Part III: Programming in Lisp

"The name of the song is called 'Haddocks' Eyes.' "

"Oh, that's the name of the song, is it?" Alice said, trying to feel interested.

"No, you don't understand," the Knight said, looking a little vexed. "That's what the name is called. The name really is 'The Aged Aged Man.' "

"Then I ought to have said 'That's what the song is called'?" Alice corrected herself.

"No, you oughtn't: that's quite another thing! The song is called 'Ways and Means': but that's only what it's called you know!"

"Well, what is the song, then?" said Alice, who was by this time completely bewildered.

"I was coming to that," the Knight said.

—Lewis Carroll, Through the Looking Glass

For the almost fifty years of its existence, Lisp has been an important language for artificial intelligence programming. Originally designed for symbolic computing, Lisp has been extended and refined over its lifetime in direct response to the needs of AI applications. Lisp is an *imperative* language: Lisp programs describe *how* to perform an algorithm. This contrasts with *declarative* languages such as Prolog, whose programs are assertions that define relationships and constraints in a problem domain. However, unlike traditional imperative languages, such as FORTRAN, C++ or Java, Lisp is *functional*: its syntax and semantics are derived from the mathematical theory of recursive functions.

The power of functional programming, combined with a rich set of high-level tools for building symbolic data structures such as predicates, frames, networks, rules, and objects, is responsible for Lisp's popularity in the AI community. Lisp is widely used as a language for implementing AI tools and models, particularly in the research community, where its high-level functionality and rich development environment make it an ideal language for building and testing prototype systems.

In Part III, we introduce the syntax and semantics of Common Lisp, with particular emphasis on the features of the language that make it useful for AI programming: the use of lists to create symbolic data structures, and the implementation of interpreters and search algorithms to manipulate these structures. Examples of Lisp programs that we develop in Part III include search engines, pattern matchers, theorem provers, rule-based expert system shells, semantic networks, algorithms for learning, and object-oriented simulations. It is not our goal to provide a complete introduction to Lisp; a number of excellent texts (see the epilogue Chapter 20) do this in

far greater detail than our space allows. Instead, we focus on using Lisp to implement the representation languages and algorithms of artificial intelligence programming.

In Chapter 11 we introduce *symbol expressions*, usually termed *s-expressions*, the syntactic basis for the Lisp language. In Chapter 12, we present lists, and demonstrate recursion as a natural tool for exploring list structures. Chapter 13 presents variables in Lisp and discusses bindings, and scope using Lisp forms including `set` and `let`. We then present abstract data types in Lisp and end the chapter with a production system implementing depth-first search.

Chapter 14 presents functions for building meta-interpreters, including the `map`, `filter`, and `lambda` forms. These functions are then used for building search algorithms in Lisp. As in Prolog, *open* and *closed* lists are used to design depth-first, breadth-first, and best-first search algorithms. These search algorithms are designed around the production system pattern and are in many ways similar to the Prolog search algorithms of Chapter 4.

Chapter 15 creates a *unification* algorithm in Lisp in preparation for, in Chapter 16, logic programming in Lisp. This unification, or general pattern matching algorithm, supports the design of a `read-eval-print` loop that implements embedded interpreters. In Chapter 16 we present a full interpreter for expressions in a restricted form of the predicate calculus. This, in turn, sets up the full expert system shell of Chapter 17.

Chapter 17 first presents streams and delayed evaluation as a lead in to presenting `lisp-shell`, a general-purpose expert system shell in Lisp for problems represented in the predicate calculus. `lisp-shell` requires that the facts and rules of the problem domain to be translated into a pseudo Horn clause form.

In Chapter 18 we present object-oriented structures built in Lisp. We see the language as implementing the three components of object-oriented design: inheritance, encapsulation, and polymorphism. We see this implemented first in semantic networks and then in the full object system using the CLOS (Common Lisp Object System) library. We use CLOS to build a simulation of a heating system for a building.

In Chapter 19 we explore machine learning in Lisp building the full ID3 algorithm and testing it with a "consumer credit" example. Chapter 20 concludes Part III with a discussion of functional programming and a reference list.