Introduction

Logic programming, and Prolog in particular, solves problems that are naturally described as relations, constraints, and search. Instead of instructing the computer how to perform steps, the programmer states facts and rules; the Prolog engine then uses unification, backtracking, and constraint solving to find solutions. This discussion identifies the classes of computing problems where Prolog is most efficient, examines a concrete application—university timetabling—and explains the language features that make Prolog especially well-suited for that problem.

Types of problems best solved with Prolog

Prolog excels when a problem combines these elements: rich relational structure, nondeterministic search, logical inference, and explicit constraints. Typical domains include knowledge representation and expert systems, symbolic reasoning and theorem proving, natural language parsing (via definite clause grammars), and combinatorial constraint problems such as scheduling, configuration, and resource allocation. The Constraint Logic Programming (CLP) extension—which integrates constraint solving into logic programming—pushes Prolog into many optimization and scheduling tasks where pruning the search space with domain-specific constraints is essential. The survey literature frames CLP as the fusion that gives logic programming practical power for a wide class of constraint-heavy problems (Jaffar & Maher, 1994).

Specific application: university timetabling

University course timetabling is a clear, well-studied example. The problem requires assigning courses to time slots and rooms while satisfying hard constraints (no student or teacher conflicts, room capacities, no overlapping assignments) and optimizing soft constraints (preferred times, compact schedules). Researchers have successfully modeled timetabling directly in Prolog using CLP(FD) (Constraint Logic Programming over Finite Domains). In these implementations the timetable variables are given finite domains (possible time slots, rooms), and constraints such as non-overlap, cardinality, and serialized tasks are posted so the solver incrementally reduces domains and prunes infeasible search branches—often yielding practical solutions with less custom search code than imperative approaches (Rudová et al., 2003).

What Prolog features make it a good fit

Several features of Prolog and modern Prolog systems are critical:

• Declarative rules and facts. The problem is expressed as relations and constraints rather than control flow, which maps directly to human descriptions of timetabling requirements. This makes models easier to write and maintain (Donaldson, 2019).

• Unification and pattern matching. Prolog’s built-in unification simplifies matching complex structures (for example, student-course enrollments or room attributes) and propagates variable bindings during the search.

• Backtracking and search control. Prolog naturally supports nondeterministic exploration; combined with labeling heuristics and explicit search strategies, it finds assignments that satisfy all constraints.

• Constraint libraries (CLP(FD)). Modern Prolog implementations include efficient finite-domain constraint solvers that propagate constraints and prune domains before search, drastically reducing the search space for scheduling problems. Libraries support global constraints (e.g., all_different, disjoint2) which capture common timetable constraints succinctly and efficiently (SWI-Prolog, n.d.).

• Extensible I/O and integration. Systems like SWI-Prolog include interfaces to databases, web formats, and GUIs, making it practical to integrate timetabling solutions with university information systems (SWI-Prolog, n.d.).

Implementation outline

1. Encode facts: students, courses, teachers, rooms, and allowed slots.
2. Declare variables: for each course create variables for time slot and room with finite domains.
3. Post hard constraints: use all_different or disjoint constraints to prevent conflicts; enforce capacity and availability limits.
4. Add soft constraints as optimization criteria or penalty terms.
5. Use a labeling strategy (heuristics) to guide search and call the CLP solver to find feasible or optimal assignments.
6. Validate and, if needed, iterate by refining constraints or heuristic order.

Conclusion

Problems that combine relational structure, heavy constraints, and the need for nondeterministic search are natural candidates for Prolog and its CLP extensions. University timetabling is a representative case: the mapping from requirements to constraints is direct, constraint propagation reduces search complexity, and Prolog’s declarative style makes the solution compact and maintainable. For scheduling and other combinatorial constraint problems, Prolog often yields clearer implementations and competitive performance compared with general-purpose imperative languages (Jaffar & Maher, 1994).

Word count: *640*

References

Donaldson, T. (2019). Introduction to Prolog. Retrieved October 25, 2025, from https://www.sfu.ca/~tjd/383summer2019/prolog_intro.html

Jaffar, J., & Maher, M. J. (1994). *Constraint logic programming:* A survey. *Journal of Logic Programming*, 19–20, 503–581. https://courses.grainger.illinois.edu/cs522/sp2016/ConstraintLogicProgrammingASurvey.pdf

Rudová, H., et al. (2003). *University course timetabling with soft constraints* (Unitime / research paper). Retrieved from https://www.unitime.org/papers/patat03.pdf.

SWI-Prolog. (n.d.). *CLP(FD): Constraint Logic Programming over Finite Domains* (documentation). Retrieved from https://www.swi-prolog.org/pldoc/man?section=clpfd.

SWI-Prolog. (n.d.). *Features*. Retrieved from https://www.swi-prolog.org/features.html.