1 Idioms, Patterns, and Programming

Chapter Objectives

This chapter introduces the ideas that we use to organize our thinking about languages and how they shape the design and implementation of programs.

These are the ideas of language, idiom, and design pattern.

Chapter Contents

1.1 Introduction

Idioms and Patterns

As with any craft, programming contains an undeniable element of experience. We achieve mastery through long practice in solving the problems that inevitably arise in trying to apply technology to actual problem situations. In writing a book that examines the implementation of major AI algorithms in a trio of languages, we hope to support the reader's own experience, much as a book of musical etudes helps a young musician with their own exploration and development.

As important as computational theory, tools, and experience are to a programmer's growth, there is another kind of knowledge that they only suggest. This knowledge comes in the form of pattern languages and idioms, and it forms a major focus of this book. The idea of pattern languages originated in architecture (Alexander et al. 1977) as a way of formalizing the knowledge an architect brings to the design of buildings and cities that will both support and enhance the lives of their residents. In recent years, the idea of pattern languages has swept the literature on software design (Gamma, et al. 1995; Coplein & Schmidt 1995; Evans 2003), as a way of capturing a master's knowledge of good, robust program structure.

A design pattern describes a typical design problem, and outlines an approach to its solution. A pattern language consists of a collection of related design patterns. In the book that first proposed the use of pattern languages in architecture, Christopher Alexander et al. (1977, page x) state that a pattern

> *describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.*

Design patterns capture and communicate a form of knowledge that is essential to creating computer programs that users will embrace, and that

programmers will find to be elegant, logical, and maintainable. They address programming and languages, not in terms of Turing completeness, language paradigms, compiler semantics, or any of the other qualities that constitute the core of computer science, but rather as tools for practical problem solving. To a large extent, you can think of this book as presenting a pattern language of the core problems of AI programming, and examples – the patterns – of their solution.

Idioms are a form and structure for knowledge that helps us bridge the differences between patterns as abstract descriptions of a problem and its solutions and an understanding of how best to implement that solution in a given programming language. A language idiom is the expression of a design pattern in a given language. In this sense, *design patterns + idioms = quality programs*.

Sample Design Patterns

Consider, for example, the simple, widely used design pattern that we can call `map` that applies some operator `O` to every element of a list `L`. We can express this pattern in a pseudo code function as follows:

```
map(operator O, list L)
{
    if (L contains no elements) quit;
    h ← the first element of L.
    apply O to h;
    map(O, L minus h);
}
```

This `map` function produces a stream of results: `O` applied to each element of the list `L`. As our definition of pattern specifies, this describes a solution to a recurring problem, and also fosters unlimited variations, depending on the type of the elements that make up the list `L`, and the nature of the operator, `O`.

Now, let us consider a fragment of Lisp code that implements this same `map` pattern, where `f` is the mapped operator (in Lisp a function) and `list` is the list:

```
(defun map (f list)
    (cond ((null list) nil)
          (t (cons (apply f (car list))
                   (map f (cdr list))))))
```

This function `map`, created by using the built-in Lisp `defun` function, not only implements the `map` pattern, but also illustrates elements of the Lisp programming idiom. These include the use of the operators *car* and *cdr* to separate the list into its head and tail, the use of the *cons* operator to place the results into a new list, and also the use of recursion to move down the list. Indeed, this idiom of recursively working through a list is so central to Lisp, that compiler writers are expected to optimize this sort of tail recursive structure into a more efficient iterative implementation.

Let us now compare the Lisp `map` to a Java implementation that demonstrates how idioms vary across languages:

```
public Vector map(Vector l)
{
    Vector result = new Vector();
    Iterator iter = l.iterator();
    while(iter.hasNext())
    {
        result.add(f(iter.next));
    }
    return result;
}
```

The most striking difference between the Java version and the Lisp version is that the Java version is iterative. We could have written our list search in a recursive form (Java supports recursion, and compilers should optimize it where possible), but Java also offers us iterators for moving through lists. Since the authors of Java provide us with list iterators, and we can assume they are implemented efficiently, it makes sense to use them. The Java idiom differs from the Lisp idiom accordingly.

