AI Algorithms, Data Structures, and Idioms in Prolog, Lisp, and Java

AI Algorithms, Data Structures, and Idioms in Prolog, Lisp, and Java

George F. Luger
William A. Stubblefield

Executive Editor	Michael Hirsch
Acquisitions Editor	Matt Goldstein
Editorial Assistant	Sarah Milmore
Managing Editor	Jeff Holcomb
Digital Assets Manager	Marianne Groth
Senior Media Producer	Bethany Tidd
Marketing Manager	Erin Davis
Senior Author Support/ Technology Specialist	Joe Vetere
Senior Manufacturing Buyer	Carol Melville
Text Design, Composition, and Illustrations	George F Luger
Cover Design	Barbara Atkinson
Cover Image	© Tom Barrow

ISBN-13: 978-0-13-607047-4
ISBN-10: 0-13-607047-7

1 2 3 4 5 6 7 8 9 10—OPM—12 11 10 09 08

Contents

Preface

What we have to learn to do
We learn by doing…

- Aristotle, *Ethics*

Why Another Programming Language Book?

Writing a book about designing and implementing representations and search algorithms in Prolog, Lisp, and Java presents the authors with a number of exciting opportunities.

The first opportunity is the chance to compare three languages that give very different expression to the many ideas that have shaped the evolution of programming languages as a whole. These core ideas, which also support modern AI technology, include functional programming, list processing, predicate logic, declarative representation, dynamic binding, meta-linguistic abstraction, strong-typing, meta-circular definition, and object-oriented design and programming. Lisp and Prolog are, of course, widely recognized for their contributions to the evolution, theory, and practice of programming language design. Java, the youngest of this trio, is both an example of how the ideas pioneered in these earlier languages have shaped modern applicative programming, as well as a powerful tool for delivering AI applications on personal computers, local networks, and the world wide web.

The second opportunity this book affords is a chance to look at Artificial Intelligence from the point of view of the craft of programming. Although we sometimes are tempted to think of AI as a theoretical position on the nature of intelligent activity, the complexity of the problems AI addresses has made it a primary driver of progress in programming languages, development environments, and software engineering methods. Both Lisp and Prolog originated expressly as tools to address the demands of symbolic computing. Java draws on object-orientation and other ideas that can trace their roots back to AI programming. What is more important, AI has done much to shape our thinking about program organization, data structures, knowledge representation, and other elements of the software craft. Anyone who understands how to give a simple, elegant formulation to unification-based pattern matching, logical inference, machine learning theories, and the other algorithms discussed in this book has taken a large step toward becoming a master programmer.

The book's third, and in a sense, unifying focus lies at the intersection of these points of view: how does a programming language's formal structure interact with the demands of the art and practice of programming to

create the idioms that define its accepted use. By idiom, we mean a set of conventionally accepted patterns for using the language in practice. Although not the only way of using a language, an idiom defines patterns of use that have proven effective, and constitute a common understanding among programmers of how to use the language. Programming language idioms do much to both enable, as well as support, ongoing communication and collaboration between programmers.

These, then, are the three points of view that shape our discussion of AI programming. It is our hope that they will help to make this book more than a practical guide to advanced programming techniques (although it is certainly that). We hope that they will communicate the intellectual depth and pleasure that we have found in mastering a programming language and using it to create elegant and powerful computer programs.

The Design of this Book

There are five sections of this book. The first, made up of a single chapter, lays the conceptual groundwork for the sections that follow. This first chapter provides a general introduction to programming languages and style, and asks questions such as "What is a *master programmer*?" What is a programming language *idiom*?," and "How are identical *design patterns* implemented in different languages?" Next, we introduce a number of design patterns specific to supporting data structures and search strategies for complex problem solving. These patterns are discussed in a "language neutral" context, with pointers to the specifics of the individual programming paradigms presented in the subsequent sections of our book. The first chapter ends with a short historical overview of the evolution of the logic-based, functional, and object-oriented approaches to computer programming languages.

Part II of this book presents Prolog. For readers that know the rudiments of first-order predicate logic, the chapters of Part II can be seen as a tutorial introduction to Prolog, the language for **pro**gramming in **log**ic. For readers lacking any knowledge of the propositional and predicate calculi we recommend reviewing an introductory textbook on logic. Alternatively, Luger (2005, Chapter 2) presents a full introduction to both the propositional and predicate logics. The Luger introduction includes a discussion, as well as a pseudo code implementation, of unification, the pattern-matching algorithm at the heart of the Prolog engine.

The design patterns that make up Part II begin with the "flat" logic-based representation for facts, rules, and goals that one might expect in any relational data base formalism. We next show how recursion, supported by unification-based pattern matching, provides a natural design pattern for tree and graph search algorithms. We then build a series of abstract data types, including sets, stacks, queues, and priority queues that support patterns for search. These are, of course, abstract structures, crafted for the specifics of the logic-programming environment that can search across state spaces of arbitrary content and complexity. We then build and demonstrate the "production system" design pattern that supports rule based programming, planning, and a large number of other AI technologies. Next, we present structured representations, including

semantic networks and frame systems in Prolog and demonstrate techniques for implementing single and multiple inheritance representation and search. Finally, we show how the Prolog design patterns presented in Part II can support the tasks of machine learning and natural language understanding.

Lisp and functional programming make up Part III. Again, we present the material on Lisp in the form of a tutorial introduction. Thus, a programmer with little or no experience in Lisp is gradually introduced to the critical data structures and search algorithms of Lisp that support symbolic computing. We begin with the (recursive) definition of symbol-expressions, the basic components of the Lisp language. Next we present the "assembly instructions" for symbol expressions, including car, cdr, and cons. We then assemble new patterns for Lisp with cond and defun. Finally, we demonstrate the creation and/or evaluation of symbol expressions with quote and eval. Of course, the ongoing discussion of variables, binding, scope, and closures is critical to building more complex design patterns in Lisp.

Once the preliminary tools and techniques for Lisp are presented, we describe and construct many of the design patterns seen earlier in the Prolog section. These include patterns supporting breadth-first, depth-first, and best-first search as well as meta-interpreters for rule-based systems and planning. We build and demonstrate a recursion-based unification algorithm that supports a logic interpreter in Lisp as well as a stream processor with delayed evaluation for handling potentially infinite structures. We next present data structures for building semantic networks and object systems. We then present the Common Lisp Object system (CLOS) libraries for building object and inheritance based design patterns. We close Part III by building design patterns that support decision-tree based machine learning.

Java and its idioms are presented in Part IV. Because of the complexities of the Java language, Part IV is not presented as a tutorial introduction to the language itself. It is expected that the reader has completed at least an introductory course in Java programming, or at the very least, has seen object-oriented programming in another applicative language such as C++, C#, or Objective C. But once we can assume a basic understanding of Java tools, we do provide a tutorial introduction to many of the design patterns of the language.

The first chapter of Part IV, after a brief overview of the origins of Java, goes through many of the features of an object-oriented language that will support the creation of design patterns in that environment. These features include the fundamental data structuring philosophy of encapsulation, polymorphism, and inheritance. Based on these concepts we briefly address the analysis, iterative design, programming and test phases for engineering programs. After the introductory chapter we begin pattern building in Java, first considering the representation issue and how to represent predicate calculus structures in Java. This leads to building

patterns that support breadth-first, depth-first, and best-first search. Based on patterns for search, we build a production system, a pattern that supports the rule-based expert system. Our further design patterns support the application areas of natural language processing and machine learning. An important strength that Java offers, again because of its object-orientation and modularity is the use of public domain (and other) libraries available on the web. We include in the Java section a number of web-supported AI algorithms, including tools supporting work in natural language, genetic and evolutionary programming (a-life), natural language understanding, and machine learning (WEKA).

The final component of the book, Part V, brings together many of the design patterns introduced in the earlier sections. It also allows the authors to reinforce many of the common themes that are, of necessity, distributed across the various components of the presentation, We conclude with general comments supporting the craft of programming.

Using this Book

This book is designed for three primary purposes. The first is as a programming language component of a general class in Artificial Intelligence. From this viewpoint, the authors see as essential that the AI student build the significant algorithms that support the practice of AI. This book is designed to present exactly these algorithms. However, in the normal lecture/lab approach taken to teaching Artificial Intelligence at the University level, we have often found that it is difficult to cover more than one language per quarter or semester course. Therefore we expect that the various parts of this material, those dedicated to either Lisp, Prolog, or Java, would be used individually to support programming the data structures and algorithms presented in the AI course itself. In a more advanced course in AI it would be expected that the class cover more than one of these programming paradigms.

The second use of this book is for university classes exploring programming paradigms themselves. Many modern computer science departments offer a final year course in comparative programming environments. The three languages covered in our book offer excellent examples on these paradigms. We also feel that a paradigms course should not be based on a rapid survey of a large number of languages while doing a few "finger exercises" in each. Our philosophy for a paradigms course is to get the student more deeply involved in fewer languages, and these typically representing the declarative, functional, and object-oriented approaches to programming. We also feel that the study of idiom and design patterns in different environments can greatly expand the skill set of the graduating student. Thus, our philosophy of programming is built around the language idioms and design patterns presented in Part I and summarized in Part V. We see these as an exciting opportunity for students to appreciate the wealth and diversity of modern computing environments. We feel this book offers exactly this opportunity.

The third intent of this book is to offer the professional programmer the chance to continue their education through the exploration of multiple

programming idioms, patterns, and paradigms. For these readers we also feel the discussion of programming idioms and design patterns presented throughout our book is important. We are all struggling to achieve the status of the *master programmer*.

We have built each chapter in this book to reflect the materials that would be covered in either one or two classroom lectures, or about an hour's effort, if the reader is going through this material by herself. There are a small number of exercises at the end of most chapters that may be used to reinforce the main concepts of that chapter. There is also, near the end of each chapter, a summary statement of the core concepts covered.

Acknowledgments

First, we must thank several decades of students and colleagues at the University of New Mexico. These friends not only suggested, helped design, and tested our algorithms but have also challenged us to make them better.

Second, we owe particular thanks to colleagues who wrote algorithms and early drafts of chapters. These include Stan Lee, (PhD student at UNM) for the Prolog chapter on Earley parsing, Breanna Ammons (MS in CS at UNM) for the Java version of the Earley parser and along with Robert Spurlock (CS undergraduate at UNM) the web-based NLP chapter, Paul DePalma (Professor of CS at Gonzaga University) for the Java Genetic Algorithms chapter, and Chayan Chakrabarti (MS in CS at UNM) for the web-based machine learning chapter in Java

Third, there are several professional colleagues that we owe particular debts. These include David MacQueen, University of Chicago, one of the creators of SML, Manuel Hermenegildo, The Prince of Asturias Endowed Chair of Computer Science at UNM and a designer of Ciao Prolog, Paul De Palma, Professor of Computer Science at Gonzaga University, and Alejandro Cdebaca, our friend and former student, who helped design many of the algorithms of the Java chapters.

Fourth, we thank our friends at Pearson Education who have supported our various creative writing activities over the past two decades. We especially acknowledge our editors Alan Apt, Karen Mossman, Keith Mansfield, Owen Knight, Simon Plumtree, and Matt Goldstein, along with their various associate editors, proof readers, and web support personnel.

We also thank our wives, children, family, and friends; all those that have made our lives not just survivable, but intellectually stimulating and enjoyable.

Finally, to our readers; we salute you: the art, science, and practice of programming is great fun, enjoy it!

GL

BS

July 2008

Albuquerque

PART I: Language Idioms and the Master Programmer

all good things - trout as well as eternal salvation - come by grace and grace comes by art and art does not come easy…

- *Norman Maclean*, (1989) A River Runs Through It

Language and Idioms

In defining a programming language idiom, an analogy with natural language use might help. If I ask a friend, "Do you know what time it is?" or equivalently "Do you have a watch?", I would be surprised if she simply said "yes" and turned away. These particular forms for asking someone for the time of day are idiomatic in that they carry a meaning beyond their literal interpretation. Similarly, a programming language idiom consists of those patterns of use that good programmers accept as elegant, expressive of their design intent, and that best harness the language's power. Good idiomatic style tends to be specific to a given language or language paradigm: the way experienced programmers organize a Prolog program would not constitute accepted Java style.

Language idioms serve two roles. The first is to enhance communication between programmers. As experienced programmers know, we do not simply write code for a compiler; we also write it for each other. Writing in a standard idiom makes it easier for other people to understand our intent, and to maintain and/or extend our code. Second, a language's idiom helps us to make sure we fully use the power the language designers have afforded us. People design a language with certain programming styles in mind. In the case of Java, that style was object-oriented programming, and getting full benefit of such Java features as inheritance, scoping, automatic garbage collection, exception handling, type checking, packages, interfaces, and so forth requires writing in an object-oriented idiom. A primary goal of this book is to explore and give examples of good idioms in three diverse language paradigms: the declarative (logic-based), functional, and object-oriented.

The Master Programmer

The goal of this book is to develop the idea and describe the practice of the *master programmer*. This phrase carries a decidedly working class connotation, suggesting the kind of knowledge and effort that comes through long practice and the transmission of tools and skills from master to student through the musty rituals of apprenticeship. It certainly suggests something beyond the immaculate formalization that we generally associate with scientific disciplines. Indeed, most computer science curricula

downplay this craft of programming, favoring discussions of computability and complexity, algorithms, data structures, and the software engineer's formalist longings. In reality, the idea of programming as a craft that demands skill and dedication is widely accepted in practical circles. Few major successful programming projects have existed that did not owe significant components of that success to the craftsmanship of such individuals.

But, what then, do master programmers know?

The foundation of a master programmer's knowledge is a strong understanding of the core domains of computer science. Although working programmers may not spend much (or any) time developing and publishing theorems, they almost always have a deep, intuitive grasp of algorithms, data structures, logic, complexity, and other aspects of the theory of formal systems. We could compare this to a master welder's understanding of metallurgy: she may not have a theoretician's grasp of metallic crystalline structure, but her welds do not crack. This book presumes a strong grounding in these computer science disciplines.

Master programmers also tend to be language fanatics, exhibiting a fluency in several programming languages, and an active interest in anything new and unusual. We hope that our discussion of three major languages will appeal to the craftsman's fascination with their various tools and techniques. We also hope that, by contrasting these three major languages in a sort of "comparative language" discussion, we will help programmers refine their understanding of what a language can provide, and the needs that continue to drive the evolution of programming languages.

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*.

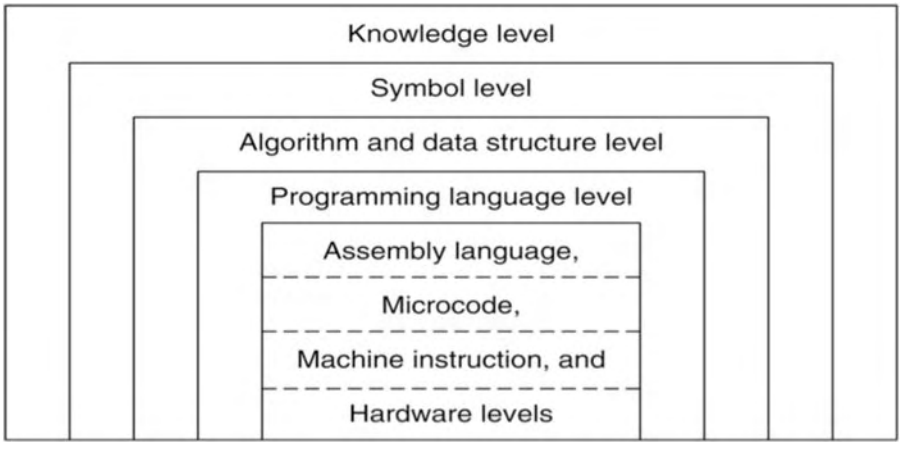

Figure 1.1 Levels of a Knowledge-Based System, adapted from Newell (1982).

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.

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.

2 Prolog: Representation

Chapter Objectives

Prolog's fundamental representations are described and built:
- Facts
- Rules
- The and, or, not, and imply connectives

The environment for Prolog is presented:
- The program as a data base of facts and relations between facts
- Predicates are for creating and modifying this data base

Prolog's procedural semantics is described with examples
- Pattern-matching

Left-to-right depth-first search
Backtracking on variable bindings
The built-in predicates for monitoring Prolog's execution are presented
- spy and trace

The list representation and recursive search are introduced
- Examples of member check and writing out lists

Representations for structured hierarchies are created in Prolog
- Semantic net and frame systems
- Inherited properties determined through recursive (tree) search

Chapter Contents

2.1 Introduction: Logic-Based Representation

Prolog and Logic

Prolog is a computer language that uses many of the representational strengths of the First-Order Predicate Calculus (Luger 2009, Chapter 2). Because Prolog has this representational power it can express general relationships between entities. This allows expressions such as "all females are intelligent" rather than the limited representations of the propositional calculus: "Kate is intelligent", "Sarah is intelligent", "Karen is intelligent", and so on for a very long time!

As in the Predicate Calculus, predicates offer the primary (and only) representational structure in Prolog. Predicates can have zero or more arguments, where their *arity* is the number of arguments. Functions may only be represented as the argument of a predicate; they cannot be a program statement in themselves. Prolog predicates have the usual `and`, `or`, `not` and `implies` connectives. The predicate representation along with its connectives is presented in Section 2.2.

Prolog also takes on many of the declarative aspects of the Predicate Calculus in the sense that a program is simply the set of all true predicates that describe a domain. The Prolog interpreter can be seen as a "theorem prover" that takes the user's query and determines whether or not it is true, as well as what variable substitutions might be required to make the query true. If the query is not true in the context of the program's specifications, the interpreter says "no."

2.2 Prolog Syntax

Facts, Rules and Connectives

Although there are numerous dialects of Prolog, the syntax used throughout this text is that of the original Warren and Pereira C-Prolog as described by Clocksin and Mellish (2003). We begin with the set of connectives that can take atomic predicates and join them with other expressions to make more complex relationships. There are, because of the usual keyboard conventions, a number of differences between Prolog and predicate calculus syntax. In C-Prolog, for example, the symbol :- replaces the ← of first-order predicate calculus. The Prolog connectives include:

ENGLISH	PREDICATE CALCULUS	Prolog
and	^	,
or	v	;
only if	←	:-
not	~	not

In Prolog, predicate names and bound variables are expressed as a sequence of alphanumeric characters beginning with an alphabetic. Variables are represented as a string of alphanumeric characters beginning (the first character, at least) with an uppercase alphabetic. Thus:

```
likes(X, susie).
```

or, better,

```
likes(Everyone, susie).
```

could represent the fact that "everyone likes Susie." Note that the scope of all variables is universal to that predicate, i.e., when a variable is used in a predicate it is understood that it is true for all the domain elements within its scope. For example,

```
likes(george, Y), likes(susie, Y).
```

represents the set of things (or people) liked by BOTH George and Susie.

Similarly, suppose it was desired to represent in Prolog the following relationships: "George likes Kate and George likes Susie." This could be stated as:

```
likes(george, kate), likes(george, susie).
```

Likewise, "George likes Kate or George likes Susie":

```
likes(george, kate); likes(george, susie).
```

Finally, "George likes Susie if George does not like Kate":

```
likes(george, susie) :- not(likes(george, kate)).
```

These examples show how the predicate calculus connectives are expressed in Prolog. The predicate names (likes), the number or order of parameters, and even whether a given predicate always has the same number of parameters are determined by the design requirements (the implicit "semantics") of the problem.

The form Prolog expressions take, as in the examples above, is a restricted form of the full predicate calculus called the "Horn Clause calculus." There are many reasons supporting this restricted form, most important is the power and computational efficiency of a *resolution refutation system*. For details see Luger (2009, Chapter 14).

A Simple Prolog Program

A Prolog program is a set of specifications in the first-order predicate calculus describing the objects and relations in a problem domain. The set of specifications is referred to as the *database* for that problem. The Prolog interpreter responds to questions about this set of specifications. Queries to the database are patterns in the same logical syntax as the database entries. The Prolog interpreter uses pattern-directed search to find whether these queries logically follow from the contents of the database.

The interpreter processes queries, searching the database in left to right depth-first order to find out whether the query is a logical consequence of the database of specifications. Prolog is primarily an interpreted language. Some versions of Prolog run in interpretive mode only, while others allow compilation of part or all of the set of specifications for faster execution. Prolog is an interactive language; the user enters queries in response to the Prolog prompt, "?-".

Let us describe a "world" consisting of George's, Kate's, and Susie's likes and dislikes. The database might contain the following set of predicates:

```
likes(george, kate).
likes(george, susie).
likes(george, wine).
likes(susie, wine).
likes(kate, gin).
likes(kate, susie).
```

This set of specifications has the obvious interpretation, or mapping, into the world of George and his friends. That world is a *model* for the database (Luger 2009, Section 2.3). The interpreter may then be asked questions:

```
?- likes(george, kate).
Yes
?- likes(kate, susie).
Yes
?- likes(george, X).
X = kate
;
X = Susie
;
X = wine
```

```
;
no
?- likes(george, beer).
no
```

Note first that in the request `likes(george, X)`, successive user prompts (`;`) cause the interpreter to return all the terms in the database specification that may be substituted for the `X` in the query. They are returned in the order in which they are found in the database: `kate` before `susie` before `wine`. Although it goes against the philosophy of nonprocedural specifications, a determined order of evaluation is a property of most interpreters implemented on sequential machines.

To summarize: further responses to queries are produced when the user prompts with the `;` (or). This forces the rejection of the current solution and a backtrack on the set of Prolog specifications for answers. Continued prompts force Prolog to find all possible solutions to the query. When no further solutions exist, the interpreter responds `no`.

This example also illustrates the *closed world assumption* or *negation as failure*. Prolog assumes that "anything is false whose opposite is not provably true." For the query `likes(george, beer)`, the interpreter looks for the predicate `likes(george, beer)` or some rule that could establish `likes(george, beer)`. Failing this, the request is false. Prolog assumes that all knowledge of the world is present in the database.

The closed world assumption introduces a number of practical and philosophical difficulties in the language. For example, failure to include a fact in the database often means that its truth is unknown; the closed world assumption treats it as false. If a predicate were omitted or there were a misspelling, such as `likes(george, beeer)`, the response remains `no`. Negation-as-failure issue is an important topic in AI research. Though negation-as-failure is a simple way to deal with the problem of unspecified knowledge, more sophisticated approaches, such as multi-valued logics (`true`, `false`, `unknown`) and nonmonotonic reasoning (see Luger 2009, Section 9.1), provide a richer interpretive context.

The Prolog expressions just seen are examples of *fact* specifications. Prolog also supports *rule* predicates to describe relationships between facts. We use the logical implication `:-` . For rules, only one predicate is permitted on the left-hand side of the *if* symbol `:-`, and this predicate must be a *positive literal*, which means it cannot have `not` in front of it. All predicate calculus expressions that contain logical implication must be reduced to this form, referred to as *Horn clause logic*. In Horn clause form, the left-hand side (conclusion) of an implication must be a single positive literal. The *Horn clause calculus* is equivalent to the full first-order predicate calculus for proofs by refutation (Luger 2009, Chapter 14).

Suppose we add to the specifications of the previous database a rule for determining whether two people are friends. This may be defined:

```
friends(X, Y) :- likes(X, Z), likes(Y, Z).
```

This expression might be interpreted as "`X` and `Y` are friends if there exists a `Z` such that `X` likes `Z` and `Y` likes `Z`." Two issues are important here. First,

because neither the predicate calculus nor Prolog has global variables, the scopes (extent of definition) of `X`, `Y`, and `Z` are limited to the `friends` rule. Second, values bound to, or unified with, `X`, `Y`, and `Z` are consistent across the entire expression. The treatment of the `friends` rule by the Prolog interpreter is seen in the following example.

With the `friends` rule added to the set of specifications of the preceding example, we can query the interpreter:

```
?- friends(george, susie).
yes
```

To solve this query, Prolog searches the database using the *backtrack* algorithm. Briefly, backtrack examines each predicate specification in the order that it was placed in the Prolog. If the variable bindings of the specification satisfy the query it accepts them. If they don't, the interpreter goes on to the next specification. If the interpreter runs into a dead end, i.e., no variable substitution satisfies it, then it backs up looking for other variable bindings for the predicates it has already satisfied. For example, using the predicate specifications of our current example, the query `friends(george, susie)` is unified with the conclusion of the rule `friends(X, Y) :- likes(X, Z), likes(Y, Z)`, with `X` as `george` and `Y` as `susie`. The interpreter looks for a `Z` such that `likes(george, Z)` is true and uses the first fact, with `Z` as `kate`.

The interpreter then tries to determine whether `likes(susie, kate)` is true. When it is found to be false, using the closed world assumption, this value for `Z` (`kate`) is rejected. The interpreter backtracks to find a second value for `Z`. `likes(george, Z)` then matches the second fact, with `Z` bound to `susie`. The interpreter then tries to match `likes(susie, susie)`. When this also fails, the interpreter goes back to the database for yet another value for `Z`. This time `wine` is found in the third predicate, and the interpreter goes on to show that `likes(susie, wine)` is true. In this case `wine` is the binding that ties `george` and `susie`.

It is important to state the relationship between universal and existential quantification in the predicate calculus and the treatment of variables in a Prolog program. When a variable is placed in the specifications of a Prolog database, it is universally quantified. For example, `likes(susie, Y)` means, according to the semantics of the previous examples, "Susie likes everyone." In the course of interpreting a query, any term, or list, or predicate from the domain of `Y`, may be bound to `Y`. Similarly, in the rule `friends(X, Y) :- likes(X, Z), likes(Y, Z)`, any `X`, `Y`, and `Z` that meets the specifications of the expression are used.

To represent an existentially quantified variable in Prolog, we may take two approaches. First, if the existential value of a variable is known, that value may be entered directly into the database. Thus, `likes(george, wine)` is an instance of `likes(george, Z)`.

Second, to find an instance of a variable that makes an expression true, we query the interpreter. For example, to find whether a `Z` exists such that `likes(george, Z)` is true, we put this query to the interpreter. It will

find whether a value of `Z` exists under which the expression is true. Some Prolog interpreters find all existentially quantified values; C-Prolog requires repeated user prompts (`;`), as shown previously, to get all values.

2.3 Creating, Changing, and Tracing a Prolog Computation

In building a Prolog program the database of specifications is created first. In an interactive environment the predicate `assert` can be used to add new predicates to the set of specifications. Thus:

```
?- assert(likes(david, sarah)).
```

adds this predicate to the computing specifications. Now, with the query:

```
?- likes(david, X).

X = sarah.
```

is returned. `assert` allows further control in adding new specifications to the database: `asserta(P)` asserts the predicate `P` at the beginning of all the predicates `P`, and `assertz(P)` adds `P` at the end of all the predicates named `P`. This is important for search priorities and building heuristics. To remove a predicate `P` from the database `retract(P)` is used. (It should be noted that in many Prologs `assert` can be unpredictable in that the exact entry time of the new predicate into the environment can vary depending on what other things are going on, affecting both the indexing of asserted clauses as well as backtracking.)

