PART IV: AI Programming in Java

for now we see as through a glass darkly.

—Paul to the Corinthians

The map is not the territory; the name is not the thing named.

—Alfred Korzybski

What have I learned but the proper use of several tools?.

—Gary Snyder *"What Have I Learned"*

Java is the third language this book examines, and it plays a different role in the development of Artificial Intelligence programming than do Lisp and Prolog. These earlier languages were tied intimately to the intellectual development of the field and, to a large extent, they both reflect and helped to shape its core ideas. We can think of Lisp as a test bed for the basic ideas of functional programming and the specification for the Physical Symbol System Hypothesis (Newell and Simon 1976): that dynamic operations on symbol structures are a necessary and sufficient specification for intelligent activity.

Similarly, Prolog was a direct exploration of declarative specifications of knowledge and intelligence. Java's place in the history of AI is different. Rather than shaping the development of artificial intelligence itself, Java, especially its object-oriented features and inheritance, are a product of earlier AI language research, especially SmallTalk and Flavors. Our inclusion of Java in this book is also evidence that Artificial Intelligence has grown up, in that most of its tools and techniques are now considered to be language independent. In fact, In Part V, Chapters 26, 29 and 31 present Java-based Artificial Intelligence software available on the Internet.

Developed in the early 1990s by the Sun Microsystems Corporation, Java was an effort to realize the full power of object-oriented programming, OOP, in a language expressly designed for large-scale software engineering. In spite of their power and flexibility, early object-oriented languages including SmallTalk, Flavors, and the Common Lisp Object System (CLOS), languages developed by the AI community itself, did not receive the attention they deserved from the software engineering community. Java, implementing many of the features of these early tools, has been more widely accepted.

Although the reasons given for the rejection of these earlier, elegant languages may appear to contemporary programmers to surpass rational understanding, there were some common themes we can mention. Many programmers, particularly those trained on languages like Basic, C, or Fortran, found their Lisp-like syntax odd. Other programmers expressed concern at potential inefficiencies introduced by the interpreted nature of these languages, their implementation linked to dynamic binding, and the limitations of their approach to automatic garbage collection.

It is also possible that these early object-oriented languages were simply lost in the PC revolution. The limited processor speed and memory of these early desktop machines made them poor platforms for memory intensive, dynamic languages like SmallTalk and Lisp. There was at least a perception that the most suitable languages for PC programming were C for systems programming and serious applications development, and Basic for the kind of rapid development at which OOP languages excel. For whatever reasons, in spite of the clear power of object-oriented programming the most widely used OOP language at the time of Java's development was C++, a language deeply flawed by a poor implementation of OOP's dynamic features, a failure to handle types properly, and lack of automatic garbage collection.

Java offers a credible alternative to this. Although it presented the programmer with the familiar C-style syntax, Java implemented such fundamental object-oriented features as dynamic method binding, interface definitions, garbage collection, and object-oriented encapsulation. It addressed the efficiency problems of interpreted languages by pre-compiling the source code into a machine independent intermediate form. This not only made Java faster than interpreted languages while preserving their dynamic capabilities, but also made it possible to compile Java programs that would run on any platform that implemented an appropriate Java Virtual Machine.

Java also offers a number of useful features for practical software engineering, such as the ability to define packages that collected class definitions under a bounded name space. What is perhaps most important, Java was developed expressly as a programming language for interactions with the World-Wide-Web. Based on Java's platform independence, such constructs as Applets, which allowed embedding a Java program in a web page, and later developments like Servlets, WebStart, Java classes for handling XML, and other features have made it the web programming language of choice.

These features also support the relationship between Java and Artificial Intelligence programming. As a credible implementation of object-oriented programming, Java offers many of the capabilities that AI programmers desire. These include the ability to easily create dynamic object structures, strong encapsulation, dynamic typing, true inheritance, and automatic garbage collection. At the same time, Java offers these features in a language that supports large-scale software engineering through packages, growing numbers of reusable software libraries, and rich development environments.

As Artificial Intelligence has matured and moved out of the laboratory into practical use, these features of Java make it well suited for AI applications.

Java supports the patterns of AI programming much more easily than C++, and does so in the context of a powerful software engineering language and environment. What is perhaps most important, Java accomplishes this in a way that promises to make it easy to apply the power of AI to the unlimited resources of the World-Wide-Web. Indeed, one of our major goals in writing this book is to examine the integration of Java into AI practice and thus, as noted above, we have several chapters that explicitly link the development of AI software directly to web-available Java tools.

In working through this section on AI Programming in Java, we recommend the reader keep these goals in mind, to think in terms of using Java to embed AI capability in larger, more general programs. Our approach will be to focus most directly on the implementation of the basic AI structures of search and representation in the Java idiom, and leave the more general topics of web programming and large-scale software engineering in Java to the many fine books on these topics. Thus, our intent is to prepare the reader to develop these powerful AI techniques in an object-oriented idiom that simplifies integrating them into practical, large-scale, software applications.

Chapter 21 begins Part IV and describes how AI representations and algorithms can be created within the object-oriented paradigm. Although some representations, including semantic networks and frames, fit the OOP paradigm well, others, including predicate calculus representation, state-space search, and reasoning through expert system rule systems require more thought. Chapter 22 begins this task by presenting an object-oriented structure that supports state-space search that is general enough to support alternative search algorithms including depth- breadth- and best first search.

Chapters 23 - 25 present OOP representations for the predicate calculus, unification, and production system problem solving. These three chapters fit together into a coherent group as we first create a Java formalism for the predicate calculus and then show inference systems based on a unification algorithm written in Java. Finally, in Chapter 25 we present a full goal-driven expert system shell able to answer the traditional `how` and `why` queries. These three chapters are designed to be *general* in that their approach to representational issues is not to simply support building specific tools, such as a rule based expert system, Rather these chapters are focused on the general issues of predicate calculus representation, the construction and use of a unification algorithm with backtracking, and the design of a predicate calculus based architecture for search

Chapter 26 is short, an introduction to web based expert system shells, with a focus on JESS, a Java-based Expert System Shell developed by Sandia National Laboratories.

Chapter 27 is the first of three chapters presenting Java-based machine learning algorithms. We begin, Chapter 27, with a Java version of the ID3 decision tree algorithm. This is a information theoretic unsupervised algorithm that learns patterns in data. Chapter 28 develops genetic operators and shows how genetic algorithms can be used to learn patterns in complex data sets. Finally, Chapter 29 introduces a number of web-based machine learning software tools written in Java.

Chapters 30 and 31 present natural language processing algorithms written in Java. Chapter 30 builds data structures for the Earley algorithm, an algorithm that adopts techniques from dynamic programming for efficient text based sentence parsing. Finally, Chapter 31 describes a number of natural language processing tools available on the internet, including LingPipe from the University of Pennsylvania, software tools available from the Stanford University language processing group, and Sun Microsystems' speech API.