Furthermore, the Java version of `map` creates the new variable, `result`. When the iterator completes its task, `result` will be a `vector` of elements, each the result of applying `f` to each element of the input list (vector). Finally, `result` must be explicitly returned to the external environment. In Lisp, however, the resulting list of mapped elements is the result of invoking the function `map` (because it is returned as a direct result of evaluating the `map` function).

Finally, we present a Prolog version of `map`. Of course in Prolog, `map` will be a represented as a predicate. This predicate has three arguments, the first the function, `f`, which will be applied to every element of the list that is the second argument of the predicate. The third argument of the predicate `map` is the list resulting from applying `f` to each element of the second argument. The pattern `[X|Y]` is the Prolog list representation, where `X` is the head of the list (`car` in Lisp) and `Y` is the list that is the rest of the list (`cdr` in Lisp). The `is` operator binds the result of `f` applied to `H` to the variable `NH`. As with Lisp, the `map` relationship is defined recursively, although no tail recursive optimization is possible in this case. Further clarifications of this Prolog specification are presented in Part II.

```
map(f, [ ], [ ]).
map(f, [H|T], [NH|NT]):-
    NH is f(H), map(f, T, NT).
```

In the three examples above we see a very simple example of a pattern having different idioms in each language, the *eval&assign* pattern. This pattern evaluates some expression and assigns the result to a variable. In Java, as we saw above, `=` simply assigns the evaluated expression on its right-hand-side to the variable on its left. In Lisp this same activity requires the `cons` of an `apply` of `f` to an element of the list. The resulting symbol expression is then simply returned as part of the evaluated function `map`. In Prolog, using the predicate representation, there are similar

differences between assignment (based on unification with patterns such as `[H|T]` and `=`) and evaluation (using `is` or making `f` be a goal).

Understanding and utilizing these idioms is an essential aspect of mastering a programming language, in that they represent expected ways the language will be used. This not only allows programmers more easily to understand, maintain, and extend each other's code, but also allows us to remain consistent with the language designer's assumptions and implementation choices.

1.2 Selected Examples of AI Language Idioms

We can think of this book, then, as presenting some of the most important patterns supporting Artificial Intelligence programming, and demonstrating their implementation in the appropriate idioms of three major languages. Although most of these patterns were introduced in this book's companion volume, *Artificial Intelligence: Structures and Strategies for Complex Problem Solving* (Luger 2009), it is worthwhile to summarize a subset of them briefly in this introduction.

Symbolic Computing: The Issue of Representation

Artificial Intelligence rests on two basic ideas: first, *representation* or the use of symbol structures to represent problem solving knowledge (state), and second, *search*, the systematic consideration of sequences of operations on these knowledge structures to solve complex problems. Symbolic computing embraces a family of patterns for representing state and then manipulating these (symbol) structures, as opposed to only performing arithmetic calculations on states. Symbolic computing methods are the foundation of artificial intelligence: in a sense, everything in this book rests upon them. The recursive list-handling algorithm described above is a fundamental symbolic computing pattern, as are the basic patterns for tree and graph manipulation. Lisp was developed expressly as a language for symbolic computing, and its s-expression representation (see Chapter 11) has proved general, powerful and long-lived.

As we develop the examples of this book, pay close attention to how these simple patterns of list, tree, and graph manipulation combine to form the more complex, problem specific patterns described below.

Search

Search in AI is also fundamental and complementary to representation (as is emphasized throughout our book. Prolog, in fact, incorporates a form of search directly into its language semantics. In addition to forming a foundation of AI, search introduces many of its thorniest problems. In most interesting problems, search spaces tend to be intractable, and much of AI theory examines the use of heuristics to control this complexity. As has been pointed out from the very beginnings of AI (Feigenbaum and Feldman 1963, Newell and Simon 1976) support of intelligent search places the greatest demands on AI programming.