It soon becomes tedious to create a set of specifications using the predicates `assert` and `retract`. Instead, the good programmer takes her favorite editor and creates a file containing all the Prolog program's specifications. Once this file is created, call it `myfile`, and Prolog is called, then the file is placed in the database by the Prolog command `consult`. Thus:

```
?- consult(myfile).

yes
```

integrates the predicates in `myfile` into the database. A short form of the `consult` predicate, and better for adding multiple files to the database, uses the list notation, to be seen shortly:

```
?- [myfile].

yes
```

If there are any syntax errors in your Prolog code the `consult` operator will describe them at the time it is called.

The predicates `read` and `write` are important for user/system communication. `read(X)` takes the next term from the current input stream and binds it to `X`. Input expressions are terminated with a "." `write(X)` puts `X` in the output stream. If `X` is unbound then an integer preceded by an underline is printed (`_69`). This integer represents the internal bookkeeping on variables necessary in a theorem-proving environment (see Luger 2009, Chapter 14).

The Prolog predicates `see` and `tell` are used to read information from and place information into files. `see(X)` opens the file `X` and defines the current input stream as originating in `X`. If `X` is not bound to an available

file `see(X)` fails. Similarly, `tell(X)` opens a file for the output stream. If no file `X` exists, `tell(X)` creates a file named by the bound value of `X`. `seen(X)` and `told(X)` close the respective files.

A number of Prolog predicates are important in helping keep track of the state of the Prolog database as well as the state of computing about the database; the most important of these are `listing`, `trace`, and `spy`. If we use `listing(predicate_name)` where `predicate_name` is the name of a predicate, such as `friends` (above), all the clauses with that predicate name in the database are returned by the interpreter. Note that the number of arguments of the predicate is not indicated; in fact, all uses of the predicate, regardless of the number of arguments, are returned.

`trace` allows the user to monitor the progress of the Prolog interpreter. This monitoring is accomplished by printing to the output file every goal that Prolog attempts, which is often more information than the user wants to have. The tracing facilities in Prolog are often rather cryptic and take some study and experience to understand. The information available in a trace of a Prolog program usually includes the following:

- The depth level of recursive calls (marked left to right on line).
- When a goal is tried for the first time (sometimes `call` is used).
- When a goal is successfully satisfied (with an `exit`).
- When a goal has further matches possible (a `retry`).
- When a goal fails because all attempts to satisfy it have failed
- The goal `notrace` stops the exhaustive tracing.

When a more selective trace is required the goal `spy` is useful. This predicate takes a predicate name as argument but sometimes is defined as a prefix operator where the predicate to be monitored is listed after the operator. Thus, `spy member` causes the interpreter to print to output all uses of the predicate `member`. `spy` can also take a list of predicates followed by their arities: `spy[member/2, append/3]` monitors `member` with two arguments and `append` with three. `nospy` removes these spy points.

2.4 Lists and Recursion in Prolog

The previous subsections presented Prolog syntax with several simple examples. These examples introduced Prolog as an engine for computing with predicate calculus expressions (in Horn clause form). This is consistent with all the principles of predicate calculus inference presented in Luger (2009, Chapter 2). Prolog uses unification for pattern matching and returns the bindings that make an expression true. These values are unified with the variables in a particular expression and are not bound in the global environment.

Recursion is the primary control mechanism for Prolog programming. We will demonstrate this with several examples. But first we consider some simple list-processing examples. The list is a data structure consisting of ordered sets of elements (or, indeed, lists). Recursion is the natural way to process the list structure. Unification and recursion come together in list

processing in Prolog. The set of elements of a list are enclosed by brackets, [ ], and are separated by commas. Examples of Prolog lists are:

```
[1, 2, 3, 4]
[[george, kate], [allen, amy], [richard, shirley]]
[tom, dick, harry, fred]
[ ]
```

The first elements of a list may be separated from the tail of the list by the bar operator, |. The tail of a list is the list with its first element removed. For instance, when the list is `[tom,dick,harry,fred]`, the first element is `tom` and the tail is the list `[dick, harry, fred]`. Using the vertical bar operator and unification, we can break a list into its components:

```
If [tom, dick, harry, fred] is matched to [X | Y],
then X = tom and Y = [dick, harry, fred].

If [tom,dick,harry,fred] is matched to the pattern
[X, Y | Z], then X = tom , Y = dick , and Z =
[harry, fred].

If [tom, dick, harry, fred] is matched to [X, Y, Z |
W], then X = tom, Y = dick, Z = harry, and W =
[fred].

If [tom, dick, harry, fred] is matched to [W, X, Y,
Z | V], then W = tom, X = dick, Y = harry, Z = fred,
and V = [ ].

[tom, dick, harry, fred] will not match [V, W, X, Y,
Z | U].

[tom, dick, harry, fred] will match [tom, X |
[harry, fred]], to give X = dick.
```

Besides "tearing lists apart" to get at particular elements, unification can be used to "build" the list structure. For example, if `X = tom`, `Y = [dick]` when `L` unifies with `[X | Y]`, then `L` will be bound to `[tom, dick]`. Thus terms separated by commas before the | are all elements of the list, and the structure after the | is always a list, the tail of the list.

Let's take a simple example of recursive processing of lists: the `member` check. We define a predicate to determine whether an item, represented by `X`, is in a list. This predicate `member` takes two arguments, an element and a list, and is true if the element is a member of the list. For example:

```
?- member(a, [a, b, c, d, e]).
yes
?- member(a, [1, 2, 3, 4]).
no
?- member(X, [a, b, c]).
X = a
;
X = b
;
X = c
```

X is capital so considered variable

```
;
no
```

To define member recursively, we first test if **X** is the first item in the list:

```
member(X, [X | T]).
```

This tests whether **X** and the first element of the list are identical. Not that this pattern will match no matter what X is bound to: an atom, a list, whatever! If the two are not identical, then it is natural to check whether **X** is an element of the rest (**T**) of the list. This is defined by:

```
member(X, [Y | T]) :- member(X, T).
```

The two lines of Prolog for checking list membership are then:

```
member(X, [X | T]).
member(X, [Y | T]) :- member(X, T).
```

This example illustrates the importance of Prolog's built-in order of search with the terminating condition placed before the recursive call, that is, to be tested before the algorithm recurs. If the order of the predicates is reversed, the terminating condition may never be checked. We now trace `member(c,[a,b,c])`, with numbering:

```
1: member(X, [X | T]).
2: member(X, [Y | T]) :- member(X, T).
?- member(c, [a, b, c]).
    call 1. fail, since c <> a
    call 2. X = c, Y = a, T = [b, c],
                  member(c, | [b,c])?
         call 1. fail, since c <> b
         call 2. X = c, Y = b, T = [c],
                       member(c, | [c])?
              call 1. success, c = c
         yes (to second call 2.)
    yes (to first call 2.)
yes
```

Good Prolog style suggests the use of *anonymous variables*. These serve as an indication to the programmer and interpreter that certain variables are used solely for pattern-matching purposes, with the variable binding itself not part of the computation process. Thus, when we test whether the element **X** is the same as the first item in the list we usually say: `member(X, [X|_])`. The use of the _ indicates that even though the tail of the list plays a crucial part in the unification of the query, the content of the tail of the list is unimportant. In the `member` check the anonymous variable should be used in the recursive statement as well, where the value of the head of the list is unimportant:

```
member(X, [X | _]).
member(X, [_ | T]) :- member(X, T).
```

Writing out a list one element to a line is a nice exercise for understanding both lists and recursive control. Suppose we wish to write out the list `[a,b,c,d]`. We could define the recursive command:

```
writelist([ ]).
writelist([H | T]) :- write(H), nl, writelist(T).
```

This predicate writes one element of the list on each line, as `nl` requires the output stream controller to begin a new line.

If we wish to write out a list in reversed order the recursive predicate must come before the `write` command. This guarantees that the list is traversed to the end before any element is written. At that time the last element of the list is written followed by each preceding element as the recursive control comes back up to the top. A reverse write of a list would be:

```
reverse_writelist([ ]).
reverse_writelist([H | T]) :- reverse_writelist(T),
     write(H), nl.
```

The reader should run `writelist` and `reverse_writelist` with trace to observe the behavior of these predicates.

2.5 Structured Representations and Inheritance Search

Semantic Nets in Prolog

Structured representations are an important component of the AI representational toolkit (Collins and Quillian 1969, Luger 2009). They also support many of the design patterns mentioned in Chapter 1. In this and the following section we consider two structured representations, the *semantic net*, and *frames* that are used almost ubiquitously in AI. We now propose a simple semantic network representational structure in Prolog and use recursive search to implement inheritance. Our language ignores the important distinction between classes and instances. This restriction simplifies the implementation of inheritance.

In the semantic net of Figure 2.1, nodes represent individuals such as the canary `tweety` and classes such as `ostrich`, `crow`, `robin`, `bird`, and `vertebrate`. `isa` links represent the class hierarchy relationship. We adopt canonical forms for the data relationships within the net. We use an `isa(Type, Parent)` predicate to indicate that `Type` is a member of `Parent` and a `hasprop(Object, Property, Value)` predicate to represent property relations. `hasprop` indicates that `Object` has `Property` with `Value`. `Object` and `Value` are nodes in the network, and `Property` is the name of the link that joins them.

A partial list of predicates describing the bird hierarchy of Figure 2.1 is:

```
isa(canary, bird).      hasprop(tweety, color, white)
isa(robin, bird).       hasprop(robin, color, red).
isa(ostrich, bird).     hasprop(canary, color, yellow).
isa(penguin, bird).     hasprop(penguin, color, brown).
isa(bird, animal).      hasprop(bird, travel, fly).
isa(fish, animal).      hasprop(ostrich, travel, walk).
isa(opus, penguin).     hasprop(fish, travel, swim).
isa(tweety, canary).    hasprop(robin, sound, sing).
hasprop(canary, sound, sing).
hasprop(bird, cover, feathers).
hasprop(animal, cover, skin).
```

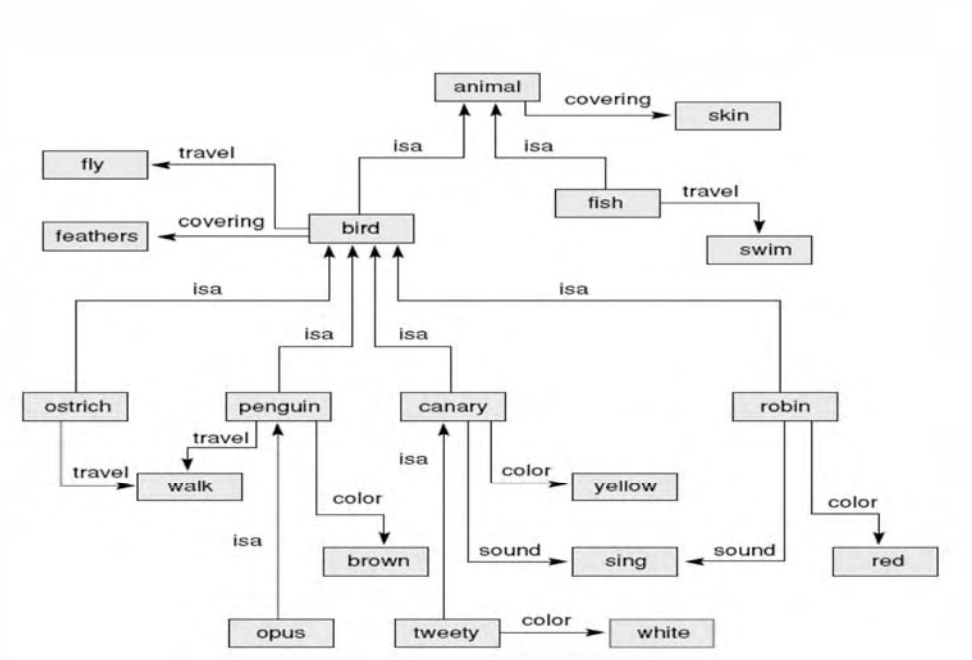

Figure 2.1 A semantic net for a bird hierarchy reflecting the Prolog code.

We create a recursive search algorithm to find whether an object in our semantic net has a particular property. Properties are stored in the net at the most general level at which they are true. Through inheritance, an individual or subclass acquires the properties of its superclasses. Thus the property `fly` holds for `bird` and all its subclasses. Exceptions are located at the specific level of the exception. Thus, `ostrich` and `penguin` travel by walking instead of flying. The `hasproperty` predicate begins search at a particular object. If the information is not directly attached to that object, `hasproperty` follows `isa` links to superclasses. If no more superclasses exist and `hasproperty` has not located the property, it fails.

```
hasproperty(Object, Property, Value) :-
    hasprop(Object, Property, Value).
hasproperty(Object, Property, Value) :-
    isa(Object, Parent),
    hasproperty(Parent, Property, Value).
```

`hasproperty` searches the inheritance hierarchy in a depth-first fashion. In the next section, we show how inheritance can be applied to a frame-based representation with both single and multiple-inheritance relations.

Frames in Prolog

Semantic nets can be partitioned, with additional information added to node descriptions, to give them a frame-like structure (Minsky 1975, Luger 2009). We present the bird example again using frames, where each frame represents a collection of relationships of the semantic net and the `isa` slots of the frame define the frame hierarchy as in Figure 2.2.

The first slot of each of the frames name that node, for example, `name(tweety)` or `name(vertebrate)`. The second slot gives the inheritance links between that node and its parents. Because our example

has a tree structure, each node has only one link, the `isa` predicate with one argument. The third slot in the node's frame is a list of features that describe that node. In this list we use any Prolog predicate such as `flies`, `feathers`, or `color(brown)`. The final slot in the frame is the list of exceptions and default values for the node, again either a single word or predicate indicating a property.

In our frame language, each frame organizes its slot names into lists of properties and default values. This allows us to distinguish these different types of knowledge and give them different behaviors in the inheritance hierarchy. Although our implementation allows subclasses to inherit properties from both lists, other representations are possible and may be useful in certain applications. We may wish to specify that only default values are inherited. Or we may wish to build a third list containing the properties of the class itself rather than the members, sometimes called *class values*. For example, we may wish to state that the class `canary` names a species of `songbird`. This should not be inherited by subclasses or instances: `tweety` does not name a species of `songbird`. Further extensions to this example are suggested in the exercises.

We now represent the relationships in Figure 2.2 with the Prolog fact predicate `frame` with four arguments. We may use the methods suggested in Chapter 5 to check the parameters of the frame predicate for appropriate type, for instance, to ensure that the third frame slot is a list that contains only values from a fixed list of properties.

```
frame(name(bird),
      isa(animal),
      [travel(flies), feathers],
      [ ]).
frame(name(penguin),
      isa(bird),
      [color(brown)],
      [travel(walks)]).
frame(name(canary),
      isa(bird),
      [color(yellow), call(sing)],
      [size(small)]).
frame(name(tweety),
      isa(canary),
      [ ],
      [color(white)]).
```

Once the full set of descriptions and inheritance relationships are defined for the frame of Figure 2.2, we create procedures to infer properties from this representation:

```
get(Prop, Object) :-
      frame(name(Object), _, List_of_properties,_),
      member(Prop, List_of_properties).
```

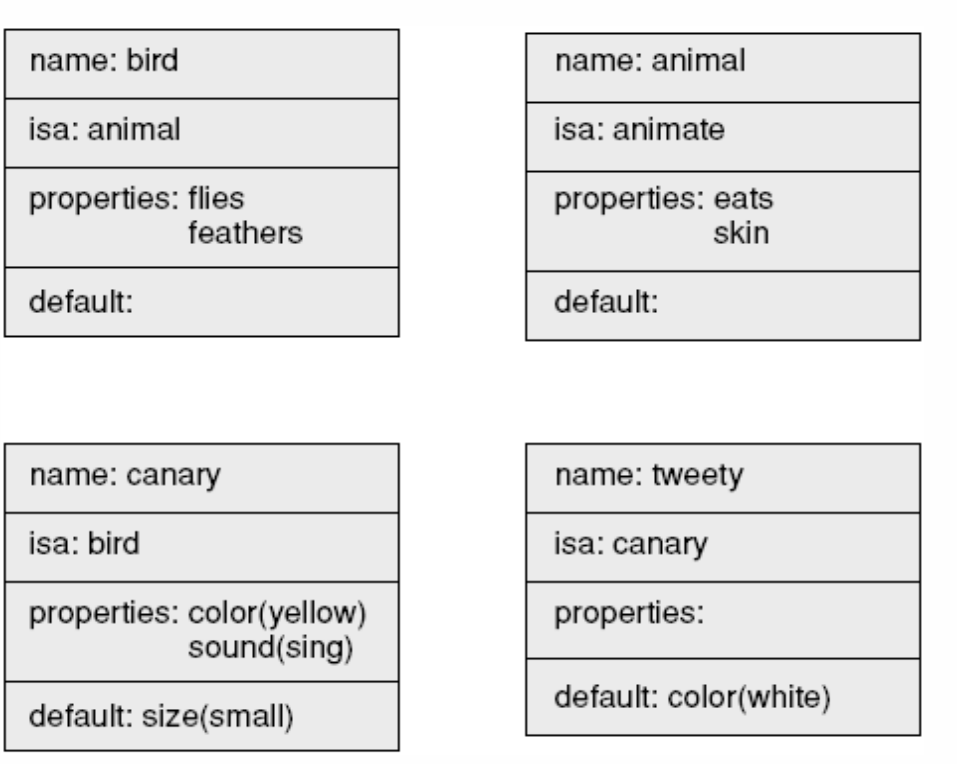

Figure 2.2 A frame system reflecting the Prolog code in the text.

```
get(Prop, Object) :-
        frame(name(Object), _, _ List_of_defaults),
        member(Prop, List_of_defaults).
get(Prop, Object) :-
        frame(name(Object), isa(Parent), _, _),
        get(Prop, Parent).
```

If the frame structure allows multiple inheritance of properties, we make this change both in our representation and in our search strategy. First, in the frame representation we make the argument of the `isa` predicate a list of superclasses of the `Object`. Thus, each superclass in the list is a parent of the entity named in the first argument of `frame`. If `opus` is a `penguin` and a `cartoon_char` we represent this:

```
frame(name(opus),
        isa([penguin, cartoon_char]),
        [color(black)],
        [ ]).
```

Now, we test for properties of `opus` by recurring up the `isa` hierarchy for both `penguin` and `cartoon_char`. We add the following `get` definition between the third and fourth `get` predicates of the previous example.

```
get(Prop, Object) :-
        frame(name(Object), isa(List), _, _),
        get_multiple(Prop, List).
```

We define get_multiple by:

```
get_multiple(Prop, [Parent  _]) :-
     get(Prop, Parent).
get_multiple(Prop, [_ Rest]) :-
     get_multiple(Prop, Rest).
```

With this inheritance preference, properties of `penguin` and its superclasses will be examined before those of `cartoon_char`.

Finally, any Prolog procedure may be attached to a frame slot. As we have built the frame representation in our examples, this would entail adding a Prolog rule, or list of Prolog rules, as a parameter of frame. This is accomplished by enclosing the entire rule in parentheses, as we will see for rules in `exshell` in Chapter 6, and making this structure an argument of the frame predicate. For example, we could design a list of response rules for `opus`, giving him different responses for different questions.

This list of rules, each rule in parentheses, would then become a parameter of the frame and, depending on the value of `X` passed to the `opus` frame, would define the appropriate response. More complex examples could be rules describing the control of a thermostat or creating a graphic image appropriate to a set of values. Examples of this are presented in both Lisp (Chapter 17) and Java (Chapter 21) where attached procedures, often called *methods*, play an important role in object-oriented representations.

Exercises

1. Create a relational database in Prolog. Represent the data tuples as facts and the constraints on the data tuples as rules. Suitable examples might be from stock in a department store or records in a personnel office.

2. Write the "member check" program in Prolog. What happens when an item is not in the list? Query to the "member" specification to break a list into its component elements.

3. Design a Prolog program `unique(Bag, Set)` that takes a *bag* (a list that may contain duplicate elements) and returns a *set* (no elements are repeated).

4. Write a Prolog program to count the elements in a list (a list within the list counts as one element). Write a program to count the atoms in a list (count the elements within any sublist). Hint: several meta-predicates such as `atom()` can be helpful.

5. Implement a frame system with inheritance that supports the definition of three kinds of slots: properties of a class that may be inherited by subclasses, properties that are inherited by instances of the class but not by subclasses, and properties of the class and its subclasses that are not inherited by instances (class properties). Discuss the benefits, uses, and problems with this distinction.

3 Abstract Data Types and Search

Chapter Objectives

Prolog's graph search representations were described and built:
- Lists

A recursive tree-walk algorithm
The cut predicate, !, for Prolog was presented:
- Controls the interpreter's backtracking

Limits variable instantiations, and thus
- May prevent the interpreter from computing good solutions

Demonstrated procedural abstraction and information hiding with Abstract Data Types
The stack operators
- The queue operators
- The priority queue operators
- Sets

Chapter Contents

3.1 Introduction

Recursion-Based Graph Search

We next introduce the 3 x 3 knight's tour problem, create a predicate calculus based representation of problem states, and a recursive search of its state space. The chess knight can move on a restricted board as on any regular chessboard: two squares one direction (horizontally or vertically) and one in the other (vertically or horizontally). Thus, the knight can go from square 1 to either square 6 or 8 or from square 9 to either 2 or 4. We ask if the knight can generate a sequence on legal moves from one square to another on this restricted chessboard, from square 1 to 9, for example. The knight's moves are represented in Prolog using `move` facts, Figure 3.1.

The `path` predicate defines an algorithm for a path between its two arguments, the present state, `X`, and the goal that it wants to achieve, `Y`. To do this it first tests whether it is where it wants to be, `path(Z, Z)`, and if not, looks for a state, `W`, to move to

The Prolog search defined by `path` is a recursive, depth-first, left-to-right, tree walk. As shown in Section 2.3, `assert` is a built-in Prolog predicate that always succeeds and has the side effect of placing its argument in the database of specifications. The `been` predicate is used to record each state as it is visited and then `not(been(X))` determines, with each new state found whether that state has been previously visited, thus avoiding looping within the search space.

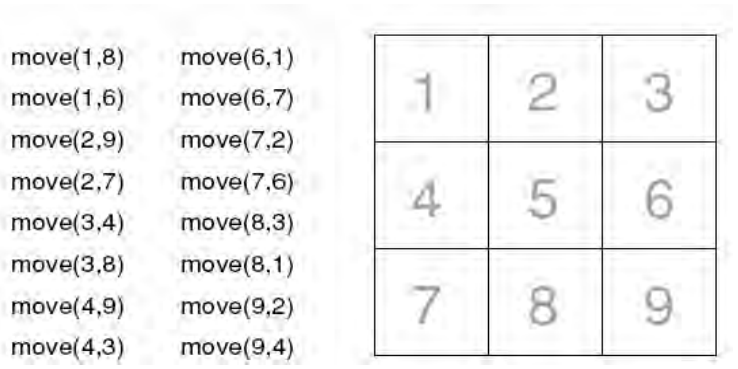

Figure 3.1. The 3 x 3 chessboard and set of legal moves expressed as Prolog facts.

```
path(Z, Z).
path(X, Y) :-
     move(X, W), not(been(W)), assert(been(W)),
     path(W, Y).
```

This use of the `been` predicate violates good programming practice in that it uses global side-effects to control search. `been(3)`, when asserted into the database, is a fact available to any other predicate and, as such, has global extension. We created `been` to modify the program execution.

A more sophisticated method for control of search is to create a list that keeps track of visited states. We create this list and make it the third argument of the `path` predicate. As each new state is encountered, the `member` predicate, Section 2.3, checks whether or not it is already a visited state. If the state is not a member of the list we put it on this list in the order in which it was encountered, the most recent state encountered the head of the list. If the new state is on the list of already visited states, the `path` predicate backtracks looking for new non-visited states. This approach remedies the problems of using global `been(W)`. The following clauses implement depth-first left-to right graph search with backtracking.

```
path(Z, Z, L).
path(X, Y, L) :-
     move(X, Z), not(member(Z, L)),
     path(Z, Y, [Z|L]).
```

The third parameter of `path` is the variable representing the list of visited states. When a new state not already on the list of visited states `L`, it is placed on the front of the state list `[Z | L]` for the next `path` call. It should be noted that all the parameters of `path` are local and their current values depend on where they are called in the graph search. Each successful recursive call adds a state to this list. If all continuations from a certain state fail, then that particular `path` call fails and the interpreter backs up to the parent call. Thus, states are added to and deleted from this state list as the backtracking search moves through the graph.

When the `path` call finally succeeds, the first two parameters are identical and the third parameter is the list of states visited, in reverse order. Thus we can print out the steps of the solution. The call to the Prolog interpreter

`path(X,Y,[X])`, where `X` and `Y` are replaced by numbers between 1 and 9, finds a path from state `X` to state `Y`, if the path exists. The third parameter initializes the path list with the starting state `X`. Note that there is no typing distinction in Prolog: the first two parameters are any representation of states in the problem space and the third is a list of states. Unification makes this generalization of pattern matching across data types possible. Thus, `path` is a general depth-first search algorithm that may be used with any graph. In Chapter 4 we use this algorithm to implement a solution to the farmer, wolf, goat, and cabbage problem, with different state specifications of state in the call to `path`.

We now present a solution for the 3 x 3 knight's tour. (It is an exercise to solve the full 8 x 8 knight's tour problem in Prolog. We refer to the two parts of the `path` algorithm by number:

```
1. is path(Z, Z, L).
2. is path(X, Y, L)  :-
        move(X, Z), not(member(Z, L)),
        path(Z, Y, [Z | L]).

?- path(1, 3, [1]).
   path(1, 3, [1]) attempts to match 1. fail 1<>3.
   path(1, 3, [1]) matches 2. X=1, Y=3, L=[1]
        move(1, Z) matches, Z=6,
        not(member(6,[1]))=true,
        call path(6, 3, [6,1]
   path(6, 3, [6,1]) trys to match 1. fail 6<>3.
   path(6, 3, [6,1]) calls 2. X=6, Y=3, L=[6, 1].
        move(6, Z) matches, Z=7,
        not(member(7, [6,1]))=true,
        call path(7, 3, [7,6,1])
   path(7, 3, [7,6,1]) trys to match 1. fail 7<>3.
   path(7, 3, [7,6,1]) in 2: X=7, Y=3, L=[7,6,1].
        move(7, Z) matches Z=6,
        not(member(6, [7,6,1])) fails,   backtrack!
        move(7, Z) matches, Z = 2,
        not(member(2, [7,6,1])) is true
        call path(2, 3, [2,7,6,1])
   path(2, 3, [2,7,6,1]) attempts 1, fail, 2 <> 3.
   path matches 2, X in 2: Y is 3, L is [2, 7, 6, 1]
        move(2, Z) matches, Z=7,
        not(member(…)) fails, backtrack!
        move(2, Z) matches Z=9,
        not(member(…)) is true,
        call path(9, 3, [9,2,7,6,1])
   path(9, 3, [9,2,7,6,1]) fails 1, 9<>3.
```

```
path matches 2, X=9, Y=3, L=[9,2,7,6,1]
     move(9, Z) matches Z = 4,
     not(member(…)) is true,
     call path(4, 3, [4,9,2,7,6,1])
path(4, 3, [4,9,2,7,6,1])fails 1, 4<>3.
path matches 2, X=4, Y=3, L is [4,9,2,7,6,1]
     move(4, Z) matches Z = 3,
     not(member(…)) true,
     call path(3, 3, [3,4,9,2,7,6,1])
path(3, 3, [3,4,9,2,7,6,1]) attempts 1, true, 3=3
```

The recursive `path` call then returns `yes` for each of its calls.

