PART II: Programming in Prolog

The only way to rectify our reasonings is to make them as tangible as those of the mathematicians, so that we can find our error at a glance, and when there are disputes among persons we can simply say, "Let us calculate… to see who is right."

—*Leibniz,* The Art of Discovery

As an implementation of logic programming, Prolog makes many important contributions to AI problem solving. First and foremost, is its direct and transparent representation and interpretation of predicate calculus expressions. The predicate calculus has been an important representational scheme in AI from the beginning, used everywhere from automated reasoning to robotics research. A second contribution to AI is the ability to create meta-predicates or predicates that can constrain, manipulate, and interpret other predicates. This makes Prolog ideal for creating meta-interpreters or interpreters written in Prolog that can interpret subsets of Prolog code. We will do this many times in the following chapters, writing interpreters for expert rule systems, exshell, interpreters for machine learning using version space search and explanation based learning models, and deterministic and stochastic natural language parsers.

Most importantly Prolog has a *declarative semantics*, a means of directly expressing problem relationships in AI. Prolog also has built-in unification, some high- powered techniques for pattern matching and a depth-first left to right search. For a full description of Prolog representation, unification, and search as well as Prolog interpreter compared to an automated theorem prover, we recommend Luger (2009, Section 14.3) or references mentioned in Chapter 10. We will also address many of the important issues of Prolog and logic programming for artificial intelligence applications in the chapters that make up Part II.

In Chapter 2 we present the basic Prolog syntax and several simple programs. These programs demonstrate the use of the predicate calculus as a representation language. We show how to monitor the Prolog environment and demonstrate the use of the *cut* with Prolog's built in depth-first left-to-right search. We also present simple structured representations including semantic nets and frames and present a simple recursive algorithm that implements inheritance search.

In Chapter 3 we create *abstract data types* (ADTs) in Prolog. These ADTs include *stacks*, *queues*, *priority queues,* and *sets*. These data types are the basis for many of the search and control algorithms in the remainder of Part II.

In particular, they are used to build a *production system* in Chapter 4, which can perform *depth-first*, *breadth-first*, and *best-first* or *heuristic* search. They also are critical to algorithms later in Part II including building planners, parsers, and algorithms for machine learning.

In Chapter 5 we begin to present the family of design patterns expressed through building *meta-interpreters*. But first we consider a number of important Prolog *meta-predicates*, predicates whose domains of interpretation are Prolog expressions themselves. For example, atom(X) succeeds if X is bound to an atom, that is if X is instantiated at the time of the atom(X) test. Meta-predicates may also be used for imposing type constraints on Prolog interpretations, and we present a small database that enforces Prolog typing constraints.

In Chapter 6 meta-predicates are used for designing *meta-interpreters* in Prolog. We begin by building a Prolog interpreter in Prolog. We extend this interpreter to rule-based expert system processing with exshell and then build a robot planner using *add-* and *delete-lists* along the lines of the older STRIPS problem solver (Fikes and Nilsson 1972, Nilsson 1980).

In Chapter 7 we demonstrate Prolog as a language for machine learning, with the design of meta-interpreters for *version space search* and *explanation-based learning*. In Chapter 8 we build a number of natural language parsers/generators in Prolog, including context-free, context-sensitive, probabilistic, and a recursive descent semantic net parser.

In Chapter 9 we present the Earley parser, a form of *chart parsing*, an important contribution to interpreting natural language structures. The Earley algorithm is built on ideas from dynamic programming (Luger 2009, Section 4.1.2 and 15.2.2) where the chart captures sub-parse components as they are generated while the algorithm moves across the words of the sentence. Possible parses of the sentence are retrieved from the chart after completion of its left-to-right generation of the chart.

Part II ends with Chapter 10 where we return to the discussion of the general issues of programming in logic, the design of meta-interpreters, and issues related to procedural versus declarative representation for problem solving. We end Chapter 10 presenting an extensive list of references on the Prolog language.