Search related design patterns and problems we will examine in this book include implementations of the basic search algorithms (breadth-first, depth-first, and best-first), management of search history, and the recovery of solution paths with the use of those histories.

A particularly interesting search related problem is in the representation

and generation of problem states. Conceptually, AI search algorithms are general: they can apply to any search space. Consequently, we will define general, reusable search "frameworks" that can be applied to a range of problem representations and operations for generating new states. How the different programming paradigms address this issue is illuminating in terms of their language-based idioms.

Lisp makes no syntactic distinction between functions and data structures: both can be represented as symbol expressions (see s-expression, Chapter 11), and both can be handled identically as Lisp objects. In addition, Lisp does not enforce strong typing on s-expressions. These two properties of the language allow us to define a general search algorithm that takes as parameters the starting problem state, and a list of Lisp functions, often using the map design pattern described earlier, for producing child states.

Prolog includes a list representation that is very similar to lists in Lisp, but differs in having built-in search and pattern matching in a language supporting direct representation of predicate calculus rules. Implementing a generalized search framework in Prolog builds on this language's unique idioms. We define the operators for generating states as rules, using pattern matching to determine when these rules apply. Prolog offers explicit meta-level controls that allow us to direct the pattern matching, and control its built-in search.

Java presents its own unique idioms for generalizing search. Although Java provides a "reflection" package that allows us to manipulate its objects, methods, and their parameters directly, this is not as simple to do as in Lisp or Prolog. Instead, we will use Java interface definitions to specify the methods a state object must have at a general level, and define search algorithms that take as states instances of any class that instantiates the appropriate interface (see Chapters 22-24).

These three approaches to implementing search are powerful lessons in the differences in language idioms, and the way they relate to a common set of design patterns. Although each language implements search in a unique manner, the basic search algorithms (breadth-, depth-, or best-first) behave identically in each. Similarly, each search algorithm involves a number of design patterns, including the management of problem states on a list, the ordering of the state list to control search, and the application of state-transition operators to a state to produce its descendants. These design patterns are clearly present in all algorithms; it is only at the level of language syntax, semantics, and idioms that these implementations differ.

Pattern Matching

Pattern matching is another support technology for AI programming that spawns a number of useful design patterns. Approaches to pattern matching can vary from checking for identical memory locations, to comparing simple regular-expressions, to full pattern-based unification across predicate calculus expressions, see Luger (2009, Section 2.3). Once again, the differences in the way each language implements pattern matching illustrate critical differences in their semantic structure and associated idioms.

Prolog provides unification pattern matching directly in its interpreter: unification and search on Predicate Calculus based data structures are the

basis of Prolog semantics. Here, the question is not how to implement pattern matching, but how to use it to control search, the flow of program execution, and the use of variable bindings to construct problem solutions as search progresses. In this sense, Prolog gives rise to its own very unique language idioms.

Lisp, in contrast, requires that we implement unification pattern matching ourselves. Using its basic symbolic computing capabilities, Lisp makes it straightforward to match recursively the tree structures that implicitly define predicate calculus expressions. Here, the main design problem facing us is the management of variable bindings across the unification algorithm. Because Lisp is so well suited to this type of implementation, we can take its implementation of unification as a "reference implementation" for understanding both Prolog semantics, and the Java implementation of the same algorithm.

Unlike Lisp, which allows us to use nested s-expressions to define tree structures, Java is a strongly typed language. Consequently, our Java implementation will depend upon a number of user-created classes to define expressions, constants, variables, and variable bindings. As with our implementation of search, the differences between the Java and Lisp implementations of pattern matching are interesting examples of the differences between the two languages, their distinct idioms, and their differing roles in AI programming.