In summary, the recursive `path` call is a *shell* or general control structure for search in a graph: in `path(X, Y, L)`, `X` is the present state; `Y` is the goal state. When `X` and `Y` are identical, the recursion terminates. `L` is the list of states on the current path towards state `Y`, and as each new state `Z` is found with the call `move(X, Z)` it is placed on front of the list: `[Z | L]`. The state list is checked, using `not(member(Z, L))`, to be sure the path does not loop.

In Chapter 4, we generalize this approach creating a closed list that retains all the states visited and does not remove visited states when the path predicate backtracks from a state that has no "useful" children. The difference between the state list `L` in the `path` call above and the closed set in Chapter 4 is that closed records all states visited, while the state list L keeps track of only the present path of states.

3.2 Using `cut` to Control Search in Prolog

The predicate *cut* is represented by an exclamation point, !. The syntax for cut is that of a goal with no arguments. Cut has several side effects: first, when originally encountered it always succeeds, and second, if it is "failed back to" in backtracking, it causes the entire goal in which it is contained to fail. For a simple example of the effect of the cut, we create a two-move `path` call from the knight's tour example that we just presented. Consider the predicate path2:

```
path2(X, Y) :- move(X, Z), move(Z, Y).
```

There is a two-move path between X and Y if there exists an intermediate state Z between them. For this example, assume part of the knight's tour database:

```
move(1, 8).
move(6, 7).
move(6, 1).
move(8, 3).
move(8, 1).
```

The interpreter finds all the two-move paths from 1; there are four:

```
?- path2(1, W).
W = 7
```

```
;
W = 1
;
W = 3
;
W = 1
;
no
```

When `path2` is altered with cut, only two answers result:

```
path2(X, Y) :- move(X, Z), !, move(Z, Y)
?- path2(1, W).
W = 7
;
W = 1
;
no
```

The `no` response happens because variable `Z` takes on only one value (the first value it is bound to), namely 6. Once the first subgoal succeeds, `Z` is bound to 6 and the cut is encountered. This prohibits further backtracking using the first `move` subgoal and allowing any further bindings for the variable `Z`.

There are several justifications for the use of cut in Prolog programming. First, as this example demonstrated, it allows the programmer to control precisely the shape of the search tree. When further (exhaustive) search is not required, the tree can be explicitly pruned at that point. This allows Prolog code to have the flavor of function calling: when one set of values (bindings) is "returned" by a Prolog predicate (or set of predicates) and the cut is encountered, the interpreter does not search for any other unifications. Thus, if that set of values does not lead to a solution then no further values are attempted. Of course, in the context of the mathematical foundations of the predicate calculus itself, cut may prevent the computation of possible interpretations of the particular predicate calculus and as a result eliminate a possible answer or *model*, (Luger 2009, Sections 2.3, 14.3).

A second use of the cut controls recursive calls. For example, in the `path` call:

```
path(Z, Z, L).
path(X, Z, L) :- move(X, Y), not(member(Y, L)),
      path(Y, Z, [Y|L]),!.
```

The addition of cut means that (at most) one solution to the graph search is produced. This single solution is produced because further solutions occur after the clause `path(Z, Z, L)` is satisfied. If the user asks for more solutions, `path(Z, Z, L)` fails, and the second `path` call is reinvoked to continue the (exhaustive) search of the graph. When the cut is placed after the recursive `path` call, the call cannot be reentered (backed into) for further search.

Important side effects of the cut are to make the program run faster and to

conserve memory locations. When cut is used within a predicate, the pointers in memory needed for backtracking to predicates to the left of the cut are not created. This is, of course, because they will never be needed. Thus, cut produces the desired solution, and only the desired solution, with a more efficient use of memory.

The cut can also be used with recursion to reinitialize the path call for further search within the graph. This will be demonstrated with the general search algorithms presented in Chapter 4. For this purpose we also need to develop several abstract data types.

3.3 Abstract Data Types (ADTs) in Prolog

Programming, in almost any environment, is enhanced by creating procedural abstractions and by hiding information. Because the *set*, *stack*, *queue*, and *priority queue* data structures are important support constructs for graph search algorithms, a major component of AI problem solving, we build them in Prolog in the present section. We will use these ADTs in the design of the Prolog search algorithms presented in Chapter 4.

Since lists, recursion, and pattern matching, as emphasized throughout this book, are the primary tools for building and searching graph structures. These are the pieces with which we build our ADTs. All list handling and recursive processing that define the ADT are "hidden" within the ADT abstraction, quite different than the normal static data structure.

The ADT Stack

A *stack* is a linear structure with access at one end only. Thus all elements must be added to, *pushed*, and removed, *popped*, from the structure at that access end. The stack is sometimes referred to as a last-in-first-out (LIFO) data structure. We will see its use with depth-first search in Chapter 4. The operators that we will define for a stack are:

1. Test whether the stack is empty.
2. *Push* an element onto the stack.
3. *Pop* or remove, the top element from the stack.
4. *Peek* (often called *Top*) to see the top element on the stack without popping it.
5. `Member_stack`, checks whether an element is in the stack.
6. `Add_list_to stack`, adds a list of elements to the stack.

Operators 5 and 6 may be built from 1–4.

We now build these operators in Prolog, using the list primitives:

1. `empty_stack([ ]).`

This predicate can be used either to test a stack to see whether it is empty or to generate a new empty stack.

2–4. `stack(Top, Stack, [Top | Stack]).`

This predicate performs the push, pop, and peek predicates depending on the variable bindings of its arguments. For instance, push produces a new stack as the third argument when the first two arguments are bound. Likewise, pop produces the top element of the stack when the third argument is bound to the stack. The second argument will then be bound to the new stack, once the

top element is popped. Finally, if we keep the stack as the third argument, the first argument lets us peek at its top element.

5. `member_stack(Element, Stack) :-`
   `member(Element, Stack).`

This allows us to determine whether an element is a member of the stack. Of course, the same result could be produced by creating a recursive call that peeked at the next element of the stack and then, if this element did not match Element, popped the stack. This would continue until the empty stack predicate was true.

6. `add_list_to_stack(List, Stack, Result) :-`
   `append(List, Stack, Result).`

`List` is added to `Stack` to produce `Result`, a new stack. Of course, the same result could be obtained by popping `List` (until empty) and pushing each element onto a temporary stack. We would then pop the temporary stack and push each element onto the `Stack` until `empty_stack` is true for the temporary stack. `append` is described in detail in Chapter 10.

A final predicate for printing a stack in reverse order is `reverse_print_stack`. This is very useful when a stack has, in reversed order, the current path from the start state to the present state of the graph search. We will see several examples of this in Chapter 4.

```
reverse_print_stack(S) :-
     empty_stack(S).
reverse_print_stack(S) :-
     stack(E, Rest, S),
     reverse_print_stack(Rest),
     write(E), nl.
```

The ADT Queue

A *queue* is a first-in-first-out (FIFO) data structure. It is often characterized as a list where elements are taken off or *dequeued* from one end and elements are added to or *enqueued* at the other end. The queue is used for defining breadth-first search in Chapter 4. The queue operators are:

1. `empty_queue([ ]).`

This predicate tests whether a queue is empty or initializes a new empty queue.

2. `enqueue(E, [ ], [E]).`
   `enqueue(E, [H | T], [H | Tnew]) :-`
   `enqueue(E, T, Tnew).`

This recursive predicate adds the element **E** to a queue, the second argument. The new augmented queue is the third argument.

3. `dequeue(E, [E | T], T).`

This predicate produces a new queue, the third argument, which is the result of taking the next element, the first argument, off the original queue, the second argument.

4. `dequeue(E, [E | T], _).`

This predicate lets us peek at the next element, **E**, of the queue.

5. `member_queue(Element, Queue) :-`
   `member(Element, Queue).`

This tests whether `Element` is a member of `Queue`.

6. `add_list_to_queue(List, Queue, Newqueue) :-`
   `append(Queue, List, Newqueue).`

This predicate enqueues an entire list of elements. Of course, 5 and 6 can be created using 1–4; `append` is presented in Chapter 10.

The ADT Priority Queue

A *priority queue* orders the elements of a regular queue so that each new element added to the priority queue is placed in its sorted order, with the "best" element first. The *dequeue* operator removes the "best" sorted element from the priority queue. We will use the priority queue in the design of the best-first search algorithm in Chapter 4.

Because the priority queue is a sorted queue, many of its operators are the same as the queue operators, in particular, `empty_queue`, `member_queue`, and `dequeue` (the "best" of the sorted elements will be next for the `dequeue`). `enqueue` in a priority queue is the `insert_pq` operator, as each new item is placed in its proper sorted order.

```
insert_pq(State, [ ], [State]) :- !.
insert_pq(State, [H | Tail], [State, H | Tail]) :-
     precedes(State, H).
insert_pq(State, [H | T], [H | Tnew]) :-
     insert_pq(State, T, Tnew).
precedes(X, Y) :- X < Y.       % < depends on problem
```

The first argument of this predicate is the new element that is to be inserted. The second argument is the previous priority queue, and the third argument is the augmented priority queue. The `precedes` predicate checks that the order of elements is preserved. Another priority queue operator is `insert_list_pq`. This predicate is used to merge an unsorted list or set of elements into the priority queue, as is necessary when adding the children of a state to the priority queue for best-first search, Chapter 4. `insert_list_pq` uses `insert_pq` to put each individual new item into the priority queue:

```
insert_list_pq([ ], L, L).
insert_list_pq([State | Tail], L, New_L) :-
     insert_pq(State, L, L2),
     insert_list_pq(Tail, L2, New_L).
```

The ADT Set

Finally, we describe the ADT `set`. A *set* is a collection of elements with no element repeated. Sets can be used for collecting all the children of a state or for maintaining the set of all states visited while executing a search algorithm.

In Prolog a set of elements, e.g., {a,b}, may be represented as a list, `[a,b]`, with the order of the list not important. The set operators include `empty_set`, `member_set`, `delete_if_in`, and `add_if_not_in`. We also include the traditional operators for

combining and comparing sets, including `union`, `intersection`, `set_difference`, `subset`, and `equal_set`.

```
empty_set([ ]).
member_set(E, S) :-
     member(E, S).
delete_if_in_set(E, [ ], [ ]).
delete_if_in_set(E, [E | T], T) :- !.
delete_if_in_set(E, [H | T], [H | T_new]) :-
     delete_if_in_set(E, T, T_new), !.
add_if_not_in_set(X, S, S) :-
     member(X, S), !.
add_if_not_in_set(X, S, [X | S]).
union([ ], S, S).
union([H | T], S, S_new) :-
     union(T, S, S2),
     add_if_not_in_set(H, S2, S_new),!.
subset([ ], _).
subset([H | T], S) :-
     member_set(H, S),
     subset(T, S).
intersection([ ], _, [ ]).
intersection([H | T], S, [H | S_new]) :-
     member_set(H, S),
     intersection(T, S, S_new), !.
intersection([_ | T], S, S_new) :-
     intersection(T, S, S_new), !.
set_difference([ ], _, [ ]).
set_difference([H | T], S, T_new) :-
     member_set(H, S),
     set_difference(T, S, T_new), !.
set_difference([H | T], S, [H | T_new]) :-
     set_difference(T, S, T_new), !.
equal_set(S1, S2) :-
     subset(S1, S2),
     subset(S2, S1).
```

In Chapters 4 and 5 we use many of these abstract data types to build more complex graph search algorithms and meta-interpreters in Prolog. For example, the stack and queue ADTs are used to build the "open" list that organizes depth-first and breadth-first search. The set ADTs coordinate the "closed" list that helps prevent cycles in a search.

Exercises

1. Write Prolog code to solve the full 8 X 8 knight's tour problem. This will require a lot of effort if all the `move(X, Y)` facts on the full chessboard are itemized. It is much better to create a set of eight predicates that capture the general rules for moving the knight on the full chessboard. You will also have to create a new representation for the squares on the board. Hint: consider a predicate containing the two element order pair, for example, `state(Row, Column)`.

2. Take the path algorithm presented for the knight's tour problem in the text. Rewrite the `path` call of the recursive code given in Section 3.1 to the following form:

```
path(X, Y) :- path(X, W), move(W, Y).
```

Examine the trace of this execution and describe what is happening.

3. Create a three step path predicate for the knight's tour:

```
path3(X, Y) :- move(X, Z), move(Z, W), move (W, Y).
```

Create a tree that demonstrates the full search for the `path3` predicate. Place the cut operator between the second and third `move` predicates. Show how this prunes the tree. Next place the cut between the first and second `move` predicates and again demonstrate how the tree is pruned. Finally, put two cuts within the `path3` predicate and show how this prunes the search.

4. Write and test the ADTs presented in Section 3.3. `trace` will let you monitor the Prolog environment as the ADTs execute.

4 Depth-, Breadth-, and Best-First Search Using the Production System Design Pattern

Chapter Objectives

A production system was defined and examples given:
- Production rule sets

Control strategies

A production system written in Prolog was presented:
- A rule set and control strategy for the Farmer Wolf, Goat, and Cabbage problem

Search strategies for the production system created using Abstract Data Types

Depth-first search and the stack operators
- Breadth-first search and the queue operators
- Best first search and the priority queue operators
- Sets were used for the closed list in all searches

Chapter Contents

4.1 Production System Search in Prolog

The Production System

The *production system* (Luger 2009, Section 6.2) is a model of computation that has proved particularly important in AI, both for implementing search algorithms and for modeling human problem solving behavior. A production system provides pattern-directed control of a problem-solving process and consists of a set of *production rules*, a *working memory*, and a *recognize–act* control cycle.

A *production system* is defined by:

The set of production rules. These are often simply called *productions*. A production is a *condition–action* pair and defines a single chunk of problem-solving knowledge. The *condition part* of the rule is a pattern that determines when that rule may be applied by matching data in the working memory. The *action part* of the rule defines the associated problem-solving step.

Working memory contains a description of the *current state of the world* in a reasoning process. This description is a pattern that, in *data-driven reasoning*, is matched against the condition part of a production to select appropriate problem-solving actions. The actions of production rules are specifically designed to alter the contents of working memory, leading to the next phase of the recognize-act cycle.

The recognize–act cycle. The control structure for a production system is simple: *working memory* is initialized with the beginning problem description. The current state of the problem solving is maintained as a set of patterns in working memory. These patterns are matched against the conditions of the

production rules; this produces a subset of the production rules, called the *conflict set*, whose conditions match the patterns in working memory. One of the productions in the conflict set is then selected (*conflict resolution*) and the production is *fired*. After the selected production rule is fired, the control cycle repeats with the modified working memory. The process terminates when the contents of working memory do not match any rule conditions.

Conflict resolution chooses a rule from the conflict set for firing. Conflict resolution strategies may be simple, such as selecting the first rule whose condition matches the state of the world, or may involve complex rule selection heuristics. The *pure* production system model has no mechanism for recovering from dead ends in the search; it simply continues until no more productions are enabled and halts. Many practical implementations of production systems allow backtracking to a previous state of working memory in such situations. A schematic drawing of a production system is presented in Figure 4.1.

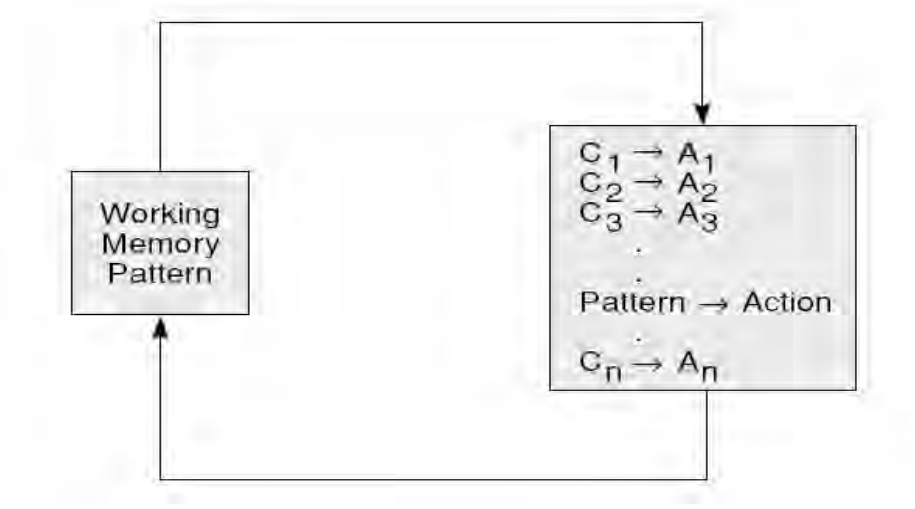

Figure 4.1. The production system. Control loops from the working memory through the production rules until no rule matches a working memory pattern.

Example 4.1: The Knight's Tour Revisited

The 3 x 3 knight's tour problem may be solved with a production system, Figure 4.1. Each move can be represented as a rule whose condition is the location of the knight on a particular square and whose action moves the knight to another square. Sixteen productions, presented in Table 4.1, represent all possible moves of the knight.

We next specify a recursive procedure to implement a control algorithm for the production system. We will use the recursive path algorithm of Section 3.1, where the third argument of the path predicate is the list of already visited states. Because `path(Z, Z, L)` will unify only with predicates whose first two arguments are identical, such as `path(3, 3, _)` or `path(5, 5, _)`, it defines the desired terminating condition. If `path(X, X, L)` does not succeed we look at the production rules for a next state and then recur.

RULE #	CONDITION	ACTION
1	knight on square 1	move knight to square 8
2	knight on square 1	move knight to square 6
3	knight on square 2	move knight to square 9
4	knight on square 2	move knight to square 7
5	knight on square 3	move knight to square 4
6	knight on square 3	move knight to square 8
7	knight on square 4	move knight to square 9
8	knight on square 4	move knight to square 3
9	knight on square 6	move knight to square 1
10	knight on square 6	move knight to square 7
11	knight on square 7	move knight to square 2
12	knight on square 7	move knight to square 6
13	knight on square 8	move knight to square 3
14	knight on square 8	move knight to square 1
15	knight on square 9	move knight to square 2
16	knight on square 9	move knight to square 4

Table 4.1. Production rules for the 3 x 3 knight tour problem.

The general recursive path definition is given by two predicate calculus formulas:

```
path(Z, Z, L).
path(X, Y, L) :-
     move(X, Z), not(member(Z, L)),
     path(Z, Y, [Z | L]).
```

Working memory, represented by the parameters of the recursive `path` predicate, contains both the current board state and the goal state. The control regime applies rules until the current state equals the goal state and then halts. A simple conflict resolution scheme would fire the first rule that did not cause the search to loop. Because the search may lead to dead ends (from which every possible move leads to a previously visited state and thus a loop), the control regime must also allow backtracking; an execution of this production system that determines whether a path exists from square 1 to square 2 is charted in Table 4.2.

Production systems are capable of generating infinite loops when searching a state space graph. These loops are particularly difficult to spot in a production system because the rules can fire in any order. That is, looping may appear in the execution of the system, but it cannot easily be found from a syntactic inspection of the rule set.

LOOP	CURRENT	GOAL	CONFLICT RULES	USE RULE
0	1	2	1, 2	1
1	8	2	13, 14	13
2	3	2	5, 6	5
3	4	2	7, 8	7
4	9	2	15, 16	15
5	2	2	No Rules Match	Halt

Table 4.2. The iterations of the production system finding a path from square 1 to square 2.

For example, with the "move" rules of the knight's tour problem ordered as in Table 4.1 and a conflict resolution strategy of selecting the first match, the pattern `move(2, X)` would match with `move(2, 9)`, indicating a move to square 9. On the next iteration, the pattern `move(9, X)` would match with `move(9, 2)`, taking the search back to square 2, causing a loop. The `not(member(Z, L))` will check the list of visited states. The actual conflict resolution strategy was therefore: *select the first matching move that leads to an unvisited state*. In a production system, the proper place for recording such case-specific data as a list of previously visited states is not a global closed list but within the working memory itself, as we see in the next sections where the parameters of the `path` call make up the content of working memory.

4.2 A Production System Solution to the FWGC Problem

Example 4.2: The Farmer, Wolf, Goat, and Cabbage Problem

In Section 4.1 we described the production system and demonstrated a simple depth-first search for the restricted Knight's Tour problem. In this section we write a production system solution to the farmer, wolf, goat, and cabbage (FWGC) problem. In Section 4.3 we use the simple abstract data types created in Chapter 3 to create depth-, breadth-, and best-first solutions for production system problems. The FWGC problem is stated as follows:

> A farmer with his wolf, goat, and cabbage come to the edge of a river they wish to cross. There is a boat at the river's edge, but, of course, only the farmer can row. The boat also can carry only two things (including the rower) at a time. If the wolf is ever left alone with the goat, the wolf will eat the goat; similarly, if the goat is left alone with the cabbage, the goat will eat the cabbage. Devise a sequence of crossings of the river so that all four characters arrive safely on the other side of the river.

We now create a production system solution to this problem. First, we observe that the problem may be represented as a search through a graph. To do this we consider the possible moves that might be available at any time in the solution process. Some of these moves are eventually ruled out because they produce states that are unsafe (something will be eaten).

For the moment, suppose that all states are safe, and simply consider the graph of possible states. We often take this approach to problem solving,

relaxing various constraints so that we can see the general structure of the search problem. After we have described the full graph then it is often straightforward to add constraints that prohibit parts of the graph – the "illegal" states – from further exploration. The boat can be used in four ways: to carry the farmer and wolf, the farmer and goat, the farmer and cabbage, or the farmer alone. A state of the world is some combination of the characters on the two banks. Several states of the search are represented in Figure 4.2. States of the world may be represented using the predicate, `state(F, W, G, C)`, with the location of the farmer as first parameter, location of the wolf as second parameter, the goat as third, and the cabbage as fourth. We assume that the river runs "north to south" and that the characters are on either the east, `e`, or west, `w`, bank. Thus, `state(w, w, w, w)` has all characters on the west bank to start the problem.

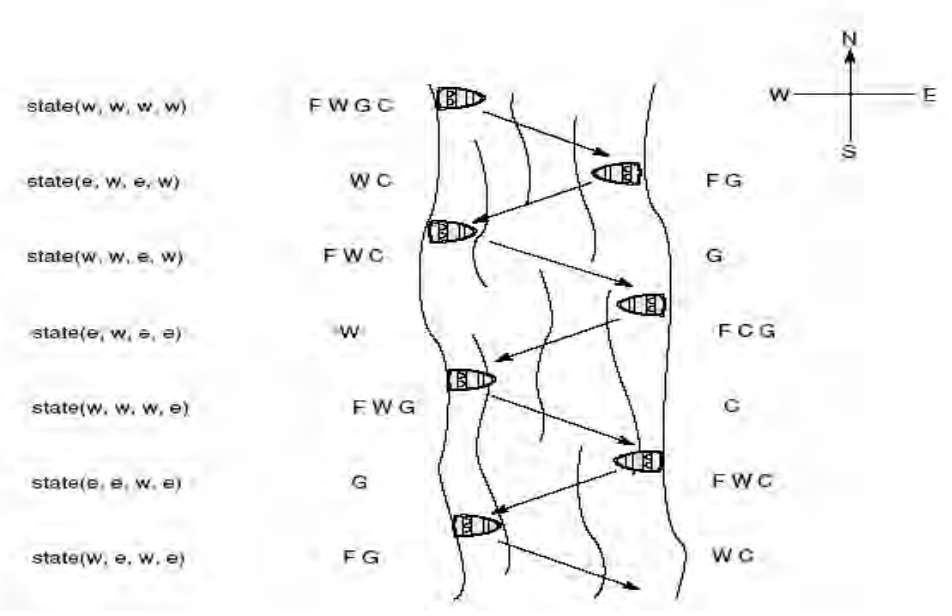

Figure 4.2. State representation and sample crossings of the F, W, G, C problem.

It must be pointed out that these choices are conventions that have been arbitrarily chosen by the authors. Indeed, as researchers in AI continually point out, the selection of an appropriate representation is often the most critical aspect of problem solving. These conventions are selected to fit the predicate calculus representation in Prolog. Different states of the world are created by different crossings of the river, represented by changes in the values of the parameters of the `state` predicate, as in Figure 4.2. Other representations are certainly possible.

We next describe a general graph for this river-crossing problem. For the time being, we ignore the fact that some states are unsafe. In Figure 4.3 we see the beginning of the graph of possible moves back and forth across the river. Since the farmer always rows, it is not necessary to have a separate representation for the location of the boat. Figure 4.3 represents part of the graph that is to be searched for a solution path.

The recursive path call described in Section 4.1 provides the control mechanism for the production system search. The production rules change state in the search. We define these *if*… *then*… rules in Prolog form. We take a direct approach here requiring that the pattern for the present state and the pattern for the next state both be placed in the head of the Horn clause, or to the left of :-. These are the arguments to the move predicate.

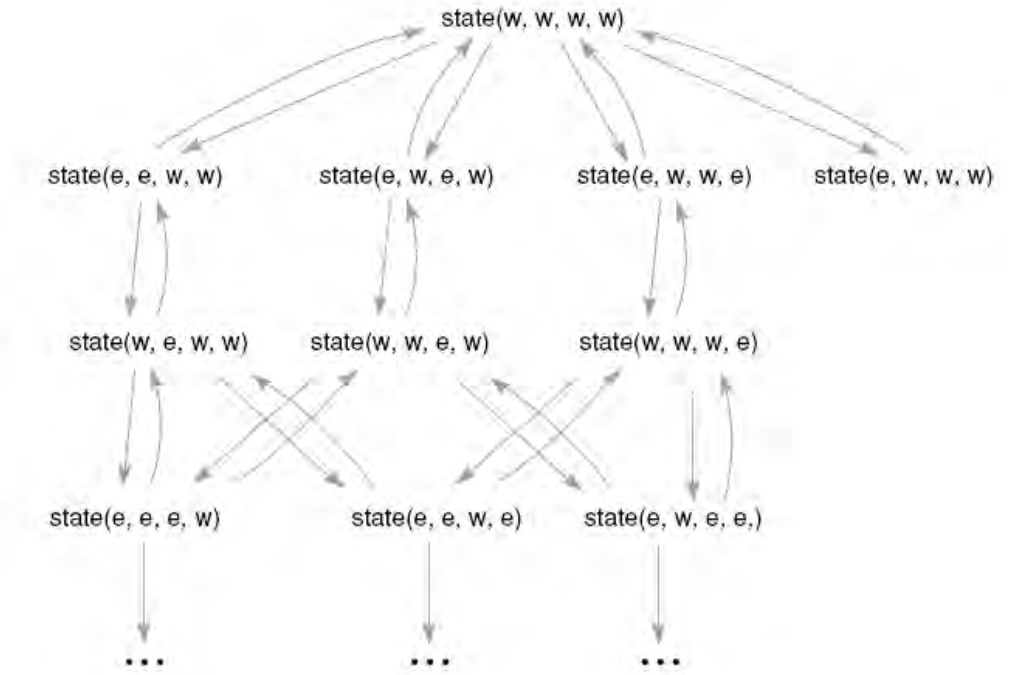

Figure 4.3. The beginning portion of the state space graph in the FWGC problem, including unsafe states.

