to reword portions of the text. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using Jason.* John Wiley & Sons, 2007.

[2] Giuseppe De Giacomo, Yves Lespérance, Hector J. Levesque, and Sebastian Sardina. IndiGolog: A high-level programming language for embedded reasoning agents. *Multi-Agent Programming*, pages 31–72, 2009.

[3] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. volume 31, pages 59–83, 1997.

[4] Fikile Mposula and Kayode Oyetade. Enhancing Programming Education through Game-Based Learning for Improved Engagement and Learning Outcomes. *Educational Process: International Journal*, 2025.

[5] Seymour Papert. *Mindstorms: Children, Computers, and Powerful Ideas.* Basic Books, 1980.

[6] Richard E. Pattis. *Karel the Robot: A Gentle Introduction to the Art of Programming.* John Wiley & Sons, 1981.

[7] Anthony Robins, Janet Rountree, and Nathan Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137–172, 2003. doi: 10.1076/csed.13.2.137.14200.

[8] E. Skvortsov. Full support of PostgreSQL engine comes to Logica. Google Open Source Blog, September 2023.

[9] E. Skvortsov, Y. Xia, O. Garg, S. Bowers, and B. Ludäscher. Logical robots: Declarative multi-agent programming in Logica, 2026. To appear in International Conference on Autonomous Agents and Multiagent Systems (AAMAS).

[10] E. S. Skvortsov, Y. Xia, S. Bowers, and B. Ludäscher. The Logica System: Elevating SQL Databases to Declarative Data Science Engines. In *Datalog-2.0*, volume 3801 of *CEUR Workshop Proceedings*, pages 69–73, 2024.

[11] E. S. Skvortsov, Y. Xia, and B. Ludäscher. Logica: Declarative Data Science for Mere Mortals. In *EDBT*, pages 842–845, 2024.