Structured Types and Inheritance (Frames)

Although the basic symbolic structures (lists, trees, etc.) supported by all these languages are at the foundation of AI programming, a major focus of AI work is on producing representations that reflect the way people think about problems. This leads to more complex structures that reflect the organization of taxonomies, similarity relationships, ontologies, and other cognitive structures. One of the most important of these comes from frame theory (Minsky 1975; Luger 2009, Section 7.1), and is based on structured data types (collections of individual attributes combined in a single object or *frame*), explicit relationships between objects, and the use of class inheritance to capture hierarchical organizations of classes and their attributes.

These representational principles have proved so effective for practical knowledge representation that they formed the basis of object-oriented programming: Smalltalk, the CommonLisp Object System libraries (CLOS), C++, and Java. Just as Prolog bases its organization on predicate calculus and search, and Lisp builds on (functional) operations on symbolic structures, so Java builds directly on these ideas of structured representation and inheritance.

This approach of object-oriented programming underlies a large number of design patterns and their associated idioms (Gamma, et al. 1995; Coplein & Schmidt 1995), as merited by the expressiveness of the approach. In this book, we will often focus on the use of structured representations not simply for design of program code, but also as a tool for knowledge representation.

Meta-Linguistic Abstraction

Meta-linguistic abstraction is one of the most powerful ways of organizing programs to solve complex problems. In spite of its imposing title, the

idea behind meta-linguistic abstraction is straightforward: rather than trying to write a solution to a hard problem in an existing programming language, use that language to create another language that is better suited to solving the problem. We have touched on this idea briefly in this introduction in our mention of general search frameworks, and will develop it throughout the book (e.g., Chapters 5, 15, 26).

One example of meta-linguistic abstraction that is central to AI is the idea of an *inference engine*: a program that takes a declarative representation of domain knowledge in the form of rules, frames or some other representation, and applies that knowledge to problems using general inference algorithms. The commonest example of an inference engine is found in a rule-based expert system shell. We will develop such a shell, EXSHELL in Prolog (Chapter 6), Lisp-shell in Lisp (Chapter 17), and an equivalent system in Java (Chapter 26), providing similar semantics in all three language environments. This will be a central focus of the book, and will provide an in-depth comparison of the programming idioms supported by each of these languages.

Knowledge-Level Design

This discussion of AI design patterns and language idioms has proceeded from simple features, such as basic, list-based symbol processing, to more powerful AI techniques such as frame representations and expert system shells. In doing so, we are adopting an organization parallel to the theoretical discussion in Artificial Intelligence: Strategies and Structures for Complex Problem Solving (Luger 2009). We are building a set of tools for programming at what Allen Newell (1982) has called the *knowledge level*.

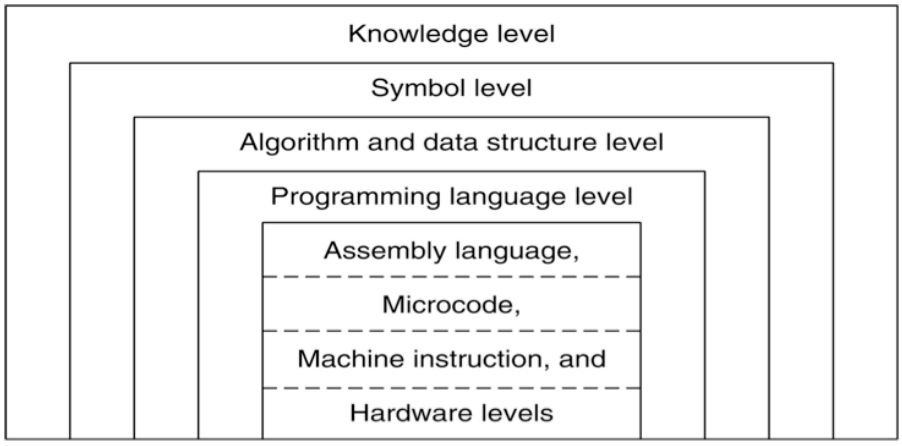

Figure 1.1 Levels of a Knowledge-Based System, adapted from Newell (1982).