The constraints that the production rule requires to fire and return the next state are placed to the right of :-. As shown in the following example, these conditions are expressed as unification constraints. The first rule is for the farmer to take the wolf across the river. This rule must account for both the transfer from east to west and the transfer from west to east, and it must not be applicable when the farmer and wolf are on opposite sides of the river. Thus, it must transform `state(e, e, G, C)` to `state(w, w, G, C)` and `state(w, w, G, C)` to `state(e, e, G, C)`. It must also fail for `state(e, w, G, C)` and `state(w, e, G, C)`. The variables `G` and `C` represent the fact that the third and fourth parameters can be bound to either `e` or `w`. Whatever their values, they remain the same after the move of the farmer and wolf to the other side of the river. Some of the states produced may indeed be "unsafe."

The following rule operates only when the farmer and wolf are in the same location and takes them to the opposite side of the river. Note that the goat and cabbage do not change their present location, whatever it might be.

```
move(state(X, X, G, C), state(Y, Y, G, C)) :-
     opp(X, Y).
```

```
opp(e, w).
opp(w, e).
```

This rule fires when a state (the present location in the graph) is presented to the first parameter of `move` in which the farmer and wolf are at the same location. When the rule fires, a new state, the second parameter of `move`, is produced with the value of `X` opposite, `opp`, the value of `Y`. Two conditions are satisfied to produce the new state: first, that the values of the first two parameters are the same and, second, that both of their new locations are opposite their old.

The first condition was checked implicitly in the unification process, in that `move` is not matched unless the first two parameters are the same. This test may be done explicitly by using the following rule:

```
move(state(F, W, G, C), state(Z, Z, G, C)) :-
    F = W, opp(F, Z).
```

This equivalent move rule first tests whether `F` and `W` are the same and, only if they are (on the same side of the river), assigns the opposite value of `F` to `Z`. Note that Prolog can do "assignment" by the binding of variable values in unification. Bindings are shared by all occurrences of a variable in a clause, and the scope of a variable is limited to the clause in which it occurs.

Pattern matching, a powerful tool in AI programming, is especially important in pruning search. States that do not fit the patterns in the rule are automatically pruned. In this sense, the first version of the `move` rule offers a more efficient representation because unification does not even consider the state predicate unless its first two parameters are identical.

Next, we create a predicate to test whether each new state is safe, so that nothing is eaten in the process of crossing the river. Again, unification plays an important role in this definition. Any state where the second and third parameters are the same and opposite the first parameter is `unsafe`: the wolf eats the goat. Alternatively, if the third and fourth parameters are the same and opposite the first parameter, the state is `unsafe`: the goat eats the cabbage. These `unsafe` situations may be represented with the following rules.

```
unsafe(state(X, Y, Y, C)) :- opp(X, Y).
unsafe(state(X, W, Y, Y)) :- opp(X, Y).
```

Several points should be mentioned. First, if a state is to be not unsafe (i.e., safe), according to the definition of `not` in Prolog, neither of these unsafe predicates can be true. Thus, neither of these predicates can unify with the current state or, if they do unify, their conditions are not satisfied. Second, `not` in Prolog is not exactly equivalent to the logical ~ of the first-order predicate calculus; `not` is rather "negation by failure of its opposite." The reader should test a number of states to verify that `unsafe` does what it is intended to do. Now, `not unsafe` is added to the previous production rule:

```
move(state(X, X, G, C), state(Y, Y, G, C)) :-
     opp(X, Y), not(unsafe(state(Y, Y, G, C))).
```

The `not unsafe` test calls `unsafe`, as mentioned above, to see whether the generated `state` is an acceptable new state in the search. When all criteria are met, including the check in the `path` algorithm that the new state is not a member of the visited-state list, `path` is (recursively) called on this state to go deeper into the graph. When `path` is called, the new state is added to the visited-state list.

In a similar fashion, we can create the three other production rules to represent the farmer taking the goat, cabbage, and himself across the river. We have added a `writelist` command to each production rule to print a trace of the current rule. The `reverse_print_stack` command is used in the terminating condition of `path` to print out the final solution path.

Finally, we add a fifth "pseudorule" that always fires, because no conditions are placed on it, when all previous rules have failed; it indicates that the `path` call is backtracking from the current state, and then that rule itself fails. This pseudorule is added to assist the user in seeing what is going on as the production system is running. We now present the full production system program in Prolog to solve the farmer, wolf, goat, and cabbage problem. The Prolog predicates `unsafe`, `writelist`, and the ADT stack predicates of Section 3.3.1, must also be included:

```
move(state(X, X, G, C), state(Y, Y, G, C)) :-
     opp(X, Y), not(unsafe(state(Y, Y, G, C))),
     writelist([`try farmer - wolf', Y, Y, G, C]).
move(state(X, W, X, C), state(Y, W, Y, C)) :-
     opp(X, Y), not(unsafe(state(Y, W, Y, C))),
     writelist([`try farmer - goat', Y, W, Y, C]).
move(state(X, W, G, X), state(Y, W, G, Y)) :-
     opp(X, Y), not(unsafe(state(Y, W, G, Y))),
     writelist([`try farmer - cabbage', Y, W, G, Y]).
move(state(X, W, G, C), state(Y, W, G, C)) :-
     opp(X, Y), not(unsafe(state(Y, W, G, C))),
     writelist([`try farmer by self', Y, W, G, C]).

move(state(F, W, G, C), state(F, W, G, C)) :-
     writelist([ `BACKTRACK at:', F, W, G, C]), fail.

path(Goal, Goal, Been_stack) :-
     write(`Solution Path Is: ' ), nl,
     reverse_print_stack(Been_stack).

path(State, Goal, Been_stack) :-
     move(State, Next_state),
     not(member_stack(Next_state, Been_stack)),
     stack(Next_state, Been_stack, New_been_stack),
     path(Next_state, Goal, New_been_stack), !.
opp(e, w).
opp(w, e).
```

The code is called by requesting `go`, which initializes the recursive `path` call. To make running the program easier, we can create a predicate, called `test`, that simplifies the input:

```
go(Start, Goal) :-
     empty_stack(Empty_been_stack),
     stack(Start, Empty_been_stack, Been_stack),
     path(Start, Goal, Been_stack).
test :- go(state(w,w,w,w), state(e,e,e,e)).
```

The algorithm backtracks from states that allow no further progress. You may also use `trace` to monitor the various variable bindings local to each call of `path`. It may also be noted that this program is a general program for moving the four creatures from any (legal) position on the banks to any other (legal) position, including asking for a path from the goal back to the start state. Other interesting features of production systems, including the fact that different orderings of the rules can produce different searches through the graph, are presented in the exercises. A partial trace of the execution of the F, W, G, C program, showing only rules actually used to generate new states, is presented next:

```
?- test.
try farmer takes goat e w e w
try farmer takes self w w e w
try farmer takes wolf e e e w
try farmer takes goat w e w w
try farmer takes cabbage e e w e
try farmer takes wolf w w w e
try farmer takes goat e w e e
    BACKTRACK from e,w,e,e
    BACKTRACK from w,w,w,e
try farmer takes self w e w e
try farmer takes goat e e e e
Solution Path Is:
state(w,w,w,w)
state(e,w,e,w)
state(w,w,e,w)
state(e,e,e,w)
state(w,e,w,w)
state(e,e,w,e)
state(w,e,w,e)
state(e,e,e,e)
```

In summary, this Prolog program implements a production system solution to the farmer, wolf, goat, and cabbage problem. The `move` rules make up the content of the production memory. The working memory is represented by the arguments of the `path` call. The production system control mechanism is defined by the recursive `path` call. Finally, the ordering of

rules for generation of children from each state (conflict resolution) is determined by the order in which the rules are placed in the production memory. We next present depth-, breadth-, and best-first search algorithms for production system based graph search.

4.3 Designing Alternative Search Strategies

As the previous subsection demonstrated, Prolog itself uses depth-first search with backtracking. We now show how alternative search strategies can be implemented in Prolog. Our implementations of depth-first, breadth-first, and best-first search use *open* and *closed* lists to record states in the search. The open list contains all potential next states in the search. How the open list is maintained, as a stack, as a queue, or as a priority queue, determines which particular state is next, that is, search is in either depth-first, breadth-first, or as best-first modes. The closed set keeps track of all the states that have been previously visited, and is used primarily to preventing looping in the graph as well as to keep track of the current path through the space. The details of how the open and closed data structures organize a search space can be found in Luger (2009, Chapter 3 and 4). When search fails at any point we do not backtrack. Instead, open and closed are updated within the `path` call and the search continues with these revised values. The cut is used to keep Prolog from storing the old versions of the open and closed lists.

Depth-first Search

Because the values of variables are restored when recursion backtracks, the list of visited states in the depth-first path algorithm of Section 4.2 records states only if they are on the current path to the goal. Although the testing each "new" state for membership in this list prevents loops, it still allows branches of the space to be reexamined if they are reached along paths generated earlier but abandoned at that time as unfruitful. A more efficient implementation keeps track of all the states that have ever been encountered. This more complete collection of states made up the members of the set we call *closed* (see Luger 2009, Chapter 3), and `Closed_set` in the following algorithm.

`Closed_set` holds all states on the current path plus the states that were rejected when the algorithm determined they had no usable children; thus, it no longer represents the path from the start to the current state. To capture this path information, we create the ordered pair `[State, Parent]` to keep track of each state and its parent; the `Start` state is represented by `[Start, nil]`. These state–parent pairs will be used to re-create the solution path from the `Closed_set`.

We now present a shell structure for depth-first search in Prolog, keeping track of both open and closed and checking each new state to be sure it was not previously visited. `path` has three arguments, the `Open_stack`, `Closed_set`, maintained as a set, and the `Goal` state. The current state, `State`, is the next state on the `Open_stack`. The stack and set operators are found in Section 3.3.

Search starts by calling a `go` predicate that initializes the `path` call. Note that `go` places the `Start` state with the `nil` parent, `[Start, nil]`, alone on `Open_stack`; the `Closed_set` is empty:

```
go(Start, Goal) :-
     empty_stack(Empty_open),
     stack([Start, nil], Empty_open, Open_stack),
     empty_set(Closed_set),
     path(Open_stack, Closed_set, Goal).
```

The three-argument `path` call is:

```
path(Open_stack, _, _) :-
     empty_stack(Open_stack),
     write('No solution found with these rules').
path(Open_stack, Closed_set, Goal) :-
     stack([State, Parent], _, Open_stack),
     State = Goal,
     write(`A Solution is Found!'), nl,
     printsolution([State, Parent], Closed_set).
path(Open_stack, Closed_set, Goal) :-
     stack([State, Parent], Rest_open_stack,
          Open_stack),
     get_children(State, Rest_open_stack, Closed_set,
          Children),
     add_list_to_stack(Children, Rest_open_stack,
          New_open_stack),
union([[State, Parent]], Closed_set,
          New_closed_set),
path(New_open_stack, New_closed_set, Goal), !.

get_children(State, Rest_open_stack, Closed_set,
          Children) :-
bagof(Child, moves(State, Rest_open_stack,
          Closed_set, Child), Children).

moves(State, Rest_open_stack, Closed_set, [Next,
          State]) :-
     move(State, Next),
     not(unsafe(Next)),     % test depends on problem
     not(member_stack([Next,_], Rest_open_stack)),
     not(member_set([Next,_], Closed_set)).
```

We assume a set of `move` rules appropriate to the problem, and, if necessary, an `unsafe` predicate:

```
move(Present_state, Next_state) :- …     % test rules
move(Present_state, Next_state) :- …
…
```

The first `path` call terminates search when the `Open_stack` is empty, which means there are no more states on the open list to continue the search. This usually indicates that the graph has been exhaustively searched. The second `path` call terminates and prints out the solution path when the solution is found. Since the states of the graph search are maintained as

`[State, Parent]` pairs, `printsolution` will go to the `Closed_set` and recursively rebuild the solution path. Note that the solution is printed from start to goal.

```
printsolution([State, nil], _) :- write(State), nl.
printsolution([State, Parent], Closed_set) :-
    member_set([Parent, Grandparent], Closed_set),
    printsolution([Parent, Grandparent], Closed_set),
    write(State), nl.
```

The third path call uses `bagof`, a Prolog built-in predicate standard to most interpreters. `bagof` lets us gather all the unifications of a pattern into a single list. The second parameter to `bagof` is the pattern predicate to be matched in the database. The first parameter specifies the components of the second parameter that we wish to collect. For example, we may be interested in the values bound to a single variable of a predicate. All bindings of the first parameter resulting from these matches are collected in a list, the *bag*, and bound to the third parameter.

In this program, `bagof` collects the states reached by firing *all* of the enabled production rules. Of course, this is necessary to gather all descendants of a particular state so that we can add them, in proper order, to open. The second argument of `bagof`, a new predicate named `moves`, calls the `move` predicates to generate all the states that may be reached using the production rules. The arguments to `moves` are the present state, the open list, the closed set, and a variable that is the state reached by a good move. Before returning this state, `moves` checks that the new state, `Next`, is not a member of either `rest_open_stack`, `open` once the present state is removed, or `closed_set`. `bagof` calls `moves` and collects all the states that meet these conditions. The third argument of `bagof` represents the new states that are to be placed on the `Open_stack`.

For some Prolog interpreters, `bagof` fails when no matches exist for the second argument and thus the third argument, `List`, is empty. This can be remedied by substituting `(bagof(X, moves(S, T, C, X), List); List = [ ])` for the current calls to `bagof` in the code.

Finally, because the states of the search are represented as state–parent pairs, member check predicates, e.g., `member_set`, must be revised to reflect the structure of pattern matching. We test if a state–parent pair is identical to the first element of the list of state–parent pairs and then recur if it isn't:

```
member_set([State, Parent], [[State, Parent]|_]).
member_set(X, [_|T]) :- member_set(X, T).
```

Breadth-first Search

We now present the *shell* of an algorithm for breadth-first search using explicit open and closed lists. This algorithm is called by:

```
go(Start, Goal) :-
    empty_queue(Empty_open_queue),
    enqueue([Start, nil], Empty_open_queue,
        Open_queue),
    empty_set(Closed_set),
    path(Open_queue, Closed_set, Goal).
```

`Start` and `Goal` have their obvious values. The shell can be used with the move rules and unsafe predicates for any search problem. Again we create the ordered pair `[State, Parent]`, as we did with depth-first search, to keep track of each state and its parent; the start state is represented by `[Start, nil]`. This will be used by `printsolution` to re-create the solution path from the `Closed_set`. The first parameter of `path` is the `Open_queue`, the second is the `Closed_set`, and the third is the `Goal`. *Don't care* variables, those whose values are not used in a clause, are written as "_".

```
path(Open_queue, _, _) :-
     empty_queue(Open_queue),
     write('Graph searched, no solution found.').

path(Open_queue, Closed_set, Goal) :-
     dequeue([State, Parent], Open_queue, _),
     State = Goal,
     write('Solution path is: '), nl,
     printsolution([State, Parent], Closed_set).

path(Open_queue, Closed_set, Goal) :-
     dequeue([State, Parent], Open_queue,
          Rest_open_queue),
get_children(State, Rest_open_queue,
          Closed_set, Children),
     add_list_to_queue(Children, Rest_open_queue,
          New_open_queue),
     union([[State, Parent]], Closed_set,
          New_closed_set),
     path(New_open_queue, New_closed_set, Goal), !.

get_children(State, Rest_open_queue, Closed_set,
          Children) :-
     bagof(Child, moves(State, Rest_open_queue,
          Closed_set, Child), Children).

moves(State, Rest_open_queue, Closed_set, [Next,
          State]) :-
     move(State, Next),
     not(unsafe(Next)),     %test depends on problem
     not(member_queue([Next,_], Rest_open_queue)),
     not(member_set([Next,_], Closed_set)).
```

This algorithm is a shell in that no `move` rules are given. These must be supplied to fit the specific problem domain, such as the FWGC problem of Section 4.2. The queue and set operators are found in Section 3.3.

The first `path` termination condition is defined for the case that `path` is called with its first argument, `Open_queue`, empty. This happens only when no more states in the graph remain to be searched and the solution has not been found. A solution is found in the second path predicate when the head of the `open_queue` and the `Goal` state are identical. When

`path` does not terminate, the third call, with `bagof` and `moves` predicates, gathers all the children of the current state and maintains the queue. (The detailed actions of these two predicates were described in Section 4.3.2.) In order to recreate the solution path, we saved each state as a state–parent pair, `[State, Parent]`. The start state has the parent `nil`. As noted in Section 4.3.1, the state–parent pair representation makes necessary a slightly more complex pattern matching in the `member`, `moves`, and `printsolution` predicates.

Best-first Search

Our shell for best-first search is a modification of the breadth-first algorithm in which the open queue is replaced by a priority queue, ordered by heuristic merit, which supplies the current state for each new call to `path`. In our algorithm, we attach a heuristic measure permanently to each new state on open and use this measure for ordering the states on open. We also retain the parent of each state. This information is used by `printsolution`, as in depth- and breadth-first search, to build the solution path once the goal is found.

To keep track of all required search information, each state is represented as a list of five elements: the state description, the parent of the state, an integer giving the depth in the graph of the state's discovery, an integer giving the heuristic measure of the state, and the integer sum of the third and fourth elements. The first and second elements are found in the usual way; the third is determined by adding one to the depth of its parent; the fourth is determined by the heuristic measure of the particular problem. The fifth element, used for ordering the states on the `open_pq`, is `f(n) = g(n) + h(n)`. A justification for using this approach to order states for heuristic search, usually referred to as the *A Algorithm*, is presented in Luger (2009, Chapter 4).

As before, the `move` rules are not specified; they are defined to fit the specific problem. The ADT operators for *set* and *priority queue* are presented in Section 3.3. `heuristic`, also specific to each problem, is a measure applied to each state to determine its heuristic weight, the value of the fourth parameter in its descriptive list.

This best-first search algorithm has two termination conditions and is called by:

```
go(Start, Goal) :-
     empty_set(Closed_set),
     empty_pq(Open),
     heuristic(Start, Goal, H),
     insert_pq([Start, nil, 0, H, H], Open, Open_pq),
     path(Open_pq, Closed_set, Goal).
```

`nil` is the parent of `Start` and `H` its heuristic evaluation. The code for best-first search is:

```
path(Open_pq, _,_) :-
     empty_pq(Open_pq),
     write('Graph searched, no solution found.').
```

```
path(Open_pq, Closed_set, Goal) :-
     dequeue_pq([State, Parent, _, _, _], Open_pq,_),
     State = Goal,
     write('The solution path is: '), nl,
     printsolution([State, Parent, _, _, _],
          Closed_set).
path(Open_pq, Closed_set, Goal) :-
     dequeue_pq([State, Parent, D, H, S], Open_pq,
          Rest_open_pq),
     get_children([State, Parent, D, H, S],
          Rest_open_pq, Closed_set, Children, Goal),
     insert_list_pq(Children, Rest_open_pq,
          New_open_pq),
     union([[State, Parent, D, H, S]], Closed_set,
          New_closed_set),
     path(New_open_pq, New_closed_set, Goal), !.
```

`get_children` is a predicate that generates all the children of `State`. It uses `bagof` and `moves` predicates as in the previous searches, with details earlier this Section. A set of `move` rules, a `safe` check for legal moves, and a `heuristic` must be specifically defined for each application. The `member` check must be specifically designed for five element lists.

```
get_children([State,_,D,_, _],Rest_open_pq,
          Closed_set,Children,Goal)  :-
     bagof(Child, moves([State, _, D, _, _],
          Rest_open_pq, Closed_set, Child,Goal),
          Children).
moves([State, _, Depth, _, _], Rest_open_pq,
          Closed_set,[Next,State,New_D,H,S], Goal) :-
     move(State, Next),
     not(unsafe(Next)),        % application specific
     not(member_pq([Next, _, _, _, _],Rest_open_pq)),
     not(member_set([Next, _, _, _, _],Closed_set)),
     New_D is Depth + 1,
     heuristic(Next, Goal, H), % application specific
     S is New_D + H.
```

`printsolution` prints the solution path, recursively finding state–parent pairs by matching the first two elements in the state description with the first two elements of the five element lists that make up the `Closed_set`.

```
printsolution([State, nil, _, _, _], _) :-
     write(State), nl.
printsolution([State, Parent, _, _, _], Closed_set):-
     member_set([Parent, Grandparent, _, _, _],
          Closed_set),
     printsolution([Parent, Grandparent, _, _, _],
          Closed_set),
     write(State), nl.
```

In Chapter 5 we further generalize the approach taken so far in that we present a set of built-in Prolog meta-predicates, predicates like `bagof`, that explicitly manipulate other Prolog predicates. This will set the stage for creating meta-interpreters in Chapter 6.

Exercises

1. Take the `path` algorithm presented for the knight's tour problem in the text. Rewrite the `path` call in the recursive code to the following form:

```
path(X, Y) :- path(X, W), move(W, Y).
```

Examine the trace of this execution and describe what is happening.

2. Write the Prolog code for the farmer, wolf, goat, and cabbage problem, Section 4.2:

A. Execute this code and draw a graph of the search space.
B. Alter the rule ordering to produce alternative solution paths.
C. Use the shell in the text to produce a breadth-first problem.
D. Describe a heuristic that might be appropriate for this problem.
E. Build the heuristic search solution.

3. Do A - E as in Exercise 2 to create a production system solution for the Missionary and Cannibal problem. Hint: you may want the `is` operator, see Section 5.3.

> Three missionaries and three cannibals come to the bank of a river they wish to cross. There is a boat that will hold only two, and any of the group is able to row. If there are ever more missionaries than cannibals on any side of the river the cannibals will get converted. Devise a series of moves to get all the people across the river with no conversions.

4. Use and extend your code to check alternative forms of the missionary and cannibal problem—for example, when there are four missionaries and four cannibals and the boat holds only two. What if the boat can hold three? Try to generalize solutions for the whole class of missionary and cannibal problems.

5. Write a production system Prolog program to solve the full 8 x 8 Knight's Tour problem. Do tasks A - E as described in Exercise 2.

6. Do A - E as in Exercise 2 above for the Water Jugs problem:

> There are two jugs, one holding 3 and the other 5 gallons of water. A number of things can be done with the jugs: they can be filled, emptied, and dumped one into the other either until the poured-into jug is full or until the poured-out-of jug is empty. Devise a sequence of actions that will produce 4 gallons of water in the larger jug. (Hint: use only integers.)

5 Meta-Linguistic Abstraction, Types, and Meta-Interpreters

Chapter Objectives

A number of Prolog meta-predicates are presented, including:

- `Atom`
- `clause`
- `univ` (`=..`)
- `call`

The type system for Prolog:

- Programmer implements typing as needed
- Types as run time constraints rather than enforced at compile time

Unification and variable binding explained

Evaluation versus unification

- `is` versus `=`
- Difference lists demonstrated

Chapter Contents

5.1 Meta-Interpreters, Types, and Unification

Meta-Logical Predicates

In this chapter we first consider a set of powerful Prolog predicates, called *meta-predicates*. These predicates take as their scope other predicates in the Prolog environment. Thus they offer tools for building *meta-interpreters*, interpreters in a language that are able to interpret specifications in that language. An example will be to build a rule interpreter in Prolog, an interpreter that can manipulate and interpret rule sets, specified in Prolog syntax. These interpreters can also be used to query the user, offer explanations of the interpreter's decisions, implement multi-valued or fuzzy logics, and run any Prolog code.

In Section 5.1 we introduce a useful set of meta-predicates. In Section 5.2 we discuss data typing for Prolog and describe how type constraints can be added to a prolog system. An example of a typed relational database in Prolog is given. Finally, in Section 5.3, we discuss unification and demonstrate with difference lists how powerful this can be.

Meta-logical constructs extend the expressive power of any programming environment. We refer to these predicates as *meta* because they are designed to match, query, and manipulate other predicates that make up the specifications of the problem domain. That is, they can be used to reason about Prolog predicates rather than the terms or objects these other predicates denote. We need meta-predicates in Prolog for (at least) five reasons:

To determine the "type" of an expression;
To add "type" constraints to logic programming applications;
To build, take apart, and evaluate Prolog structures;
To compare values of expressions;
To convert predicates passed as data to executable code.

We have actually seen a number of meta-predicates already. In Chapter 2 we described how global structures, which are those that can be accessed by the entire clause set, are entered into a Prolog program. The command `assert(C)` adds the clause `C` to the current set of clauses. There are dangers associated with programming with predicates such as `assert` and `retract`. Because these predicates are able to create and remove global structures, they can introduce side effects into the program, and may cause other problems associated with poorly structured programs. Yet, it is sometimes necessary to use global structures to draw on the power of Prolog's built-in database and pattern matching. We do this when creating *semantic nets* and *frames* in a Prolog environment, as in Section 2.4. We may also use global structures to describe new results as they are found with a rule-based expert system shell, as in Section 6.2. We want this information to be global so that other predicates (rules) may access it when appropriate.

Other meta-predicates that are useful for manipulating representations include:

`var(X)` succeeds only when `X` is an unbound variable.

`nonvar(X)` succeeds only when `X` is bound to a nonvariable term.

`=..` creates a list from a predicate term.

For example, `foo(a, b, c) =.. Y` unifies `Y` with `[foo, a, b, c]`. The head of the list `Y` is the predicate name, and its tail is the predicate's arguments. `=..` also can be used to bind alternative variable patterns, of course. Thus, if `X =.. [foo, a, b, c]` succeeds, then `X` has the value `foo(a, b, c)`.

`functor(A, B, C)` succeeds with `A` a term whose principal `functor` has name `B` and arity `C`.

For example, `functor(foo(a, b), X, Y)` will succeed with variables `X` = `foo` and `Y` = 2. `functor(A, B, C)` can also be used with any of its arguments bound in order to produce the others, such as all the terms with a certain name and/or arity.

`clause(A, B)` unifies `B` with the body of a clause whose head is A.

For example, if `p(X) :- q(X)` exists in the database, then `clause(p(a), Y)` will succeed with `Y` = `q(a)`. This is useful for controlling rule chaining in an interpreter, as seen in Chapter 6.

`any_predicate(…, X, …) :- X` executes predicate `X`, the argument of any predicate.

Thus a predicate, here `X`, may be passed as a parameter and executed at any desired time. `call(X)`, where `X` is a clause, also succeeds with the execution of predicate `X`.

This short list of meta-logical predicates will be very important in building and interpreting AI data structures. Because Prolog can manipulate its own structures in a straightforward fashion, it is easy to implement interpreters that modify the Prolog semantics, as we see next.

5.2 Types in Prolog

For a number of problem-solving applications, the unconstrained use of unification can introduce unintended error. Prolog is an untyped language, in that unification simply matches patterns, without restricting them according to data type. For example, `append(nil, 6, 6)` can be inferred from the definition of `append`, as we will see in Chapter 10. Strongly typed languages such as Pascal have shown how type checking helps programmers avoid these problems. Researchers have proposed adding types to Prolog (Neves et al. 1986, Mycroft and O'Keefe 1984).

Typed data are particularly appropriate in a relational database (Neves et al. 1986, Malpas 1987). The rules of logic can be used as constraints on the data and the data can be typed to enforce consistent and meaningful interpretation of the queries. Suppose that a department store database has `inventory`, `suppliers`, `supplier_inventory`, and other appropriate relations. We define a database as relations with named fields that can be thought of as sets of tuples. For example, `inventory` might consist of 4-tuples, where:

```
< Pname, Pnumber, Supplier, Weight >  inventory
```

only when `Supplier` is the supplier name of an `inventory` item numbered `Pnumber` that is called `Pname` and has weight `Weight`. Suppose further:

```
< Supplier, Snumber, Status, Location >  suppliers
```

only when `Supplier` is the name of a supplier numbered `Snumber` who has status `Status` and lives in city `Location`. Suppose finally:

```
< Supplier, Pnumber, Cost, Department >
     supplier_inventory
```

only if `Supplier` is the name of a supplier of part number `Pnumber` in the amount of `Cost` to department `Department`.

We may define Prolog rules that implement various queries and perform type checking across these relationships. For instance, the query "are there suppliers of part number 1 that live in London?" is given in Prolog as:

```
?- getsuppliers(Supplier,1, london).
```

The rule:

```
getsuppliers(Supplier, Pnumber, City) :-
     cktype(City, suppliers, city),
     suppliers(Supplier, _, _,City),
     cktype(Pnumber, inventory, number),
     supplier_inventory(Supplier, Pnumber, _, _),
     cktype(Supplier, inventory, name).
```

implements this query and also enforces the appropriate constraints across the tuples of the database. First the variables `Pnumber` and `City` are bound when the query unifies with the head of the rule; the predicate `cktype` tests that `Supplier` is an element of the set of suppliers, that `1` is a legitimate inventory number, and that `london` is a suppliers' city.

We define `cktype` to take three arguments: a value, a relation name, and a field name, and to check that each value is of the appropriate type for that relation. For example, we may define lists of legal values for `Supplier`, `Pnumber`, and `City` and enforce data typing by requiring member checks of candidate values across these lists. Alternatively, we may define logical constraints on possible values of a type; for example, we may require that inventory numbers be less than 1000.

We should note the differences in type checking between standard languages such as Pascal and Prolog. We might define a Pascal data type for suppliers as:

```
type supplier = record
     sname: string;
     snumber: integer;
     status: boolean;
     location: string
     end
```

The Pascal programmer defines new types, here `supplier`, in terms of already defined types, such as `boolean` or `integer`. When the programmer uses variables of this type, the compiler automatically enforces type constraints on their values.

In Prolog, we can represent the supplier relation as instances of the form:

```
supplier(sname(Supplier),
     snumber(Snumber),
     status(Status),
     location(Location)).
```

We then implement type checking by using rules such as `getsuppliers` and `cktype`. The distinction between Pascal and Prolog type checking is clear and important: the Pascal type declaration tells the compiler the form for both the entire structure (record) and the individual components (`boolean`, `integer`, `string`) of the data type. In Pascal we declare variables to be of a particular type (`record`) and then create procedures to access these typed structures.

```
procedure changestatus (X: supplier);
     begin
          if X.status then. …
```

Because it is nonprocedural, Prolog does not separate the declaration from the use of data types, and type checking is done as the program is executing. Consider the rule:

```
supplier_name(supplier(sname(Supplier),
                       snumber(Snumber),
                       status(true),
                       location (london))) :-
     integer(Snumber), write(Supplier).
```

`supplier_name` takes as argument an instance of the `supplier` predicate and writes the name of the `Supplier`. However, this rule will succeed only if the supplier's number is an integer, the status is active (`true`), and the supplier lives in `london`. An important part of this type check is handled by the unification algorithm (`status` and `location`) and the rest is the built-in system-predicate `integer`. Further constraints could restrict values to be from a particular list; for example, `Snumber` could be constrained to be from a list of supplier numbers. We define constraints on database queries using rules such as `cktype` and `supplier_name` to implement type checking when the program is executed.

So far, we have seen three ways that data may be typed in Prolog. First, and most powerful, is the program designer's use of unification and syntactic patterns to constrain variable assignment. Second, Prolog itself provides predicates to do limited type checking. We saw this with meta-predicates such as `var(X)`, `clause(X,Y)`, and `integer(X)`. The third use of typing occurred in the inventory example where rules checked lists of legitimate `Supplier`, `Pnumbers`, and `Cities` to enforce type constraints.

A fourth, and more radical approach is the complete predicate and data type check proposed by Mycroft and O'Keefe (1984). Here all predicate names are typed and given a fixed arity. Furthermore, all variable names are themselves typed. A strength of this approach is that the constraints on the constituent predicates and variables of the Prolog program are themselves enforced by a (meta) Prolog program. Even though the result may be slower program execution, the security gained through total type enforcement may justify this cost.

To summarize, rather than providing built-in type checking as a default, Prolog allows run-time type checking under complete programmer control. This approach offers a number of benefits for AI programmers, including the following:

1. The programmer is not forced to adhere to strong type checking at all times. This allows us to write predicates that work across any type of object. For example, the `member` predicate performs general member checking, regardless of the type of elements in the list.
2. User flexibility with typing helps exploratory programming. Programmers can relax type checking in the early stages of program development and introduce it to detect errors as they come to better understand the problem.
3. AI representations seldom conform to the built-in data types of languages such as Pascal, C++, or Java. Prolog allows

types to be defined using the full power of predicate calculus. The database example showed this flexibility.

4. Because type checking is done at run time rather than compile time, the programmer determines when the program should perform a check. This allows programmers to delay type checking until it is necessary or until certain variables have become bound.
5. Programmer control of type checking at run time also supports the creation of programs that build and enforce new types during execution. This can be of use in a learning or a natural language processing program, as we see in Chapters 7, 8, and 9.

In the next section we take a closer look at unification in Prolog. As we noted earlier, unification is the technical name for pattern matching, especially when applied to expressions in the Predicate Calculus. The details for implementing this algorithm may be found in Luger (2009, Section 2.3). In Prolog, unification is implemented with backtracking that supports full systematic instantiation of all values defined for the problem domain. To master the art of Prolog programming the sequential actions of the interpreter, sometimes referred to as Prolog's "procedural semantics" must be fully understood.

5.3 Unification, Engine of Variable Binding and Evaluation

An important feature of Prolog programming is the interpreter's behavior when considering a problem's specification and faced with a particular query. The query is matched with the set of specifications to see under what constraints it might be true. The interpreter's action, left-to-right depth first backtracking across all specified variable bindings, is a variation of the search of a resolution-based reasoning system.

But Prolog is NOT a full mathematically sound theorem prover, as it lacks several important constraints, including the occurs check, and Prolog also supports the use of cut. For details see Luger 2009, Section 14.3). The critical point is that Prolog performs a systematic search across database entries, rather than, as in traditional languages, a sequential evaluation of statements and expressions. This has an important result: variables are bound (assigned values or instantiated) by *unification* and not by an evaluation process, unless, of course, an evaluation is explicitly requested. This paradigm for programming has several implications.

The first and perhaps most important result is the relaxation of the requirement to specify variables as input or output. We saw this power briefly with the `member` predicate in Chapter 2 and will see it again with the `append` predicate in Chapter 10. `append` can either join lists together, test whether two lists are correctly appended, or break a list into parts consistent with the definition of `append`. We use unification as a constraint handler for parsing and generating natural language sentences in Chapters 7 and 8.

Unification is also a powerful technique for rule-based and frame-based

expert systems. All production systems require a form of this matching, and it is often necessary to write a unification algorithm in languages that don't provide it, see, for example, Section 15.1 for a Lisp implementation of unification.

An important difference between unification-based computing and the use of more traditional languages is that unification performs syntactic matches (with appropriate parameter substitutions) on structures. It does *not* evaluate expressions. Suppose, for example, we wished to create a `successor` predicate that succeeds if its second argument is the arithmetic successor of its first argument. Not understanding the unification/evaluation paradigm, we might be tempted to define `successor`:

```
successor (X, Y) :- Y = X + 1.
```

This will fail because the `=` operator does not evaluate its arguments but only attempts to unify the expressions on either side. This predicate succeeds if `Y` unifies with the structure `X + 1`. Because 4 does not unify with 3 + 1, the call `successor(3, 4)` fails! On the other hand, demonstrating the power of unification, `=` can test for equivalence, as defined by determining whether substitutions exist that can make *any* two expressions equivalent. For example, whether:

```
friends (X, Y) = friends(george, kate).
```

In order to correctly define `successor` (and other related arithmetic predicates), we need to be able to evaluate arithmetic expressions. Prolog provides an operator, `is`, for just this task. `is` evaluates the expression on its right-hand side and attempts to unify the result with the object on its left. Thus:

```
X is Y + Z.
```

unifies `X` with the value of `Y` added to `Z`. Because it performs arithmetic evaluation, if `Y` and `Z` do not have values (are not bound at execution time), the evaluation of `is` causes a run-time error. Thus, `X is Y + Z` cannot (as one might think with a declarative programming language) give a value to `Y` when `X` and `Z` are bound. Therefore programs must use `is` to evaluate expressions with arithmetic operators, `+`, `–`, `*`, `/`, and `mod`.

Finally, as in the predicate calculus, variables in Prolog may have one and only one binding within the scope of a single expression. Once given a value, through local assignment or unification, variables can never take on a new value, except through backtracking in the and/or search space of the current interpretation. Upon backtracking, all the instances of the variable within the scope of the expression take on the new value. Thus, `is` cannot function as a traditional assignment operator; and expressions such as `X is X + 1` will always fail.

Using `is`, we now properly define `successor(X, Y)` where the second argument has a numeric value that is one more than the first:

```
successor (X, Y) :- Y is X + 1.
```

`successor` will now have the correct behavior as long as `X` is bound to a numeric value at the time that the `successor` predicate is called.

`successor` can be used either to compute `Y`, given `X`, or to test values assigned to `X` and `Y`:

```
?- successor (3, X).
X = 4
Yes
?- successor (3, 4).
Yes
?- successor (4, 2).
No
?- successor (Y, 4).
failure, error in arithmetic expression
```

since `Y` is not bound at the time that `successor` is called.

As this discussion illustrates, Prolog does not evaluate expressions as a default as in traditional languages such as C++ and Java. The programmer must explicitly call for evaluation and assignment using `is`. Explicit control of evaluation, as also found in Lisp, makes it easy to treat expressions as data, passed as parameters, and creating or modifying them as needed within the program. This feature, like the ability to manipulate predicate calculus expressions as data and execute them using `call`, greatly simplifies the development of different interpreters, such as the expert system shell of the next chapter.

We close this discussion of the power of unification-based computing with an example that does string catenation through the use of *difference lists*. As an alternative to the standard Prolog list notation, we can represent a list as the difference of two lists. For example, `[a, b]` is equivalent to `[a, b | [ ] ] – [ ]` or `[a, b, c] – [c]`. This representation has certain expressive advantages over the traditional list syntax. When the list `[a, b]` is represented as the difference `[a, b | Y] – Y`, it actually describes the potentially infinite class of all lists that have `a` and `b` as their first two elements. Now this representation has an interesting property, namely addition:

```
X – Z = X – Y + Y – Z
```

We can use this property to define the following single-clause logic program where `X – Y` is the first list, `Y – Z` is the second list, and `X – Z` is the result of catenating them, as in Figure 5.1: We create the predicate `catenate` that takes two list `X` and `Y` and creates `Z`:

```
catenate(X – Y, Y – Z, X – Z).
```

This operation joins two lists of any length in constant time by unification on the list structures, rather than by repeated assignment based on the length of the lists (as with `append`, Chapter 10). Thus, the `catenate` call gives:

```
?- catenate ([a, b  Y] – Y, [1, 2, 3] – [ ], W).
Y = [1, 2, 3]
W = [a, b, 1, 2, 3] – [ ]
```

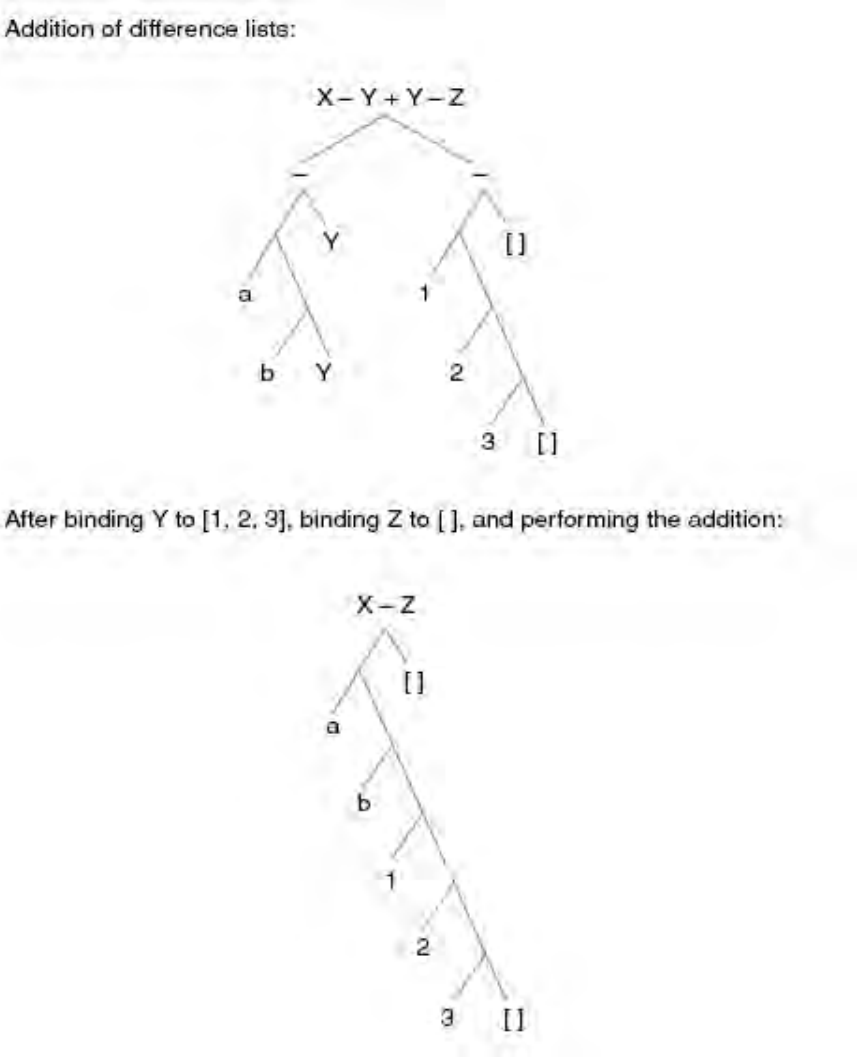

Figure 5.1 Tree diagrams: list catenation using difference lists.

As may be seen in Figure 5.1, the (subtree) value of `Y` in the second parameter is unified with both occurrences of `Y` in the first parameter of `catenate`. This demonstrates the power of unification, not simply for substituting values for variables but also for matching general structures: all occurrences of `Y` take the value of the entire subtree. The example also illustrates the advantages of an appropriate representation. Thus *difference lists* represent a whole class of lists, including the desired catenation.

In this section we have discussed a number of idiosyncrasies and advantages of Prolog's unification-based approach to computing. Unification is at the heart of Prolog's declarative semantics. For a more complete discussion of Prolog's semantics see Luger (2009, Section 14.3).

In Chapter 6 we use Prolog's declarative semantics and unification-based pattern matching to design three meta-interpreters: Prolog in Prolog, the shell for an expert system, and a planner.

Exercises

1. Create a type check that prevents the member check predicate (that checks whether an item is a member of a list of items) from crashing when called on `member(a, a)`. Will this "fix" address the `append(nil, 6, 6)` anomaly that is described in Chapter 9? Test it and see.

2. Create the "inventory supply" database of Section 5.2. Build type checks for a set of six useful queries on these data tuples.

3. Is the difference list `catenate` really a linear time `append` (Chapter 10)? Explain.

4. Explore the powers of unification. Use `trace` to see what happens when you query the Prolog interpreter as follows. Can you explain what is happening?

a: `X = X + 1`

b: `X is X + 1`

c: `X = foo(X)`

6 Three Meta-Interpreters: Prolog in Prolog, EXSHELL, and a Planner

Chapter Objectives

Prolog's meta-predicates used to build three meta-interpreters
- Prolog in Prolog
- An expert system shell: `exshell`
- A planner in Prolog

The Prolog in Prolog interpreter:
- Left-to-right and depth-first search
- Solves for a goal look first for facts, then rules, then ask user

`exshell` performed, using a set of `solve` predicates:
- Goal-driven, depth-first search
- Answers `how` (rule stack) and `why` (proof tree)
- Pruned search paths using the Stanford certainty factor algebra

The Prolog planner
- Uses an add and delete list to generate new states
- Performs depth-first and left-to-right search for a plan

Chapter Contents

6.1 An Introduction to Meta-Interpreters: Prolog in Prolog

Meta Interpreters

In both Lisp and Prolog, it is easy to write programs that manipulate expressions written in that language's syntax. We call such programs *meta-interpreters*. In an example we will explore throughout this book, an *expert system shell* interprets a set of rules and facts that describe a particular problem. Although the rules of a problem situation are written in the syntax of the underlying language, the meta-interpreter redefines their semantics. The "tools" for supporting the design of a meta-interpreter in Prolog were the *meta predicates* presented in Chapter 5.

In this chapter we present three examples of meta-interpreters. As our first example, we define the semantics of pure Prolog using the Prolog language itself. This is not only an elegant statement of Prolog semantics, but also will serve as a starting point for more complex meta-interpreters. `solve` takes as its argument a Prolog goal and processes it according to the semantics of Prolog:

```
solve(true) :-!.
solve(not A) :- not(solve(A)).
solve((A, B)) :-!, solve(A), solve(B).
solve(A) :- clause(A, B), solve(B).
```

The first `solve` predicate checks to see if its argument is a fact and true. The second checks to determine whether the argument of `solve` is false and makes the appropriate change. The third `solve` predicate sees if the argument of the `solve` predicate is the `and` of two predicates and then calls `solve` on the first followed by calling `solve` on the second. Actually, the third `solve` can handle any number of `and`ed goals, calling `solve` on the first and then calling `solve` on the set of `and`ed remaining goals. Finally, when the three previous attempts have failed, `solve`, using the `clause` metapredicate, finds a rule whose head is the goal and then calls `solve` on the body of that rule. `solve` implements the same left-to-right, depth-first, goal-directed search as the built-in Prolog interpreter.

If we assume the following simple set of assertions,

```
p(X, Y) :- q(X), r(Y).
q(X) :- s(X).
r(X) :- t(X).
s(a).
t(b).
t(c).
```

`solve` has the behavior we would expect of Prolog:

```
?- solve(p(a, b)).
Yes
?- solve(p(X, Y)).
X = a, Y = b;
X = a, Y = c;
No
?- solve(p(f, g)).
no
```

The ability easily to write meta-interpreters for a language has certain theoretical advantages. For example, McCarthy wrote a simple Lisp meta-interpreter as part of a proof that the Lisp language is Turing complete (McCarthy 1960). From a more practical standpoint, we can use meta-interpreters to extend or modify the semantics of the underlying language to better fit our application. This is the programming methodology of *meta-linguistic abstraction*, the creation of a high-level language that is designed to help solve a specific problem.

For example, we can extend the standard Prolog semantics so as to ask the user about the truth-value of any goal that does not succeed (using the four `solve` predicates above) in the knowledge base. We do this by adding the following clauses to the end of the previous definitions of `solve`:

```
solve(A) :- askuser(A).
askuser(A) :- write(A),
     write('? Enter true if the goal is true, false
          otherwise'), nl.
     read(true).
```

Because we add this definition to the end of the other `solve` rules, it is called only if all of these earlier `solve` rules fail. `solve` then calls `askuser` to query the user for the truth value of the goal (`A`). `askuser` prints the goal and instructions for answering. `read(true)` attempts to unify the user's input with the term `true`, failing if the user enters `false` (or anything that does not unify with `true`). In this way we have changed the semantics of `solve` and extended the behavior of Prolog. An example, using the simple knowledge base defined above, illustrates the behavior of the augmented definition of `solve`:

```
?- solve(p(f, g)).
s(f)? Enter true if the goal is true, false
otherwise
true.
t(g)? Enter true if the goal is true, false
otherwise
true.
yes
```

Another extension to the meta-interpreter allows it to respond to "why" queries. When the interpreter asks the user a question, the user can respond with `why`; the appropriate response to this query is the current rule that the program is trying to solve. We implement this by storing the stack of rules in the current line of reasoning as the second parameter to `solve`. Whenever `solve` calls `clause` to solve a goal, it places the selected rule on the stack. Thus, the rule stack records the chain of rules from the top-level goal to the current subgoal.

Because the user may now enter two valid responses to a query, `askuser` calls `respond`, which either succeeds if the user enters `true` (as before) or prints the top rule on the stack if the user enters `why`. `respond` and `askuser` are mutually recursive, so that after printing the answer to a `why` query, `respond` calls `askuser` to query the user about the goal again. Note, however, that `respond` calls `askuser` with the tail of the rule stack. Thus, a series of `why` queries will simply chain back up the rule stack until the stack is empty – the search is at the root node of the tree – letting the user trace the entire line of reasoning.

```
solve(true, _) :-!.
solve(not(A), Rules) :- not(solve(A, Rules)).
solve((A, B), Rules) :- !,
     solve(A, Rules), solve(B, Rules).
solve(A, Rules) :-
     clause(A, B), solve(B, [(A :- B) | Rules]).
solve(A, Rules) :- askuser(A, Rules).
askuser(A, Rules) :-
     write(A),
     write('? Enter true if goal is true,
          false otherwise'),nl,
     read(Answer), respond(Answer, A, Rules).
```

```
respond(true, _, _).
respond(why, A, [Rule | Rules]) :-
     write(Rule), nl,
     askuser(A, Rules).
respond(why, A, [ ]) :- askuser(A, [ ]).
```

For example, suppose we run `solve` on the simple database introduced earlier in the section:

```
?- solve(p(f, g), [ ]).
s(f)? Enter true if goal is true, false otherwise
why.
q(f) :- s(f)
s(f)? Enter true if goal is true, false otherwise
why.
p(f,g) :- (q(f), r(g))
s(f)? Enter true if goal is true, false otherwise
true.
t(g)? Enter true if goal is true, false otherwise
true.
yes
```

Note how successive `why` queries actually trace back up the full line of reasoning.

A further useful extension to the `solve` predicate constructs a proof tree for any successful goal. The ability to build proof trees provides expert system shells with the means of responding to "how" queries; it is also important to any algorithm, such as explanation-based learning (Chapter 7), that reasons about the results of a problem solver.

We may modify the pure Prolog interpreter to build a proof tree recursively for a goal as it solves that goal. In the definition that follows, the proof is returned as the second parameter of the `solve` predicate. The proof of the atom `true` is that atom; this halts the recursion. In solving a goal `A` using a rule `A :- B`, we construct the proof of `B` and return the structure (`A :- ProofB`). In solving a conjunction of two goals, `A` and `B`, we simply conjoin the proof trees for each goal: (`ProofA, ProofB`).

The definition of a meta-interpreter that supports the construction of the proof trees is:

```
solve(true, true) :-!.
solve(not(A), not ProofA) :-
     not(solve(A, ProofA)).
solve((A, B),(ProofA, ProofB)) :-
     solve(A, ProofA), solve(B, ProofB).
solve(A, (A :- ProofB)) :-
     clause(A, B), solve(B, ProofB).
solve(A, (A :- given)) :-
askuser(A).
```

```
askuser(A, Proof) :-
     write(A),
     write('enter true if goal is true,
          false otherwise'),
      read(true).
```

Running this on our simple database gives the results:

```
?- solve(p(a, b), Proof).
Proof = p(a, b) :-
       ((q(a) :-
            (s(a) :-
                 true)),
       r(b) :-
            (t(b) :-
                 true)))
```

In the next section, we use these same techniques to implement an expert system shell. `exshell` uses a knowledge base in the form of rules to solve problems. It asks the user for needed information, keeps a record of case-specific data, responds to `how` and `why` queries, and implements the Stanford certainty factor algebra (Luger 2009, Section 9.2.1). Although this program, `exshell`, is much more complex than the Prolog meta-interpreters discussed above, it is just an extension of this methodology. Its heart is a `solve` predicate implementing a back-chaining search.

6.2 A Shell for a Rule-Based Expert System

EXSHELL

In this section we present the key predicates used in the design of an interpreter for a goal-driven, rule-based expert system. At the end of this section, we demonstrate the performance of `exshell` using an automotive diagnostic knowledge base. If the reader would prefer to read through this trace before examining `exshell`'s key predicates, we encourage looking ahead.

An `exshell` knowledge base consists of rules and specifications of queries that can be made to the user. Rules are represented using a two-parameter `rule` predicate of the form `rule(R, CF)`. The first parameter is an assertion to the knowledge base, written using standard Prolog syntax. Assertions may be Prolog rules, of the form (`G :- P`), where `G` is the head of the rule and `P` is the conjunctive pattern under which `G` is true. The first argument to the rule predicate may also be a Prolog fact. `CF` is the confidence the designer has in the rule's conclusions. `exshell` implements the certainty factor algebra of MYCIN, (Luger 2009, Section 9.2.1), and we include a brief overview of the Stanford algebra here. Certainty factors (`CF`s) range from +100, a fact that is true, to –100, something that is known to be false. If the `CF` is around 0, the truth value is unknown. Typical rules from a knowledge base for diagnosing automotive failures are:

```
rule((bad_component(starter) :-
     (bad_system(starter_system),
          lights(come_on))), 50).
rule(fix(starter, 'replace starter'),100).
```

Thts first rule states that if the bad system is shown to be the starter system and the lights come on, then conclude that the bad component is the starter, with a certainty of 50. Because this rule contains the symbol `:-` it must be surrounded by parentheses. The second rule asserts the fact that we may fix a broken starter by replacing it, with a certainty factor of 100. `exshell` uses the `rule` predicate to retrieve those rules that conclude about a given goal, just as the simpler versions of `solve` in Section 6.1 used the built-in `clause` predicate to retrieve rules from the global Prolog database.

`exshell` supports user queries for unknown data. However, because we do not want the interpreter to ask for every unsolved goal, we allow the programmer to specify exactly what information may be obtained from asking. We do this with the `askable` predicate:

```
askable(car_starts).
```

Here `askable` specifies that the interpreter may ask the user for the truth of the `car_starts` goal when nothing is known or can be concluded about that goal.

In addition to the programmer-defined knowledge base of rules and askables, `exshell` maintains its own record of case-specific data. Because the shell asks the user for information, it needs to remember what it has been told; this prevents the program from asking the same question twice during a consultation (decidedly non-expert behavior!).

The heart of the `exshell` meta-interpreter is a predicate of four arguments called, surprisingly, `solve`. The first of these arguments is the goal to be solved. On successfully solving the goal, `exshell` binds the second argument to the (accumulated) confidence in the goal as computed from the knowledge base. The third argument is the rule stack, used in responding to `why` queries, and the fourth is the cutoff threshold for the certainty factor algebra. This allows pruning of the search space if the confidence falls below a specified threshold.