Allen Newell (1982) has distinguished between the *knowledge level* and the *symbol level* in describing an intelligent system. As may be seen in Figure 1.1 (adapted from Newell, 1982), the symbol level is concerned with the particular formalisms used to represent problem solving knowledge, for example the predicate calculus. Above this symbol level is the knowledge level concerned with the knowledge content of the program and the way in which that knowledge is used.

The distinction between the symbol and knowledge level is reflected in the

architectures of expert systems and other knowledge-based programs (see Chapters 6, 15, and 25). Since the user will understand these programs in terms of their knowledge content, these programs must preserve two invariants: first, as just noted, there must be a knowledge-level characterization, and second, there must be a clear distinction between this knowledge and its control. We see this second invariant when we utilize the *production system* design pattern in Chapters 6, 15, and 25. Knowledge level concerns include questions such as: What queries will be made of the system? What objects and/or relations are important in the domain? How is new knowledge added to the system? Will information change over time? How will the system need to reason about its knowledge? Does the problem domain include missing or uncertain information?

The symbol level, just below the knowledge level, defines the knowledge representation language, whether it be direct use of the predicate calculus or production rules. At this level decisions are made about the structures required to represent and organize knowledge. This separation from the knowledge level allows the programmer to address such issues as expressiveness, efficiency, and ease of programming, that are not relevant to the programs higher level intent and behavior.

The implementation of the *algorithm and data structure* level constitutes a still lower level of program organization, and defines an additional set of design considerations. For instance, the behavior of a logic-based or function-based program should be unaffected by the use of a hash table, heap, or binary tree for implementing its symbol tables. These are implementation decisions and invisible at higher levels. In fact, most of the techniques used to implement representation languages for AI are common computer science techniques, including binary trees and tables and an important component of the knowledge-level design hypothesis is that they be hidden from the programmer.

In thinking of knowledge level programming, we are defining a hierarchy that uses basic programming language constructs to create more sophisticated symbol processing languages, and uses these symbolic languages to capture knowledge of complex problem domains. This is a natural hierarchy that moves from machine models that reflect an underlying computer architecture of variables, assignments and processes, to a symbolic layer that works with more abstract ideas of symbolic representation and inference. The knowledge level looks beyond symbolic form to the semantics of problem solving domains and their associated knowledge relationships.

The importance of this multi-level approach to system design cannot be overemphasized: it allows a programmer to ignore the complexity hidden at lower levels and focus on issues appropriate to the current level of abstraction. It allows the theoretical foundations of artificial intelligence to be kept free of the nuances of a particular implementation or programming language. It allows us to modify an implementation, improving its efficiency or porting it to another machine, without affecting its specification and behavior at higher levels. But the AI programmer begins addressing the problem-solving task from the programming language level.

In fact, we may characterize the programmer's ability to use design patterns and their associated idioms as her ability to bridge and link the algorithms and data structures afforded by different language paradigms with the symbol level in the process of building expressive knowledge-intensive programs.

To a large extent, then, our goal in writing this book is to give the reader the intellectual tools for programming at the knowledge level. Just as an experienced musician thinks past the problems of articulating individual notes and chords on their instrument to the challenges of harmonic and rhythmic structure in a composition, or an architect looks beyond the layout of floor plans to ways buildings will interact with their occupants over time, we believe the goal of a programmer's development is to think of computer programs in terms of the knowledge they incorporate, and the way they engage human beings in the patterns of their work, communication and relationships. Becoming the "master programmer" we mentioned earlier in this introduction requires the ability to think in terms of the human activities a program will support, and simultaneously to understand the many levels of abstraction, algorithms, and data structures that lie between those activities and the comparatively barren structures of the "raw" programming language

1.3 A Brief History of Three Programming Paradigms

We conclude this chapter by giving a brief description of the origins of the three programming languages we present. We also give a cursory description of the three paradigms these languages represent. These details are precursors of and an introduction to the material presented in the next three parts of this book.