In attempting to satisfy a goal, `G`, `solve` first tries to match `G` with any facts that it already has obtained from the user. We represent known facts using the two-parameter `known(A, CF)` predicate. For example, `known(car_starts, 85)` indicates that the user has already told us that the car starts, with a confidence of `85`. If the goal is unknown, `solve` attempts to solve the goal using its knowledge base. It handles the negation of a goal by solving the goal and multiplying the confidence in that goal by –1. It solves conjunctive goals in left-to-right order. If `G` is a positive literal, `solve` tries any rule whose head matches `G`. If this fails, solve queries the user. On obtaining the user's confidence in a goal, `solve` asserts this information to the database using a `known` predicate.

```
% Case 1: truth value of goal is already known
solve(Goal, CF, _, Threshold) :
     known(Goal, CF), !,
     above_threshold(CF, Threshold).
```

```
% Case 2: negated goal
solve(not(Goal), CF, Rules, Threshold) :-!,
     invert_threshold(Threshold, New_threshold),
     solve(Goal, CF_goal, Rules, New_threshold),
     negate_cf(CF_goal, CF).
% Case 3: conjunctive goals
solve((Goal_1,Goal_2), CF, Rules, Threshold) :- !,
     solve(Goal_1, CF_1, Rules, Threshold),
     above_threshold(CF_1, Threshold),
     solve(Goal_2, CF_2, Rules, Threshold),
     above_threshold(CF_2, Threshold),
     and_cf(CF_1, CF_2, CF).
% Case 4: back chain on a rule in knowledge base
solve(Goal, CF, Rules, Threshold) :-
     rule((Goal :- (Premise)), CF_rule),
     solve(Premise, CF_premise, [rule((Goal :-
            Premise), CF_rule)|Rules], Threshold),
     rule_cf(CF_rule, CF_premise, CF),
     above_threshold(CF, Threshold).
% Case 5: fact assertion in knowledge base
solve(Goal, CF, _, Threshold) :-
     rule(Goal, CF),
     above_threshold(CF, Threshold).
% Case 6: ask user
solve(Goal, CF, Rules, Threshold) :-
     askable(Goal),
     askuser(Goal, CF, Rules), !,
     assert(known(Goal, CF)),
     above_threshold(CF, Threshold).
```

We start a consultation using a two-argument version of `solve`. The first argument is the top-level goal in the knowledge base, and the second is a variable that will be bound to the confidence in the goal's truth as inferred from the knowledge base. `solve/2` (`solve` with arity of 2) prints a set of instructions to the user, calls `retractall(known(_,_))` to clean up any residual information from previous uses of `exshell`, and calls `solve/4` initialized with appropriate values:

```
solve(Goal, CF) :-
     print_instructions,
     retractall(known(_, _)),
     solve(Goal, CF, [ ], 20).
```

`print_instructions` gives allowable responses to an exshell query:

```
print_instructions :- nl,
      write('Response must be either:'), nl,
      write('Confidence in truth of query.'), nl,
```

```
        write('A number between –100 and 100.'), nl,
        write('why.'), nl,
        write('how(X), where X is a goal'), nl.
```

The next set of predicates computes certainty factors. Again, `exshell` uses a form of the Stanford certainty factor algebra. Briefly, the certainty factor of the `and` of two goals is the minimum of the certainty factors of the individual goals; the certainty factor of the negation of a fact is –1 times the certainty of that fact. Confidence in a fact concluded using a rule equals the certainty of the premise multiplied by the certainty factor in the rule. `above_threshold` determines whether the value of a certainty factor is too low given a particular `threshold`. `exshell` uses the `threshold` value to prune a goal if its certainty gets too low.

Note that we define `above_threshold` separately for negative and positive values of the threshold. A positive `threshold` enables us to prune if the goal's confidence is less than `threshold`. However, a negative `threshold` indicates that we are trying to prove a goal false. Thus for negative goals, we prune search if the value of the goal's confidence is greater than the `threshold`. `invert_threshold` is called to multiply `threshold` by –1.

```
and_cf(A, B, A) :- A = < B.

and_cf(A, B, B) :- B < A.

negate_cf(CF, Negated_CF) :-
     Negated_CF is - 1 * CF.

rule_cf(CF_rule, CF_premise,CF) :-
     CF is (CF_rule * CF_premise/100).

above_threshold(CF, T) :-
     T > = 0, CF >= T.

above_threshold(CF, T) :-
     T < 0, CF =< T.

invert_threshold(Threshold, New_threshold) :-
     New_threshold is –1 * Threshold.
```

`askuser` writes out a query and reads the user's `Answer`; the `respond` predicates take the appropriate action for each user input.

```
askuser(Goal, CF, Rules) :-
     nl, write('User query:'),
     write(Goal), nl, write('?'),
     read(Answer),
     respond(Answer, Goal, CF, Rules).
```

The user can respond to this query with a CF between 100 and –100, for confidence in the goal's truth, `why` to ask why the question was asked, or `how(X)` to inquire how result `X` was established. The response to `why` is the rule currently on top of the rule stack. As with our implementation of Prolog in Prolog in Section 6.1, successive `why` queries will pop back up the rule stack, enabling the user to reconstruct the entire line of reasoning. If the user answer matches `how(X)`, `respond` calls `build_proof` to

build a proof tree for `X` and `write_proof` to print that proof in a readable form. There is a "catchall" `respond` for unknown input values.

```
% Case 1: user enters a valid confidence factor
respond(CF, _, CF, _) :-
     number(CF),
     CF =< 100, CF >= –100.
% Case 2: user enters a why query
respond(why, Goal, CF, [Rule | Rules]) :-
     write_rule(Rule),
     askuser(Goal, CF, Rules).
respond(why, Goal, CF, [ ]) :-
     write('Back to top of rule stack.'),
     askuser(Goal, CF, [ ]).
% Case 3: user enters a how query. Build/print proof
respond(how(X), Goal, CF, Rules) :-
     build_proof(X, CF_X, Proof), !,
     write(X), write('concluded with certainty'),
     write(CF_X), nl, nl,
     write('The proof is '), nl, nl,
     write_proof(Proof, 0), nl, nl,
     askuser(Goal, CF, Rules).
% User enters how query, could not build proof
respond(how(X), Goal, CF, Rules) :-
     write('The truth of '), write(X), nl,
     write('is not yet known.'), nl,
     askuser(Goal, CF, Rules).
% Case 4: User presents unrecognized input
respond(_, Goal, CF, Rules) :-
     write('Unrecognized response.'), nl,
     askuser(Goal, CF, Rules).
```

`build_proof` is parallel to `solve/4`, but `build_proof` does not ask the user for unknowns, as these were already saved as part of the case-specific data. `build_proof` constructs a proof tree as it proves the goal.

```
build_proof(Goal, CF, (Goal, CF :- given)) :-
     known(Goal, CF), !.
build_proof(not Goal, CF, not Proof) :- !,
     build_proof(Goal, CF_goal, Proof),
     negate_cf(CF_goal, CF).
build_proof((Goal_1, Goal_2), CF,
          (Proof_1, Proof_2)) :- !,
     build_proof(Goal_1, CF_1, Proof_1),
     build_proof(Goal_2, CF_2, Proof_2),
     and_cf(CF_1, CF_2, CF).
```

```
build_proof(Goal, CF, (Goal, CF :- Proof)) :-
     rule((Goal :- Premise), CF_rule),
     build_proof(Premise, CF_premise, Proof),
     rule_cf(CF_rule, CF_premise, CF).
build_proof(Goal, CF, (Goal, CF :- fact)) :-
     rule(Goal, CF).
```

The final predicates create a user interface. The interface requires the bulk of the code! First, we define a predicate `write_rule:`

```
write_rule(rule((Goal :- (Premise)), CF)) :-
     write(Goal), write(':-'), nl,
     write_premise(Premise), nl,
     write('CF = '), write(CF), nl.
write_rule(rule(Goal, CF)) :-
     write(Goal), nl, write('CF = '), write(CF), nl.
```

`write_premise` writes the conjuncts of a rule premise:

```
write_premise((Premise_1, Premise_2)) :- !,
     write_premise(Premise_1),
     write_premise(Premise_2).
write_premise(not Premise) :- !,
     write(''), write(not), write(''),
     write(Premise), nl.
write_premise(Premise) :-
     write(''), write(Premise), nl.
```

`write_proof` prints proof, using indents to show the tree's structure:

```
write_proof((Goal, CF :- given), Level) :-
     indent(Level), write(Goal), write(' CF= '),
     write(CF),write(' given by the user'), nl, !.
write_proof((Goal, CF :- fact), Level) :-
     indent(Level), write(Goal), write(' CF = '),
     write(CF),
     write(' was a fact of knowledge base'), nl, !.
write_proof((Goal, CF :- Proof), Level) :-
     indent(Level), write(Goal), write(' CF = '),
     write(CF), write(' :-'), nl, New_level is
          Level + 1,write_proof(Proof,New_level), !.
write_proof(not Proof, Level) :-
     indent(Level), write((not)), nl,
     New_level is Level + 1,
     write_proof(Proof, New_level), !.
write_proof((Proof_1, Proof_2),Level) :-
     write_proof(Proof_1, Level),
     write_proof(Proof_2, Level), !.

indent(0).
```

```
indent(l) :-
     write(''), l_new is l – 1, indent(l_new).
```

As an illustration of the behavior of `exshell`, consider the following sample knowledge base for diagnosing car problems. The top-level goal is fix/1. The knowledge base decomposes the problem solution into finding the `bad_system`, finding the `bad_component` within that system, and finally linking the diagnosis to `Advice` for its solution. Note that the knowledge base is incomplete; there are sets of symptoms that it cannot diagnose. In this case, `exshell` simply fails. Extending the knowledge base to some of these cases and adding a rule that succeeds if all other rules fail are interesting challenges and left as exercises. The following set of rules is segmented to show reasoning on each level of the search tree presented in Figure 6.1. The top segment, `rule((fix(Advice),` is at the root of the tree:

```
rule((fix(Advice) :-                % Top-level query
     (bad_component(X), fix(X,Advice))), 100).
rule((bad_component(starter) :-
     (bad_system(starter_system),
          lights(come_on))), 50).
rule((bad_component(battery) :-
     (bad_system(starter_system),
          not lights(come_on))), 90).
rule((bad_component(timing) :-
     (bad_system(ignition_system),
          not tuned_recently)), 80).
rule((bad_component(plugs) :-
     (bad_system(ignition_system),
          plugs(dirty))), 90).
rule((bad_component(ignition_wires) :-
     (bad_system(ignition_system),
          not plugs(dirty), tuned_recently)), 80).
rule((bad_system(starter_system) :-
     (not car_starts, not turns_over)), 90).
rule((bad_system(ignition_system) :-
     (not car_starts, turns_over, gas_in_carb)),80).
rule((bad_system(ignition_system) :-
     (runs(rough), gas_in_carb)), 80).
rule((bad_system(ignition_system) :-
     (car_starts, runs(dies), gas_in_carb)), 60).

rule(fix(starter, 'replace starter'), 100).
rule(fix(battery, 'replace/recharge battery'), 100).
rule(fix(timing, 'get the timing adjusted'), 100).
rule(fix(plugs, 'replace spark plugs'), 100).
rule(fix(ignition_wires, 'check ignition'),100).

askable(car_starts).       % May ask user about goal
```

```
askable(turns_over).
askable(lights(_)).
askable(runs(_)).
askable(gas_in_carb).
askable(tuned_recently).
askable(plugs(_)).
```

Next we demonstrate, `exshell` using this knowledge base. Figure 6.1 presents the trace and the search space: solid lines are searched, dotted lines are not searched, and bold lines indicate the solution.

```
?- solve(fix(X), CF).
Response must be either:
    A confidence in the truth of the query.
    This is a number between –100 and 100.
    why.
    how(X), where X is a goal
User query:car_starts
? –100.
User query:turns_over
? 85.
User query:gas_in_carb
? 75.
User query:tuned_recently
? –90.
    X = 'get the timing adjusted' CF = 48.0
```

We now run the problem again using `how` and `why` queries. Compare the responses with the corresponding subtrees and search paths of Figure 6.1:

?- solve(fix(X), CF).

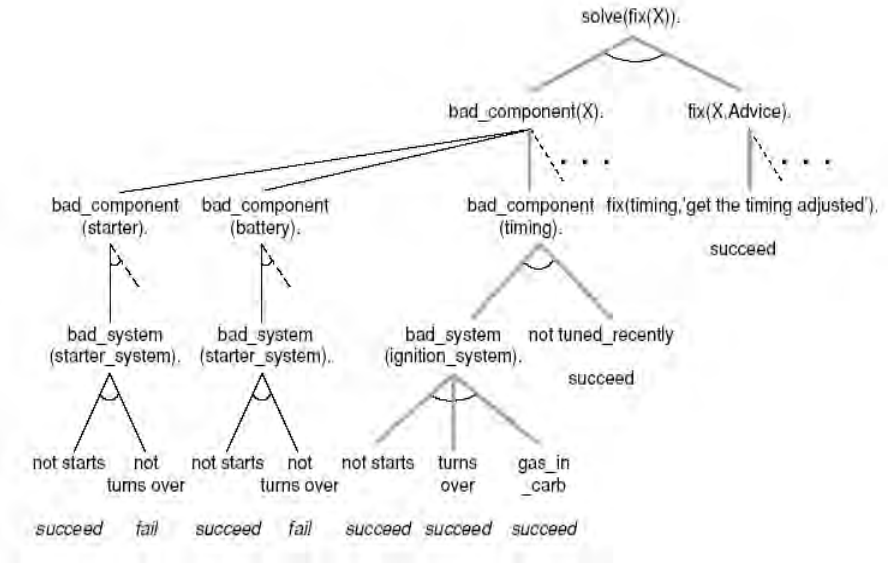

Figure 6.1. The graph searched in an automotive diagnosis consultation; dashed lines are branches not examined, bold lines indicate the solution.

```
Response must be either:
     A confidence in the truth of the query.
     This is a number between –100 and 100.
     why.
     how(X), where X is a goal
User query:car_starts
? –100.
User query:turns_over
? why.
bad_system(starter_system):-
     not car_starts
     not turns_over
     CF = 90
User query:turns_over
? why.
bad_component(starter):-
     bad_system(starter_system)
     lights(come_on)
     CF = 50
User query:turns_over
? why.
fix(_0):-
     bad_component(starter)
     fix(starter,_0)
     CF = 100
User query:turns_over
? why.
Back to top of rule stack.
User query:turns_over
? 85.
User query:gas_in_carb
? 75.
User query:tuned_recently
? why.
bad_component(timing):-
     bad_system(ignition_system)
     not tuned_recently
     CF = 80
User query:tuned_recently
? how(bad_system(ignition_system)).
bad_system(ignition_system) was concluded with
          certainty 60.0
The proof is
bad_system(ignition_system) CF= 60.0 :-
     not car_starts CF = –100 was given by the user
     turns_over CF = 85 was given by the user
     gas_in_carb CF = 75 was given by the user
User query:tuned_recently
? –90.
X = 'get the timing adjusted' CF = 48.0
```

6.3 A Prolog Planner

For the third meta-interpreter of Chapter 6 we present a predicate calculus-based planning algorithm. In many ways this approach to planning is similar to early work in planning at SRI-International (Fikes and Nilsson 1971, Fikes et al. 1972). Our planner is predicate calculus based in that the PC representation is used to describe both the states of the planning world (the state descriptions) as well as the rules for changing the state of the world. In this section we create a Prolog version of that algorithm.

We represent the states of the world, including the `start` and `goal`, as lists of predicates that have interpretations as states of the world. Thus, the `start` and `goal` states are each described as a list of predicates:

```
start = [handempty, ontable(b), ontable(c), on(a,b),
          clear(c), clear(a)]

goal = [handempty, ontable(a), ontable(b), on(c,b),
          clear(a), clear(c)]
```

These states are seen, with a portion of the search space, in Figure 6.2.

The moves in this blocks world are described using an *add* and *delete* list. The add and delete list describes how the list of predicates describing a new state of the solution is created from the list describing the previous state: some predicates are added to the state list and others are deleted. The `move` predicates for state change have three arguments. First is the `move` predicate name with its arguments. The second argument is the list of preconditions: the predicates that must be true of the description of the present state of the world for the `move` rule to be applied to that state. The third argument is the list containing the add and delete predicates: the predicates that are added to and/or deleted from the state of the world to create the new state of the world that results from applying the `move` rule. Notice how useful the ADT set operators of union, intersection, set difference, etc., are in manipulating the preconditions and the predicates in the add and delete list.

Four of the moves within this blocks world may be described:

```
move(pickup(X), [handempty, clear(X), on(X,Y)],
   [del(handempty), del(clear(X)), del(on(X,Y)),
        add(clear(Y)), add(holding(X))]).

move(pickup(X), [handempty, clear(X), ontable(X)],
     [del(handempty), del(clear(X)),
        del(ontable(X)), add(holding(X))]).

move(putdown(X), [holding(X)],
     [del(holding(X)), add(ontable(X)),
        add(clear(X)), add(handempty)]).

move(stack(X,Y), [holding(X), clear(Y)],
     [del(holding(X)), del(clear(Y)),
        add(handempty),add(on(X,Y)),add(clear(X))]).
```

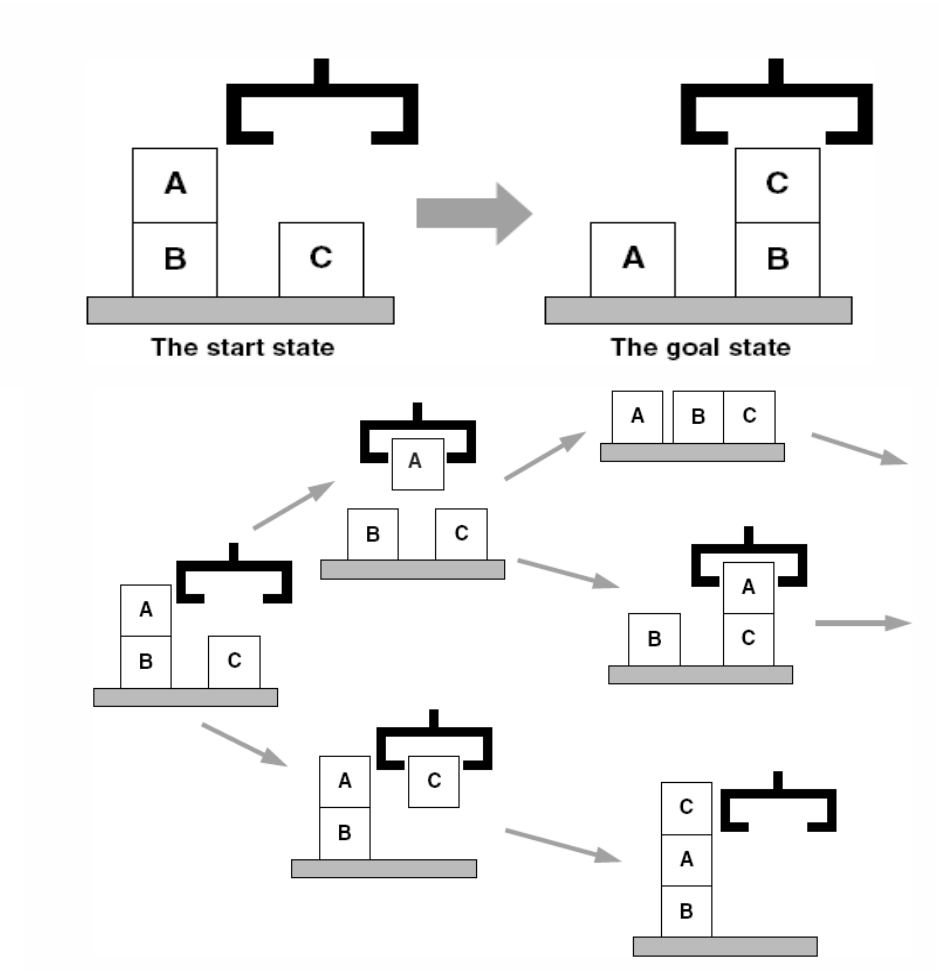

Figure 6.2. The start and goal states along with the initial portion of the search space for the blocks world planner.

Finally, we present the recursive controller for the plan generation. The first plan predicate gives the successful termination conditions (goal state description) for the plan when the `Goal` is produced. The final plan predicate states that after exhaustive search, no plan is possible. The recursive plan generator:

1. Searches for a `move` relationship.

2. Checks, using the `subset` operator, whether the state's `Preconditions` are met.

3. The `change_state` predicate produces a new `Child_state` using the add and delete list.
`member_stack` makes sure the new state has not been visited before.

4. The `stack` operator pushes the new `Child_state` onto the `New_move_stack`.

5. The `stack` operator pushes the original `Name` state onto the `New_been_stack`.

6. The recursive `plan` call searches for the next state using the `Child_state` and an updated `New_move_stack` and `Been_stack`.

A number of supporting utilities, built on the stack and set ADTs of Section 3.3 are included. Of course, the search being stack-based, is depth-first with backtracking and terminates with the first path found to a goal. It is left as an exercise to build other search strategies for planning, e.g., breadth-first and best-first planners.

```
plan(State, Goal, _, Move_stack) :-
     equal_set(State, Goal),
     write('moves are'), nl,
     reverse_print_stack(Move_stack).
plan(State, Goal, Been_stack, Move_stack) :-
     move(Name, Preconditions, Actions),
     conditions_met(Preconditions, State),
     change_state(State, Actions, Child_state),
     not(member_stack(Child_state, Been_stack)),
     stack(Name, Been_stack, New_been_stack),
     stack(Child_state, Move_stack, New_move_stack),
     plan(Child_state, Goal, New_been_stack,
          New_move_stack), !.
plan(_, _, _) :-
     write('No plan possible with these moves!').
conditions_met(P, S) :-
     subset(P, S).
change_state(S, [ ], S).
change_state(S, [add(P) | T], S_new) :-
     change_state(S, T, S2),
     add_if_not_in_set(P, S2, S_new), !.
change_state(S, [del(P) | T], S_new) :-
     change_state(S, T, S2),
     delete_if_in_set(P, S2, S_new), !.
reverse_print_stack(S) :-
     empty_stack(S).
reverse_print_stack(S) :-
     stack(E, Rest, S),
     reverse_print_stack(Rest), write(E), nl.
```

Finally, we create a `go` predicate to initialize the arguments for `plan`, as well as a `test` predicate to demonstrate an easy method to save repeated creation of the same input string.

```
go(Start, Goal) :-
     empty_stack(Move_stack),
     empty_stack(Been_stack),
     stack(Start, Been_stack, New_been_stack),
     plan(Start, Goal, New_been_stack, Move_stack).

test :-
go(
     [handempty, ontable(b), ontable(c),
          on(a,b), clear(c), clear(a)],
     [handempty, ontable(a), ontable(b), on(c,b),
          clear(a), clear(c)]
  ).
```

In Chapter 7 we present two machine learning algorithms in Prolog, *version space search* and *explanation based learning*.

Exercises

1. Extend the meta-interpreter for Prolog in Prolog (Section 6.1) to include `or` and the cut.

2. Further complete the rules used with the `exshell` cars example in the text. You might add several new sets of rules for the transmission, cooling system, and brakes.

3. Create a knowledge base for a new domain for the expert system `exshell`.

4. `exshell` currently allows the user to respond to queries by entering a confidence in the query's truth, a why query, or a how query. Extend the `respond` predicate to allow the user to answer with `y` if the query is true, n if it is false. These responses correspond to having certainty factors of 100 and -100.

5. As currently designed, if `exshell` cannot solve a goal using the rule base, it fails. Extend `exshell` so if it cannot prove a goal using the rules, and if it is not `askable`, it will call that goal as a Prolog query. Adding this option requires changes to both the `solve` and `build_proof` predicates.

6. Add a predicate that that exshell does not just fail if it cannot find a solution recommendation. This could be a `solve` predicate at the very end of all `solve` predicates that prints out some message about the state of the problem solving, perhaps by binding `X`, and linking it to some `Advice`, and then succeeds. This an important consideration, guaranteeing that `exshell` terminates gracefully.

7. Finish the code for the planner of Section 6.3. Add code for a situation that requires a new set of moves and has new objects in the domain, such as adding pyramids or spheres that cannot be stacked on.

8. Add appropriate predicates and ADTs to `plan` to implement a breadth-first search controller for the planner of Section 6.3.

9. Design a best-first search controller for the planner of Section 6.3. Add heuristics to the search of your planning algorithm. Can you specify a heuristic that is admissible (Luger 2009, Section 4.3)?

7 Machine Learning Algorithms in Prolog

Chapter Objectives

Two different machine learning algorithms
- Version space search
  - Specific-to-general
  - Candidate elimination
- Explanation-based learning
  - Learning from examples
  - Generalization

Prolog meta-predicates and interpreters for learning
- Version space search
- Explanation-based learning

Chapter Contents

7.1 Machine Learning: Version Space Search

In this section and the next, we implement two machine learning algorithms: *version space search* and *explanation-based learning*. The algorithms themselves are presented in detail in Luger (2009, Chapter 10). In this chapter, we first briefly summarize them and then implement them in Prolog. Prolog is used for machine learning because, as these implementations illustrate, in addition to the flexibility to respond to novel data elements provided by its powerful built-in pattern matching, its meta-level reasoning capabilities simplify the construction and manipulation of new representations.

The Version Space Search Algorithm

Version space search (Mitchell 1978, 1979, 1982) illustrates the implementation of inductive learning as search through a concept space. A concept space is a state space representation of all possible generalizations from data in a problem domain. Version space search takes advantage of the fact that generalization operations impose an ordering on the concepts in a space, and then uses this ordering to guide the search.

Generalization and *specialization* are the most common types of operations for defining a concept space. The primary generalization operations used in machine learning and expressed in the predicate calculus (Luger 2009, Chapter 2) are:

Replacing constants with variables. For example:

```
color(ball,red)
```

generalizes to

```
color(X,red)
```

Dropping conditions from a conjunctive expression.

```
shape(X, round) ^ size(X, small) ^ color(X, red)
```

generalizes to

```
shape(X,round) ^ color(X,red)
```

Adding a disjunct to an expression.

```
shape(X,round) ^ size(X,small) ^ color(X,red)
```

generalizes to

```
shape(X,round) ^ size(X,small) ^ (color(X,red) v
    color(X,blue))
```

Replacing a property with its parent in a class hierarchy. If `primary_color` is a superclass of `red`, then

```
color(X,red)
```

generalizes to

```
color(X, primary_color)
```

We may think of generalization in set theoretic terms: let `P` and `Q` be the sets of sentences matching the predicate calculus expressions `p` and `q`, respectively. Expression `p` is more general than `q` iff $Q \subseteq P$. In the above examples, the set of sentences that match `color(X, red)` contains the set of elements that match `color(ball, red)`. Similarly, in example 2, we may think of the set of round, red things as a superset of the set of small, red, round things. Note that the "more general than" relationship defines a partial ordering on the space of logical sentences. We express this using the "≥" symbol, where $p \geq q$ means that `p` is more general than `q`. This ordering is a powerful source of constraints on the search performed by a learning algorithm.