Logic Programming in Prolog

Like Lisp, Prolog gains much of its power and elegance from its foundations in mathematics. In the case of Prolog, those foundations are predicate logic and resolution theorem proving. Of the three languages presented in this book, Prolog may well seem unusual to most programmers in that it is a declarative, rather than procedural, language. A Prolog program is simply a statement, in first-order predicate calculus, of the logical conditions a solution to a problem must satisfy. The declarative semantics do not tell the computer what to do, only the conditions a solution must satisfy. Execution of a Prolog program relies on search to find a set of variable bindings that satisfy the conditions stated in the particular goals required by the program. This declarative semantics makes Prolog extremely powerful for a large class of problems that are of particular interest to AI. These include constraint satisfaction problems, natural language parsing, and many search problems, as will be demonstrated in Part II.

A logic program is a set of specifications in formal logic; Prolog uses the first-order predicate calculus. Indeed, the name itself comes from **pro**gramming in **log**ic. An interpreter executes the program by systematically making inferences from logic specifications. The idea of using the representational power of the first-order predicate calculus to express specifications for problem solving is one of the central

contributions Prolog has made to computer science in general and to artificial intelligence in particular. The benefits of using first-order predicate calculus for a programming language include a clean and elegant syntax and a well-defined semantics.

The implementation of Prolog has its roots in research on theorem proving by J.A. Robinson (Robinson 1965), especially the creation of algorithms for resolution refutation systems. Robinson designed a proof procedure called resolution, which is the primary method for computing with Prolog. For a more complete description of resolution refutation systems and of Prolog as Horn clause refutation, see Luger (2009, Chapter 14).

Because of these features, Prolog has proved to be a useful vehicle for investigating such experimental programming issues as automatic code generation, program verification, and design of high-level specification languages. As noted above, Prolog and other logic-based languages support a declarative programming style—that is, constructing a program in terms of high-level descriptions of a problem's constraints—rather than a procedural programming style—writing programs as a sequence of instructions for performing an algorithm. This mode of programming essentially tells the computer "what is true" and "what needs to be proven (the goals)" rather than "how to do it." This allows programmers to focus on problem solving as creating sets of specifications for a domain rather than the details of writing low-level algorithmic instructions for "what to do next."

The first Prolog program was written in Marseille, France, in the early 1970s as part of a project in natural language understanding (Colmerauer, Kanoui et al. 1973, Roussel 1975, Kowalski 1979). The theoretical background for the language is discussed in the work of Kowalski, Hayes, and others (Hayes 1977, Kowalski 1979, Kowalski 1979, Lloyd 1984). The major development of the Prolog language was carried out from 1975 to 1979 at the Department of Artificial Intelligence of the University of Edinburgh. The people at Edinburgh responsible for the first "road worthy" implementation of Prolog were David H.D. Warren and Fernando Pereira. They produced the first Prolog interpreter robust enough for delivery to the general computing community. This product was built using the "C" language on the DEC-system 10 and could operate in both interpretive and compiled modes (Warren, Pereira, et al. 1979).

Further descriptions of this early code and comparisons of Prolog with Lisp may be found in Warren et al. (Warren, Pereira, et al. 1977). This "Warren and Pereira" Prolog became the early standard. The book *Programming in Prolog* (Clocksin and Mellish 1984, now in its fifth edition) was created by two other researchers at the Department of Artificial Intelligence, Bill Clocksin and Chris Mellish. This book quickly became the chief vehicle for delivering Prolog to the computing community. We use this standard, which has come to be known as Edinburgh Prolog. In fact, all the Prolog code in this book may be run on the public domain interpreter SWI-Prolog (to find, Google on swi-prolog).

Functional Programming in Lisp

Lisp was arguably the first programming language to ground its semantics in mathematical theory: the theory of partial recursive functions (McCarthy

1960, Church 1941). In contrast to most of its contemporaries, which essentially presented the architecture of the underlying computer in a higher-level form, this mathematical grounding has given Lisp unusual power, durability and influence. Ideas such as list-based data structures, functional programming, and dynamic binding, which are now accepted features of mainstream programming languages can trace their origins to earlier work in Lisp. Meta-circular definition, in which compilers and interpreters for a language are written in a core version of the language itself, was the basis of the first, and subsequent Lisp implementations. This approach, still revolutionary after more than fifty years, replaces cumbersome language specifications with an elegant, formal, public, testable meta-language kernel that supports the continued growth and refinement of the language.

Lisp was first proposed by John McCarthy in the late 1950s. The language was originally intended as an alternative model of computation based on the theory of recursive functions. In an early paper, McCarthy (McCarthy 1960) outlined his goals: to create a language for symbolic rather than numeric computation, to implement a model of computation based on the theory of recursive functions (Church 1941), to provide a clear definition of the language's syntax and semantics, and to demonstrate formally the completeness of this computational model. Although Lisp is one of the oldest computing languages still in active use (along with FORTRAN and COBOL), the careful thought given to its original design and the extensions made to the language through its history have kept it in the vanguard of programming languages. In fact, this programming model has proved so effective that a number of other languages have been based on functional programming, including SCHEME, SML-NJ, FP, and OCAML. In fact, several of these newer languages, e.g., SCHEME and SML-NJ, have been designed specifically to reclaim the semantic clarity of the earlier versions of Lisp.

The list is the basis of both programs and data structures in Lisp: Lisp is an acronym for **lis**t **p**rocessing. Lisp provides a powerful set of list-handling functions implemented internally as linked pointer structures. Lisp gives programmers the full power and generality of linked data structures while freeing them, with real-time garbage collection, from the responsibility for explicitly managing pointers and pointer operations.

Originally, Lisp was a compact language, consisting of functions for constructing and accessing lists (`car`, `cdr`, `cons`), defining new functions (`defun`), detecting equality (`eq`), and evaluating expressions (`quote`, `eval`). The only means for building program control were recursion and a single conditional. More complicated functions, when needed, were defined in terms of these primitives. Through time, the best of these new functions became part of the language itself. This process of extending the language by adding new functions led to the development of numerous dialects of Lisp, often including hundreds of specialized functions for data structuring, program control, real and integer arithmetic, input/output (I/O), editing Lisp functions, and tracing program execution. These dialects are the vehicle by which Lisp has evolved from a simple and elegant theoretical model of computing into a rich, powerful, and practical

environment for building large software systems. Because of the proliferation of early Lisp dialects, the Defense Advanced Research Projects Agency in 1983 proposed a standard dialect for the language, known as Common Lisp.

Although Common Lisp has emerged as the lingua franca of Lisp dialects, a number of simpler dialects continue to be widely used. One of the most important of these is SCHEME, an elegant rethinking of the language that has been used both for AI development and for teaching the fundamental concepts of computer science. The dialect we use throughout the remainder of our book is Common Lisp. All our code may be run on a current public domain interpreter built by Carnegie Mellon University, called CMUCL (Google CMUCL).

Object-Oriented Programming in Java

Java is the third language considered in this book. Although it does not have Lisp or Prolog's long historical association with Artificial Intelligence, it has become extremely important as a tool for delivering practical AI applications. There are two primary reasons for this. The first is Java's elegant, dynamic implementation of object-oriented programming, a programming paradigm with its roots in AI, that has proven its power for use building AI programs through Smalltalk, Flavors, the Common Lisp Object System (CLOS), and other object-oriented systems. The second reason for Java's importance to AI is that it has emerged as a primary language for delivering tools and content over the world-wide-web. Java's ease of programming and the large amounts of reusable code available to programmers greatly simplify the coding of complex programs involving AI techniques. We demonstrate this in the final chapters of Part IV.