We formalize this relationship through the notion of *covering*. If concept `p` is more general than concept `q`, we say that *`p` covers `q`*. We define the covers relation: let `p(x)` and `q(x)` be descriptions that classify objects as being positive examples of a concept. In other words, for an object `x`, `p(x)` → `positive(x)` and `q(x)` → `positive(x)`. `p` covers `q` iff `q(x)`→ `positive(x)` is a logical consequence of `p(x)` → `positive(x).`

For example, `color(X, Y)` covers `color(ball, Z)`, which in turn covers `color(ball, red)`. As a simple example, consider a domain of objects that have properties and values:

```
Sizes = {large, small}
Colors = {red, white, blue}
Shapes = {ball, brick, cube}
```

These objects can be represented using the predicate `obj(Sizes, Color, Shapes)`. The generalization operation of replacing constants with variables defines the space of Figure 7.1. We may view inductive learning as searching this space for a concept that is consistent with all the training examples.

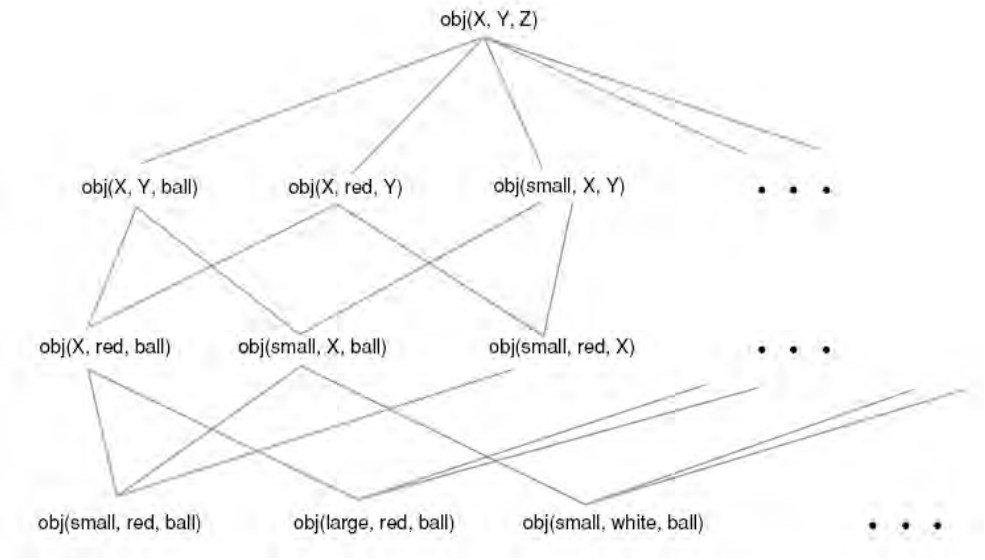

Figure 7.1. An example concept space.

We next present the *candidate elimination algorithm* (Mitchell 1982) for searching the concept space. This algorithm relies on the notion of a *version space*, which is the set of all concept descriptions consistent with the training examples. This algorithm works by reducing the size of the version space as more examples become available. The first two versions of this algorithm reduce the version space in a *specific to general* direction and a *general to specific* direction, respectively. The third version, called *candidate elimination*, combines these approaches into a bi-directional search. These versions of the candidate elimination algorithm are data driven; they generalize based on regularities found in the training data. Also, in using training data of known classification, these algorithms perform a variety of *supervised learning*.

Version space search uses both positive and negative examples of the target concept. Although it is possible to generalize from positive examples only, negative examples are important in preventing the algorithm from over generalizing. Not only must the learned concept be general enough to cover all positive examples; it also must be specific enough to exclude all negative examples. In the space of Figure 7.1, one concept that would cover all sets of exclusively positive instances would simply be `obj(X, Y, Z)`. However, this concept is probably too general, because it implies that all instances belong to the target concept. One way to avoid overgeneralization is to generalize as little as possible to cover positive examples; another is to use negative instances to eliminate overly general concepts. As Figure 7.2 illustrates, negative instances prevent overgeneralization by forcing the learner to specialize concepts in order to exclude negative instances. The algorithms of this section use both of these techniques.

We define *specific to general* search, for hypothesis set S, as:

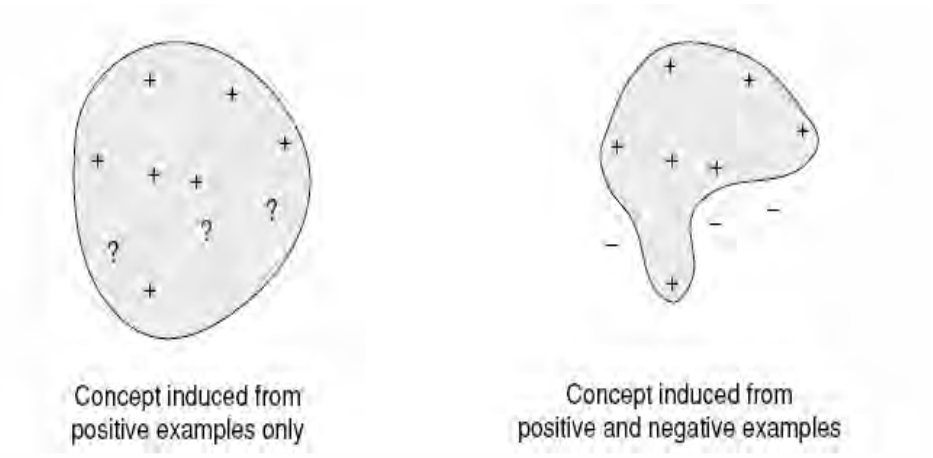

Figure 7.2. The role of negative examples in preventing overgeneralization.

```
Begin
Initialize S to first positive training instance;
N is the set of all negative instances seen so far;
For each positive instance p
    Begin
     For every s in S, if s does not match p,
     Replace s with its most specific
          generalization that matchs p;
     Delete from S all hypotheses more general than
          some other hypothesis in S;
     Delete from S all hypotheses that match a prev-
          iously observed negative instance in N;
    End;
For every negative instance n
    Begin
      Delete all members of S that match n;
      Add n to N to check future hypotheses
           for overgeneralization;
     End;
End
```

Specific to general search maintains a set, S, of *hypotheses*, or candidate concept definitions. To avoid overgeneralization, these candidate definitions are the *maximally specific generalizations* from the training data. A concept, `c`, is maximally specific if it covers all positive examples, none of the negative examples, and for any other concept, `c'`, that covers the positive examples, `c` $\leq$ `c'`. Figure 7.3 shows an example of applying this algorithm to the version space of Figure 7.1. The specific to general version space search algorithm is built in Prolog in Section 7.1.2.

We may also search the space in a general to specific direction. This algorithm maintains a set, G, of *maximally general concepts* that cover all of the positive and

none of the negative instances. A concept, **c**, is maximally general if it covers none of the negative training instances, and for any other concept, **c'**, that covers no negative training instance, **c ≥ c'**. In this algorithm, which we leave as an exercise, negative instances will lead to the specialization of candidate concepts while the algorithm uses positive instances to eliminate overly specialized concepts.

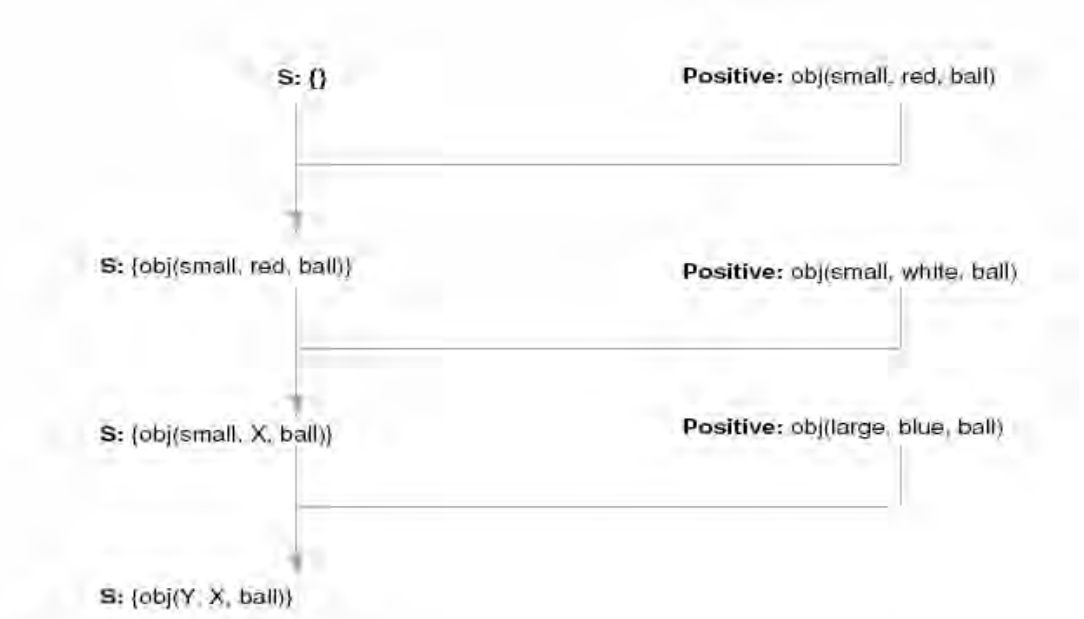

Figure 7.3. Specific to general version space search learning the concept "ball."

The *candidate elimination algorithm* combines these two approaches into a bi-directional search. This bi-directional approach has a number of benefits for learning. The algorithm maintains two sets of candidate concepts: **G**, the set of maximally general candidate concepts, and **S**, the set of maximally specific candidates. The algorithm specializes **G** and generalizes **S** until they converge on the target concept. The algorithm is described:

```
Begin
Initialize G to the most general concept in space;
Initialize S to first positive training instance;
For each new positive instance p
    Begin
     Delete all members of G that fail to match p;
     For every s in S, if s does not match p,
         replace s with its most specific
         generalizations that match p;
     Delete from S any hypothesis more general than
         some other hypothesis in S;
     Delete from S any hypothesis more general than
         some hypothesis in G;
    End;
```

```
For each new negative instance n
    Begin
     Delete all members of S that match n;
     For each g in G that matches n, replace g with
          its most general specializations that do
          not match n;
     Delete from G any hypothesis more specific than
          some other hypothesis in G;
     Delete from G any hypothesis more specific than
          some hypothesis in S;
    End;
If G = S and both are singletons, then the algorithm
    has found a single concept that is consistent
    with all the data;
If G and S become empty, then there is no concept
    that covers all positive instances and none of
    the negative instances;
End
```

Figure 7.4 illustrates the behavior of the candidate elimination algorithm in searching the version space of Figure 7.1. Note that the figure does not show those concepts that were produced through generalization or specialization but eliminated as overly general or specific. We leave the elaboration of this part of the algorithm as an exercise and show a partial implementation in the next section.

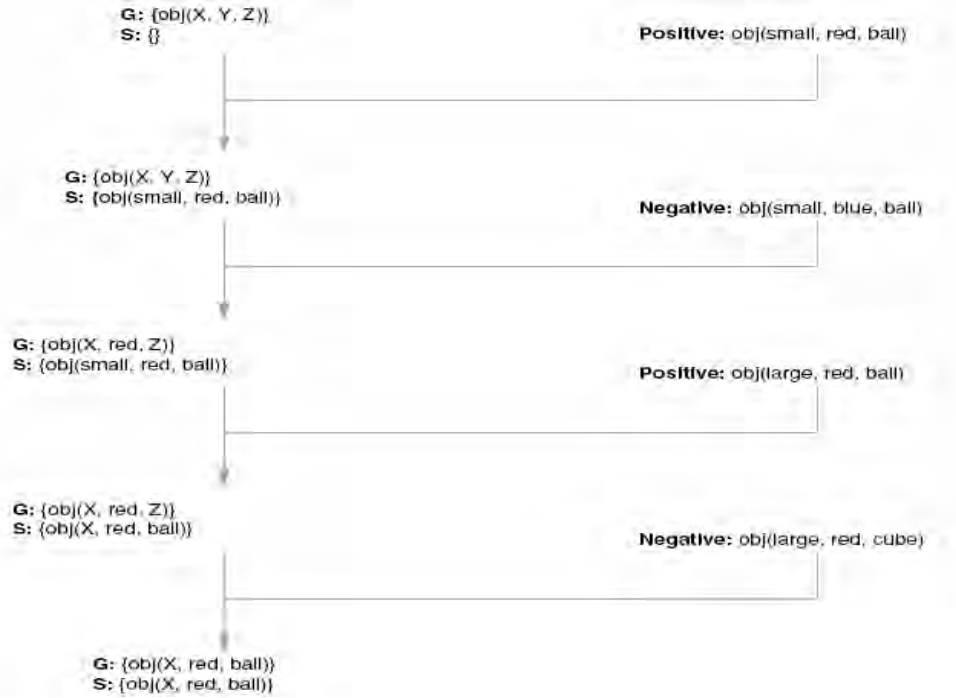

Figure 7.4. The *candidate elimination* algorithm learning the concept "red ball."

A Simple Prolog Program

We first implement the specific to general search and then the full bi-directional candidate elimination algorithm. We also give hints on how to construct the general to specific version space search. These search algorithms are independent of the representation used for concepts, as long as that representation supports appropriate generalization and specialization operations. We use a representation of objects as lists of features. For example, we describe a small, red, ball with the list:

```
[small, red, ball]
```

We represent the concept of all small, red things by including a variable in the list:

```
[small, red, X]
```

This representation we call a *feature vector,* It is less expressive than full logic, e.g., it cannot represent the class "all red or green balls." However, it simplifies generalization, and provides a strong *inductive bias* (Luger 2009, Section 10.4). We generalize a feature vector by substituting a variable for a constant, for example, the most specific common generalization of `[small, red, ball]` and `[small, green, ball]` is `[small, X, ball]`. This vector will cover both of the specializations and is the most specific vector to do so.

We define one feature vector as *covering* another if the first is either identical to or more general than the second. Note that unlike unification, `covers` is asymmetrical: values exist for which `X` covers `Y`, but `Y` does not cover `X`. For example, `[X, red, ball]` covers `[large, red, ball]` but the reverse is not true. We next define the predicate `covers` for feature vectors as:

```
covers([ ], [ ]).
covers([H1 | T1], [H2 | T2]) :-
     var(H1), var(H2), covers(T1, T2).
     % variables cover each other
covers([H1 | T1], [H2 | T2]) :-
     var(H1), atom(H2), covers(T1, T2).
     % a variable covers a constant
covers([H1 | T1], [H2 | T2]) :-
     atom(H1), atom(H2), H1 = H2,
     covers(T1, T2).
     % matching constants
```

We next need to determine whether one feature vector is strictly more general than another; i.e., the vectors are not identical. We define the `more_general/2` predicate as:

```
more_general(X, Y) :- not(covers(Y,X)),covers(X,Y).
```

We implement generalization of feature vectors as a predicate, `generalize` with three arguments, where the first argument is a feature vector representing an hypothesis (this vector may contain variables), the second argument is an instance, containing no variables. `generalize` binds its third argument to the most specific generalization of the

hypothesis that covers the instance. `generalize` recursively scans the feature vectors, comparing corresponding elements. If two elements match, the result contains the value of the hypotheses vector in that position; if two elements do not match, it places a variable in the corresponding position of the generalized feature vector. Note the use of the expression `not(Feature \= Inst_prop)`, in the second definition of `generalize`; this double negative enables us to test if two atoms will unify without actually performing the unification and forming any unwanted variable bindings. We define generalize:

```
generalize([ ], [ ], [ ]).
generalize([Feature | Rest],[Inst_prop | Rest_inst],
          [Feature | Rest_gen]) :-
     not(Feature \= Inst_prop),
     generalize(Rest, Rest_inst, Rest_gen).
generalize([Feature | Rest],[Inst_prop | Rest_inst],
          [_ | Rest_gen]) :-
     Feature \= Inst_prop,
     generalize(Rest, Rest_inst, Rest_gen).
```

These predicates define the essential operations on feature vector representations. The remainder of the implementation that follows is independent of any specific representation, and may be adapted to a variety of representations and generalization operators.

As discussed in Section 7.1, we may search a concept space in a specific to general direction by maintaining a list `H` of candidate hypotheses. The hypotheses in `H` are the most specific concepts that cover all the positive examples and none of the negative examples seen so far. The heart of the algorithm is `process` with five arguments. The first argument to `process` is a training instance, `positive(X)` or `negative(X)`, indicating that `X` is a positive or negative example. The second and third arguments are the current list of hypotheses and the list of negative instances. On completion, `process` binds its fourth and fifth arguments to the updated lists of hypotheses and to the negative examples, respectively.

The first clause in the definition below initializes an empty hypothesis set to the first positive instance. The second handles positive training instances by generalizing candidate hypotheses to cover the instance. It then deletes all over-generalizations by removing those that are more general than some other hypothesis and eliminating any hypothesis that covers some negative instance. The third clause in the definition handles negative examples by deleting any hypothesis that covers those instances.

```
process(positive(Instance), [ ], N, [Instance], N).
process(positive(Instance), H, N, Updated_H, N) :-
     generalize_set(H, Gen_H, Instance),
     delete(X, Gen_H,(member(Y, Gen_H),
          more_general(X, Y)), Pruned_H),
     delete(X, Pruned_H, (member(Y, N),
     covers(X, Y)), Updated_H).
```

```
process(negative(Instance), H, N, Updated_H,
          [InstanceN]) :-
     delete(X, H, covers(X, Instance), Updated_H).
process(Input, H, N, H, N):-      %Catches bad input
     Input \= positive(_),
     Input \= negative(_),
     write('Enter either positive(Instance) or
          negative(Instance) '), nl.
```

An interesting aspect of this implementation is the `delete` predicate, a generalization of the usual process of deleting all matches of an element from a list. One of the arguments to `delete` is a test that determines which elements to remove from the list. Using `bagof`, `delete` matches its first argument (usually a variable) with each element of its second argument (this must be a list). For each such binding, it then executes the test specified in argument three: this test is any sequence of callable Prolog goals. If a list element causes this test to fail, `delete` includes that element in the resulting list. It returns the result in its final argument. The `delete` predicate is an excellent example of the power of meta reasoning in Prolog: by letting us pass in a specification of the elements we want to remove from a list, `delete` gives us a general tool for implementing a range of list operations. Thus, `delete` lets us define the various filters used in `process/5` in an extremely compact fashion. We define `delete`:

```
delete(X, L, Goal, New_L) :-
     (bagof(X, (member(X, L), not(Goal)), New_L);
          New_L = [ ]).
```

`Generalize_set` is a straightforward predicate that recursively scans a list of hypotheses and generalizes each one against a training instance. Note that this assumes that we may have multiple candidate generalizations at one time. In fact, the feature vector representation of Section 7.1.1 only allows a single most specific generalization. However, this is not true in general and we have defined the algorithm for the general case.

```
generalize_set([ ], [ ], _).
generalize_set([Hypothesis | Rest],
          Updated_H, Instance):-
     not(covers(Hypothesis, Instance)),
     (bagof(X, generalize(Hypothesis, Instance, X),
          Updated_head); Updated_head = [ ]),
     generalize_set(Rest, Updated_rest, Instance),
     append(Updated_head, Updated_rest, Updated_H).

generalize_set([Hypothesis | Rest],
          [Hypothesis | Updated_rest], Instance) :-
     covers(Hypothesis, Instance),
     generalize_set(Rest, Updated_rest, Instance).
```

specific_to_general implements a loop that reads and processes training instances:

```
specific_to_general(H, N) :-
     write('H = '), write(H), nl, write('N = '),
     write(N), nl,
     write('Enter Instance: '), read(Instance),
     process(Instance, H, N, Updated_H, Updated_N),
     specific_to_general(Updated_H, Updated_N).
```

The following transcript illustrates the execution of this algorithm.

```
?- specific_to_general([], []).
H = [ ]
N = [ ]
Enter Instance: positive([small, red, ball]).
H = [[small, red, ball]]
N = [ ]
Enter Instance: negative([large, green, cube]).
H = [[small, red, ball]]
N = [[large, green, cube]]
Enter Instance: negative([small, blue, brick]).
H = [[small, red, ball]]
N = [[small, blue, brick], [large, green, cube]]
Enter Instance: positive([small, green, ball]).
H = [[small, _66, ball]]
N = [[small, blue, brick], [large, green, cube]]
Enter Instance: positive([large, blue, ball]).
H = [[_116, _66, ball]]
N = [[small, blue, brick], [large, green, cube]]
```

The second version of the algorithm searches in a general to specific direction, as described in Section 7.1.1. In this version, the set of candidate hypotheses are initialized to the most general possible concept. In the case of the feature vector representation, this is a list of variables. It then specializes candidate concepts to prevent them from covering negative instances. In the feature vector representation, this involves replacing variables with constants. When given a new positive instance, it eliminates any candidate hypothesis that fails to cover that instance.

We implement this algorithm in a way that closely parallels the specific to general search just described, including the use of the general delete predicate to define the various filters of the list of candidate concepts. In defining a general to specific search, process will have six arguments. The first five reflect the specific to general version: the first a training instance of the form `positive(Instance)` or `negative(Instance)`; the second is a list of candidate hypotheses; these are the most general hypotheses that cover no negative instances. The third argument is the list of positive examples, used to delete any overly specialized candidate hypothesis. The fourth and fifth arguments are the updated lists of hypotheses and positive examples, respectively. The sixth argument is a list of allowable variable substitutions for specializing concepts.

Specialization by substituting a constant for a variable requires the algorithm to know the allowable constant values for each field of the feature vector. These values will have to be passed in as the sixth argument of process. In our example of `[Size, Color, Shape]` vectors, a sample list of types might be: `[[small, medium, large], [red, white, blue], [ball, brick, cube]]`. Note that the position of each sublist determines the position in a feature vector where those values are used; for example, the first sublist defines allowable values for the first position of a feature vector. We leave construction of this algorithm as an exercise. For guidance we include a run of our implementation:

```
?- general_to_specific([[_, _, _]], [ ],
          [[small, medium, large],
           [red, blue, green],
           [ball, brick, cube]]).
H = [[_0, _1, _2]]
P = [ ]
Enter Instance: positive([small, red, ball]).
H = [[_0, _1, _2]]
P = [[small, red, ball]]
Enter Instance; negative([large, green, cube]).
H = [[small, _89, _90], [_79, red, _80],
          [_69, _70, ball]]
P = [[small, red, ball]]
Enter Instance: negative([small, blue, brick]).
H = [[_79, red, _80],[_69, _70, ball]]
P = [[small, red, ball]]
Enter Instance: positive([small, green, ball]).
H = [[_69,_70,ball]]
P = [[small, green, ball], [small, red, ball]]
```

The full candidate elimination algorithm, as described in Section 7.1.1, is a combination of the two single direction searches. As before, the heart of the algorithm is the definition of `process`, with six arguments. The first argument to `process` is a training instance. Arguments two and three are `G` and `S`, the sets of maximally general and maximally specific hypotheses respectively. The fourth and fifth arguments are bound to the updated versions of these sets. The sixth argument of `process` lists allowable variable substitutions for specializing feature vectors.

On positive instances, `process` generalizes `S`, the set of most specific generalizations, to cover the training instance. It then eliminates any elements of `S` that have been over generalized. It also eliminates any elements of `G` that fail to cover the training instance. It is interesting to note that an element of `S` is overly general if there is no element of `G` that covers it; this is true because `G` contains those candidate hypotheses that are both maximally general and cover no negative instances. `process` uses `delete` to eliminate these hypotheses.

On a negative training instance, `process` specializes all hypotheses in `G` to exclude that instance. It also eliminates any candidates in `S` that cover the negative instance. As discussed above, specialization of feature vectors requires replacing variables with constants. This requires that we pass a list of allowable substitutions as the sixth argument to `process`. We define `process`:

```
process(negative(Instance), G, S, Updated_G,
          Updated_S, Types) :-
     delete(X, S, covers(X, Instance), Updated_S),
     specialize_set(G, Spec_G, Instance, Types),
     delete(X, Spec_G, (member(Y, Spec_G),
          more_general(Y, X)), Pruned_G),
     delete(X, Pruned_G, (member(Y, Updated_S),
          not(covers(X, Y))), Updated_G).
process(positive(Instance), G, [ ],
          Updated_G, [Instance],_) :-  %Initialize S
     delete(X, G, not(covers(X, Instance)),
          Updated_G).
process(positive(Instance), G, S,
          Updated_G, Updated_S,_) :-
     delete(X, G, not(covers(X, Instance)),
          Updated_G),
     generalize_set(S, Gen_S, Instance),
     delete(X, Gen_S, (member(Y, Gen_S),
          more_general(X, Y)), Pruned_S),
     delete(X, Pruned_S, not((member(Y, Updated_G),
          covers(Y, X))), Updated_S).
process(Input, G, P, G, P,_) :-
     Input \= positive(_), Input \= negative(_),
     write(`Enter a positive(Instance) or
          negative(Instance): '), nl.
```

`generalize_set` generalizes all members of a set of candidate hypotheses to cover a training instance. It is identical to the version defined for the specific to general search. `specialize_set` takes a set of candidate hypotheses and computes all maximally general specializations of those hypotheses that exclude (do not cover) a training instance. Note the use of `bagof` to get all specializations.

```
specialize_set([ ], [ ], _, _).
specialize_set([HypothesisRest],
          Updated_H, Instance, Types) :-
     covers(Hypothesis, Instance),
     (bagof(Hypothesis, specialize(Hypothesis,
          Instance,Types), Updated_head) ;
          Updated_head = [ ]),
     specialize_set(Rest, Updated_rest, Instance,
          Types),
     append(Updated_head, Updated_rest, Updated_H).
```

```
specialize_set([HypothesisRest],
          [HypothesisUpdated_rest],Instance,Types):-
     not (covers(Hypothesis, Instance)),
     specialize_set(Rest, Updated_rest,
          Instance, Types).
```

`specialize` finds an element of a feature vector that is a variable. It binds that variable to a constant value that it selects from the list of allowable values, and which does not match the training instance. Recall that `specialize_set` called `specialize` with `bagof` to get all specializations. If we call `specialize` once, it will substitute a constant into the first variable; `bagof` causes it to produce all specializations.

```
specialize([Prop_], [Inst_prop_],
          [Instance_values_]) :-
     var(Prop), member(Prop, Instance_values),
     Prop \= Inst_prop.
specialize([_Tail], [_Inst_tail], [_Types]) :-
     specialize(Tail, Inst_tail, Types).
```

The definitions of `generalize`, `more_general`, `covers`, and `delete` are the same as in the specific to general algorithm defined above. `candidate_elim` implements a top-level read-process loop, printing out the current `G` set, the `S` set, and calls `process` on the input:

```
candidate_elim([G],[S],_) :-
     covers(G,S),covers(S,G),
     write('target concept is: '), write(G),nl.
candidate_elim(G, S, Types) :-
     write('G= '), write(G), nl, write('S= '),
     write(S), nl, write('Enter Instance: '),
     read(Instance),
     process(Instance, G, S, Updated_G,
          Updated_S, Types),
     candidate_elim(Updated_G, Updated_S, Types).
```

To conclude this section we present a trace of the candidate elimination algorithm. Note initializations of `G`, `S`, and the list of allowable substitutions:

```
?- candidate_elim([[_, _, _]], [ ],
          [[small, medium, large],
           [red, blue, green],
           [ball, brick, cube]]).
G= [[_0, _1, _2]]
S= [ ]
Enter Instance: positive([small, red, ball]).
G= [[_0, _1, _2]]
S= [[small, red, ball]]
Enter Instance: negative([large, green, cube]).
G= [[small, _96, _97], [_86, red, _87],
          [_76, _77, ball]]
```

```
S= [[small, red, ball]]
Enter Instance: negative([small, blue, brick]).
G= [[_86, red, _87], [_76, _77, ball]]
S= [[small, red, ball]]
Enter Instance: positive([small, green, ball]).
G= [[_76, _77, ball]]
S= [[small, _351, ball]]
Enter Instance: positive([large, red, ball]).
target concept is: [_76, _77, ball] yes
```

7.2 Explanation Based Learning in Prolog

The Explanation Based Learning Algorithm

In this section, we describe briefly the algorithms for explanation-based learning, Section 7.2.1 and then present a Prolog implementation of the explanation-based learning in Section 7.2.2. Our implementation is based upon Kedar-Cabelli and McCarty's formulation (Kedar-Cabelli and McCarty 1987; Luger 2009, Section 10.5.2), called `prolog_ebg`, and illustrates the power of unification in Prolog. Even though it is quite difficult to implement explanation-based learning in many languages, the Prolog version is fairly simple.