Object-oriented programming is based on the idea that programs can be best modularized in terms of objects: encapsulated structures of data and functionality that can be referenced and manipulated as a unit. The power of this programming model is enhanced by inheritance, or the ability to define sub-classes of more general objects that inherit and modify their functionality, and the subtle control object-oriented languages provide over the scoping of variables and functions alike.

The first language to build object-oriented representations was created in Norway in the 1960s. Simula-67 was, appropriately, a simulation language. Simulation is a natural application of object-oriented programming that language objects are used to represent objects in the domain being simulated. Indeed, this ability to easily define isomorphisms between the representations in an object-oriented program and a simulation domain has carried over into modern object-oriented programming style, where programmers are encouraged to model domain objects and their interactions directly in their code.

Perhaps the most elegant formulation of the object-oriented model is in the Smalltalk programming language, built at Xerox PARC in the early 1970s. Smalltalk not only presented a very pure form of object-oriented programming, but also used it as a tool for graphics programming. Many of the ideas now central to graphics interfaces, such as manipulable screen objects, event driven interaction, and so on, found their early implementation in the Smalltalk language. Other, later implementations of

object-programming include C++, Objective C, C#, and the Common Lisp Object System. The success of the model has made it rare to find a programming language that does not incorporate at least some object-oriented ideas.

Our first introduction of object-oriented languages is with the Common Lisp Object System in Chapter 18 of Part III. However, in Part IV, we have chosen Java to present the use of object-oriented tools for AI programming. Java offers an elegant implementation of object-orientation that implements single inheritance, dynamic binding, interface definitions, packages, and other object concepts in a language syntax that most programmers will find natural. Java is also widely supported and documented.

The primary reason, however, for including Java in this book is its great success as a practical programming language for a large number and variety of applications, most notably those on the world-wide-web. One of the great benefits of object-oriented programming languages is that the ability to define objects combining data and related methods in a single structure encourages the development of reusable software objects.

Although Java is, at its core, a relatively simple language, the efforts of thousands of programmers have led to large amounts of high-quality, often open source, Java code. This includes code for networking, graphics, processing html and XML, security, and other techniques for programming on the world-wide-web. We will examine a number of public domain Java tools for AI, such as expert system rule engines, machine learning algorithms, and natural language parsers. In addition, the modularity and control of the object-oriented model supports the development of large programs. This has led to the embedding of AI techniques in larger and indeed more ordinary programs. We see Java as an essential language for delivering AI in practical contexts, and will discuss the Java language in this context. In this book we refer primarily to public domain interpreters most of which are easily web accessible.

1.4 A Summary of Our Task

We hope that in reading this introductory chapter, you have come to see that our goal in writing this book is not simply to present basic implementation strategies for major Artificial Intelligence algorithms. Rather, our goal is to look at programming languages as tools for the intellectual activities of design, knowledge modeling, and system development.

Computer programming has long been the focus both for scientific theory and engineering practice. These disciplines have given us powerful tools for the definition and analysis of algorithms and for the practical management of large and small programming projects. In writing this book, it has been our overarching goal to provide a third perspective on programming languages: as tools for the art of designing systems to support people in their thinking, communication, and work.

It is in this third perspective that the ideas of idioms and patterns become

important. It is not our goal simply to present examples of artificial intelligence algorithms that can be reused in a narrow range of situations. Our goal is to use these algorithms – with all their complexity and challenges – to help programmers build a repertoire of patterns and idioms that can serve well across a wide range of practical problem solving situations. The examples of this book are not ends in themselves; they are only small steps in the maturation of the master programmer. Our goal is to see them as starting points for developing programmers' skills. We hope you will share our enthusiasm for these remarkable artist's tools and the design patterns and idioms they both enable and support.