Explanation-based learning uses an explicitly represented domain theory to construct an explanation of a training example, usually a proof that the example logically follows from the theory. By generalizing from the explanation of the instance, rather than from the instance itself, explanation-based learning filters noise, selects relevant aspects of experience, and organizes training data into a coherent structure.

There are several alternative formulations of this idea. For example, the STRIPS program for representing general operators for planning (see Section 6.3) has exerted a powerful influence on this research (Fikes et al. 1972). Meta-DENDRAL established the power of theory-based interpretation of training instances (Luger 2009, Section 10.5.1). A number of authors (DeJong and Mooney 1986, Minton 1988) have proposed alternative formulations of this idea. The *Explanation-Based Generalization* algorithm of Mitchell et al. (1986) is also typical of the genre. In this section, we examine a variation of the explanation-based learning (EBL) algorithm developed by DeJong and Mooney (1986).

EBL begins with:

1. *A target concept*. The learner's task is to determine an effective definition of this concept. Depending upon the specific application, the target concept may be a classification, a theorem to be proven, a plan for achieving a goal, or a heuristic for a problem solver.
2. *A training example*, an instance of the target.
3. *A domain theory*, a set of rules and facts that are used to explain how the training example is an instance of the goal concept.
4. *Operationality criteria*, some means of describing the form concept definitions may take.

To illustrate EBL, we present an example of learning about when an object is a cup. This is a variation of a problem explored by Winston et al. (1983) and adapted to explanation-based learning by Mitchell et al. (1986). The target concept is a rule that may be used to infer whether an object is a cup; again, we adopt a predicate calculus representation:

```
premise(X) → cup(X)
```

where `premise` is a conjunctive expression containing the variable `X`.

Assume a domain theory that includes the following rules about cups:

```
liftable(X) ^ holds_liquid(X) → cup(X)

part(Z, W) ^ concave(W) ^ points_up(W) →
     holds_liquid(Z)

light(Y) ^ part(Y, handle) → liftable(Y)

small(A) → light(A)

made_of(A, feathers) → light(A)
```

The training example is an instance of the goal concept. That is, we are given:

```
cup(obj1)
small(obj1)
part(obj1, handle)
owns(bob, obj1)
part(obj1, bottom)
part(obj1, bowl)
points_up(bowl)
concave(bowl)
color(obj1, red)
```

Finally, assume the operationality criteria require that target concepts be defined in terms of observable, structural properties of objects, such as `part` and `points_up`. We may provide domain rules that enable the learner to infer whether a description is operational, or we may simply list operational predicates.

A theorem prover constructs an explanation of why the example is an instance of the training concept: a proof that the target concept logically follows from the example, as in Figure 7.5. Note that this explanation eliminates such irrelevant concepts as `color(obj1, red)` from the training data and captures (only) those aspects of the example known to be relevant to the goal.

The next stage of explanation-based learning generalizes the explanation to produce a concept definition that may be used to recognize other cups. EBL accomplishes this by substituting variables for those constants in the proof tree that depend solely on the particular training instance, as may be seen in Figure 7.5 (bottom). Based on the generalized tree, EBL defines a new rule whose conclusion is the root of the tree and whose premise is the conjunction of the leaves:

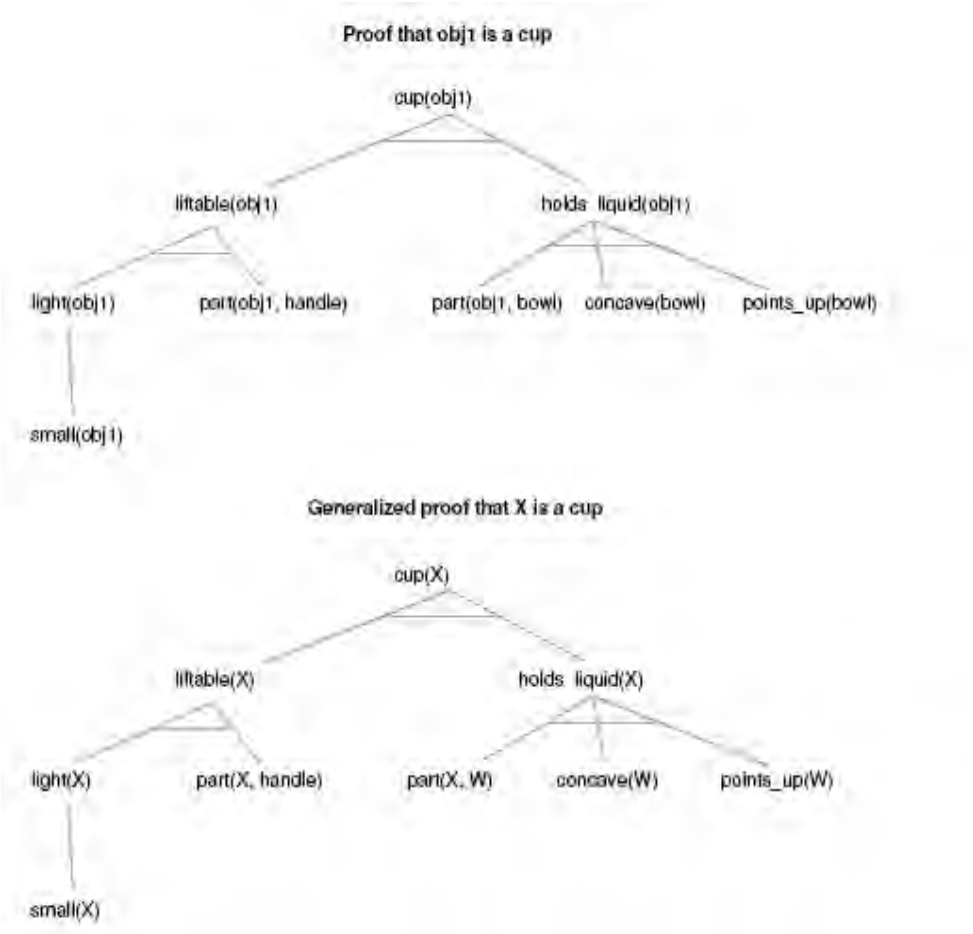

Figure 7.5. A specific (top) and generalized (bottom) proof that an object, X, is a cup.

```
small(X) ^ part(X, handle) ^ part(X, W) ^ concave(W)
  ^ points_up(W) → cup(X).
```

In constructing a generalized proof tree, our goal is to substitute variables for those constants that are part of the training instance while at the same time retaining those constants and constraints that are part of the domain theory. In this example, the constant handle originated in the domain theory rather than the training instance. We have retained it as an essential constraint in the acquired rule.

We may construct a generalized proof tree in a number of ways using a training instance as a guide. Mitchell et al. (1986) accomplish this by first constructing a proof tree that is specific to the training example and subsequently generalizing the proof through a process called *goal regression*. Goal regression matches the generalized goal (in our example, `cup(X)`) with the root of the proof tree, replacing constants with variables as required for the match. The algorithm applies these substitutions recursively through the tree until all appropriate constants have been generalized. See Mitchell et al. (1986) for a detailed description of this process. We next implement the explanation based learning algorithm in Prolog.

Prolog Implementation of EBL

Instead of building an explanation structure and maintaining separate sets of specific and general substitutions as done in Section 7.2.1, our algorithm will build both the proof of the training instance and the generalized proof tree concurrently.

For this example, we represent proof trees as we did in `exshell` (Section 6.2). When `prolog_ebg` discovers a fact, it returns this fact as the leaf of a proof tree. The proof of conjunctive goals is the conjunction of the proof of the conjuncts. The proof of a goal that requires rule chaining is represented as `(Goal :- Proof)`, where `Proof` becomes bound to the proof tree for the rule premise.

The heart of the algorithm is `prolog_ebg`. This predicate takes four arguments: the first is the goal being proved in the training example, the second is the generalization of that goal. If the domain theory enables a proof of the specific goal, it binds the third and fourth arguments to a proof tree for the goal and the generalization of that proof. For instance, implementing the cup example from Section 7.2.1, we would call `prolog_ebg` with the arguments:

```
prolog_ebg(cup(obj1), cup(X), Proof, Gen_proof).
```

We assume that Prolog has the domain theory and training instance of Section 7.2.1. When `prolog_ebg` succeeds; `Proof` and `Gen_proof` are the proof trees of Figure 7.5.

`prolog_ebg` is a straightforward variation of the `exshell` meta-interpreter of Section 6.2. The primary difference is that `prolog_ebg` solves the goal and the generalized goal in parallel. A further interesting aspect of the algorithm is the use of the predicate `duplicate` to create two versions of each rule: the first version is the rule as it appears in the domain theory, the second binds variables in the rule to the values in the training instance. We define `prolog_ebg`:

```
prolog_ebg(A, GenA, A, GenA) :- clause(A, true).
prolog_ebg((A, B), (GenA, GenB), (AProof, BProof),
          (GenAProof, GenBProof)) :- !,
     prolog_ebg(A, GenA, AProof, GenAProof),
     prolog_ebg(B, GenB, BProof, GenBProof).
prolog_ebg(A, GenA, (A :- Proof), (GenA :-
          GenProof)) :-
     clause(GenA, GenB),
     duplicate((GenA :- GenB), (A :- B)),
     prolog_ebg(B, GenB, Proof, GenProof).
```

`Duplicate` relies upon the behavior of `assert` and `retract` to create a copy of a Prolog expression with all new variables.

```
duplicate(Old, New) :-
     assert('$marker'(Old)),
     retract('$marker'(New)).
```

`extract_support` returns the sequence of the highest level operational nodes, as defined by the predicate `operational`. The

`extract_support` predicate implements a recursive tree walk, terminating the recursion when it finds nodes in the proof tree that qualifies as `operational`.

```
extract_support(Proof, Proof) :- operational(Proof).
extract_support((A :- _), A) :- operational(A).
extract_support((AProof, BProof), (A, B)) :-
     extract_support(AProof, A),
     extract_support(BProof, B).
extract_support((_ :- Proof), B) :-
     extract_support(Proof, B).
```

The final component of the explanation based generalization algorithm constructs the learned rule, using the `prolog_ebg` and `extract_support` predicates:

```
ebg(Goal, Gen_goal, (Gen_goal :- Premise)) :-
     prolog_ebg(Goal, Gen_goal, _, Gen_proof),
     extract_support(Gen_proof, Premise).
```

We illustrate the execution of these predicates with the example of learning structural definitions of cups from Section 7.2.1, as described originally by Mitchell et al. (1986). We begin with a domain theory for cups and other physical objects. The theory includes the rules:

```
cup(X) :- liftable(X), holds_liquid(X).
holds_liquid(Z) :-
     part(Z, W), concave(W), points_up(W).
liftable(Y) :-
     light(Y), part(Y, handle).
light(A):- small(A).
light(A):- made_of(A, feathers).
```

The learner is also given the following example, in which `obj1` is known to be a `cup`:

```
small(obj1).
part(obj1, handle).
owns(bob, obj1).
part(obj1, bottom).
part(obj1, bowl).
points_up(bowl).
concave(bowl).
color(obj1, red).
```

The operationality criteria define predicates that may be used in a rule:

```
operational(small(_)).
operational(part(_, _)).
```

```
operational(owns(_, _)).

operational(points_up(_)).

operational(concave(_)).
```

A run of the algorithm on the cup example illustrates the behavior of these predicates. Of course, symbols such as "_0" and "_106" indicate specific variables in Prolog, i.e., all uses of _106 represent the same variable:

```
?- prolog_ebg(cup(obj1), cup(X), Proof, Gen_proof).

X = _0,

Proof = cup(obj1) :-
        ( (liftable(obj1) :-
            ( (light(obj1) :-
                 small(obj1)),
                 part(obj1, handle))),
          (holds_liquid(obj1) :-
               (part(obj1, bowl),
                 concave(bowl),
                 points_up(bowl))))

 Gen_prooof = cup(_0) :-
        ( (liftable(_0) :-
            ( (light(_0) :-
                 small(_0)),
                 part(_0, handle))),
          (holds_liquid(_0) :-
               (part(_0, _106),
                 concave(_106),
                 points_up(_106))))
```

When we give `extract_support` the generalized proof from the previous execution of `prolog_ebg`, it returns the operational nodes of the proof, in left-to-right order:

```
?- extract_support((cup(_0) :-
        ( (liftable(_0) :-
              ( (light(_0) :-
                 small(_0)),
                 part(_0, handle))),
          (holds_liquid(_0) :-
              (part(_0,_106),
               concave(_106),
               points_up(_106))))), Premise),
_0 = _0, _106 = _1,
Premise = (small(_0), part(_0,handle)), part(_0,_1),
          concave(_1), points_up(_1)
```

Finally, `ebg` uses these predicates to construct a new rule from the example.

```
?- ebg(cup(obj1), cup(X), Rule).

X = _0,

Rule = cup(_0) :-
     ((small(_0), part(_0, handle)), part(_0,_110),
          concave(_110), points_up(_110))
```

In the next two chapters we address the problem of understanding natural language. We first, in Chapter 8, discuss issues in semantic (or language

meaning) representations, building Prolog structures for conceptual dependencies. We then build several recursive descent parsers to capture syntactic relationships in sentences. These meta-interpreters demonstrate context free, context sensitive, deterministic, and probabilistic parsing. In Chapter 9 we present the Earley parser in Prolog, which uses techniques from dynamic programming. The Earley parser is often called a *chart* parser.

Exercises

1. The run of the candidate elimination algorithm shown in Figure 7.4 does not show candidate concepts that were produced but eliminated because they were either overly general, overly specific, or subsumed by some other concept. Re-do the execution trace, showing these concepts and the reasons each was eliminated.

2. Develop a domain theory for explanation-based learning in some problem area of your choice. Trace the behavior of an explanation-based learner in applying this theory to several training instances.

3. Implement a general to specific search of the version space using the feature vector representation of Section 7.2. We can specialize feature vectors by replacing variables with constants; since this requires telling the algorithm of allowable values for each field of the feature vector, we must pass this in as an extra argument. The following definition of `run_general`, the top-level goal, illustrates the necessary initializations for the example used in the text: objects may be `small`, `medium`, or `large`, their color may be `red`, `blue`, `green`, and their shape may be `ball`, `brick`, or `cube`.

```
run_general :-
     general_to_specific([[_, _, _]], [ ],
          [[small,medium,large], [red,blue,green],
          [ball,brick,cube]]).
```

4. Create another domain theory example, as proposed in exercise 2 above, and run it with `prolog_ebg`.

5. Extend the definition of `ebg` so that, after it constructs a new rule, it asserts it to the logic database where it may be used in future queries. Test the performance of the resulting system using a theory for a suitably rich domain. You might do this by constructing a theory for a domain of your choice, or extending the theory from the cup example to allow it to explain different types of cups such as Styrofoam cups, cups without handles, etc.

8 Natural Language Processing in Prolog

Chapter Objectives

Natural language processing representations were presented
- Semantic relationships
  - Conceptual graphs
  - Verb-based case frames

Prolog was used to build a series of parsers
- Context free parsers
  - Deterministic
- Probabilistic Parsers
  - Probabilistic measures for sentence structures and words
  - Lexicalized probabilistic parsers capture word combination plausibility
- Context sensitive parsers
  - Deterministic

Recursive descent semantic net parsers
- Enforce word-based case frame constraints

Chapter Contents

8.1 Natural Language Understanding in Prolog

Because of its declarative semantics, built-in search, and pattern matching, Prolog provides an important tool for programs that process natural language. Indeed, natural language understanding was one of Prolog's earliest applications. As we will see with many examples in this chapter, we can write natural language grammars directly in Prolog, for example, context-free, context-sensitive, recursive descent semantic network, as well as stochastic parsers. Semantic representations are also easy to create in Prolog, as we see for conceptual graphs and case frames in Section 8.2. Semantic relationships may be captured either using the first-order predicate calculus or by a meta-interpreter for another representation, as suggested by semantic networks (Section 2.4.1) or frames (Sections 2.4.2 and 8.1). This not only simplifies programming, but also keeps a close connection between theories and their implementation.

In Section 8.3 we present a context-free parser and later add context sensitivity to the parse Section 8.5. We accomplish many of the same justifications for context sensitivity in parsing, e.g., noun-verb agreement, with the various probabilistic parsers of Section 8.4. Finally, semantic

inference, using graph techniques including `join`, `restrict`, and `inheritance` in conceptual graphs, can be done directly in Prolog as we see in Section 8.5.

Many of the approaches to parsing presented in this chapter have been suggested by several generations of colleagues and students.

8.2 Prolog-Based Semantic Representations

Following on the early work in AI developing representational schemes such as semantic networks, scripts, and frames (Luger 2009, Section 7.1) a number of *network languages* were developed to model the semantics of natural language and other domains. In this section, we examine a particular formalism to show how, in this situation, the problems of representing meaning were addressed. John Sowa's *conceptual graphs* (Sowa 1984) is an example of a network representation language. We briefly introduce conceptual graphs and show how they may be implemented in Prolog. A more complete introduction to this representational formalism may be found in Sowa (1984) and Luger (2009, Section 7.2).

A *conceptual graph* is a finite, connected, bipartite graph. The nodes of the graph are either *concepts* or *conceptual relations*. Conceptual graphs do not use labeled arcs; instead the conceptual relation nodes represent relations between concepts. Because conceptual graphs are bipartite, concepts only have arcs to relations, and vice versa. In Figure 8.1 `dog` and `brown` are concept nodes and `color` a conceptual relation. To distinguish these types of nodes, we represent concepts as boxes and conceptual relations as ellipses.

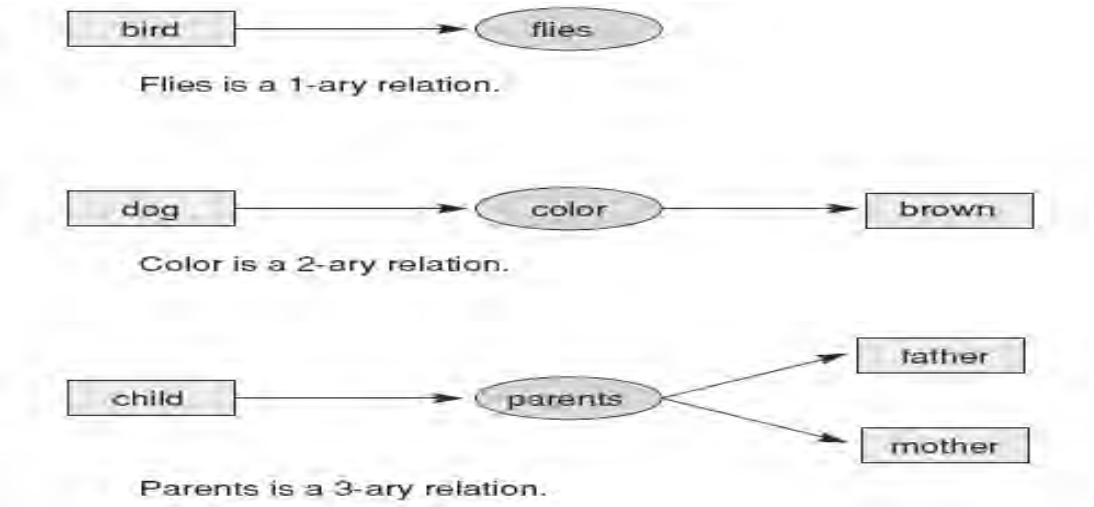

Figure 8.1. Conceptual graph relations with different arities.

In conceptual graphs, concept nodes represent either concrete or abstract objects in the world of discourse. Concrete concepts, such as a cat, telephone, or restaurant, are characterized by our ability to form an image of them in our minds. Note that concrete concepts include generic concepts such as cat or restaurant along with concepts of specific cats and restaurants. We can still form an image of a generic cat. Abstract concepts

include things such as love, beauty, and loyalty that do not correspond to images in our minds.

Conceptual relation nodes indicate a relation involving one or more concepts. One advantage of formulating conceptual graphs as bipartite graphs rather than using labeled arcs is that it simplifies the representation of relations of any number of arcs (arity). A relation of arity `n` is represented by a conceptual relation node having `n` arcs, as shown in Figure 8.1.

Each conceptual graph represents a single proposition. A typical knowledge base will contain a number of such graphs. Graphs may be arbitrarily complex but must be finite. For example, one graph in Figure 8.1 represents the proposition "A dog has a color of brown." Figure 8.2 is a graph of somewhat greater complexity that represents the sentence "Mary gave John the book." This graph uses conceptual relations to represent the cases of the verb "to give" and indicates the way in which conceptual graphs are used to model the semantics of natural language.

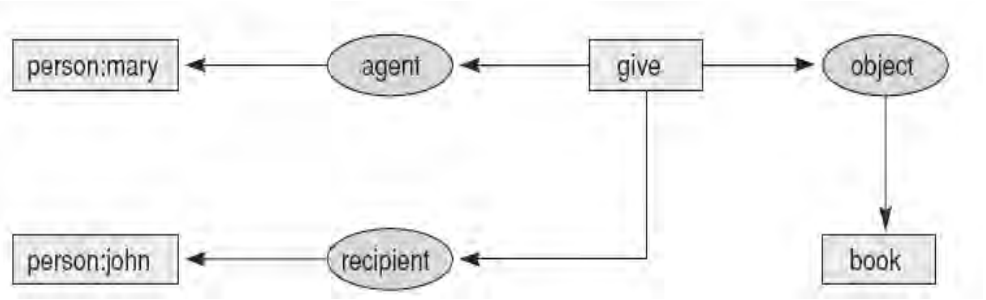

Figure 8.2. Conceptual graph of "Mary gave John the book."

Conceptual graphs can be translated directly into predicate calculus and hence into Prolog. The conceptual relation nodes become the predicate name, and the arity of the relation indicates the number of arguments of the predicate. Each Prolog predicate, as with each conceptual graph, represents a single proposition.

The conceptual graphs of Figure 8.1 may be rendered in Prolog as:

```
bird(X), flies(X).
dog(X), color (X, Y), brown(Y).
child(X), parents(X, Y, Z), father(Y), mother(Z).
```

where `X`, `Y`, and `Z` are bound to the appropriate individuals. Type information can be added to parameters as indicated in Section 5.2. We can also define the type hierarchy through a variation of `isa` predicates.

In addition to concepts, we define the relations to be used in conceptual graphs. For this example, we use the following concepts:

> *agent* links an act with a concept of type animate. agent defines the relation between an action and the animate object causing the action.
>
> *experiencer* links a state with a concept of type animate. It defines the relation between a mental state and its experiencer.
>
> *instrument* links an act with an entity and defines the instrument used in an action.

object links an event or state with an entity and represents the verb–object relation.

part links concepts of type physobj and defines the relation between whole and part.

The verb plays a particularly important role in building an interpretation, as it defines the relationships between the subject, object, and other components of the sentence. We can represent each verb using a *case frame* that specifies:

The linguistic relationships (agent, object, instrument, and so on) appropriate to that particular verb. Transitive verbs, for example, can have a direct object; intransitive verbs do not.

Constraints on the values that may be assigned to any component of the case frame. For example, in the case frame for the verb bites, we have asserted that the agent of biting must be of the type dog. This causes "Man bites dog" to be rejected as semantically incorrect.

Default values on components of the case frame. In the "bites" frame, we have a default value of teeth for the concept linked to the instrument relation.

The case frames for the verbs like and bite appear in Figure 8.3.

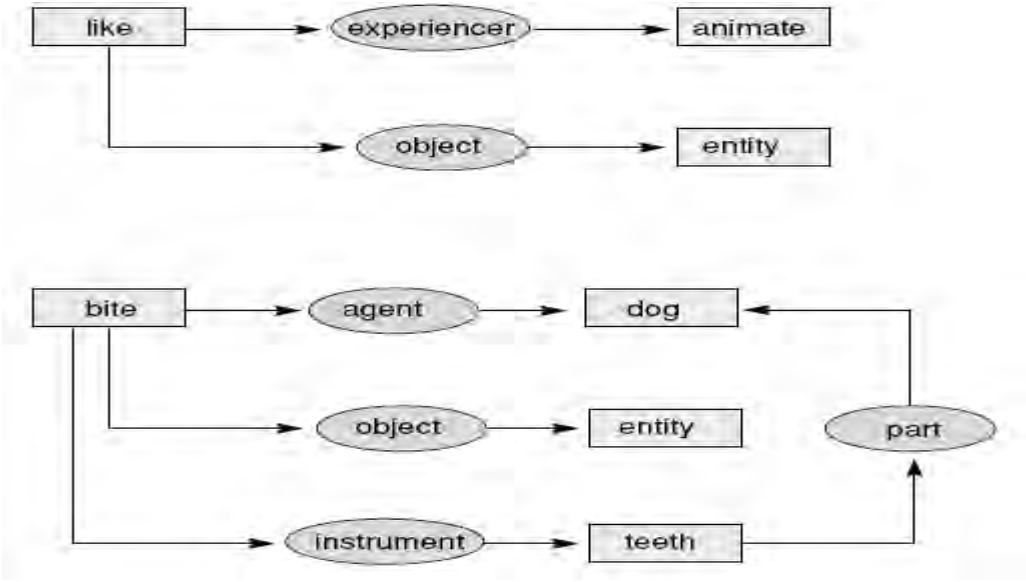

Figure 8.3. Case frames for the verbs "like" and "bite."

These verb-based case frames are also easily built in Prolog. Each verb is paired with a list of the semantic relations assumed to be part of the verb. These may include agents, instruments, and objects. We next offer examples of the verbs give and bite from Figure 8.3. For example, the verb give requires a subject, object, and indirect object. In the English sentence "John gives Mary the book," this structure takes on the obvious assignments. We can define defaults in a case frame by binding the appropriate variable values. For example, we could give bite a default instrument of teeth, and, indeed indicate that the instrument for biting, teeth, belong to the agent! Case frames for these two verbs might be:

```
verb(give,
       [human (Subject),
              agent (Subject, give),
              act_of_giving (give),
              object (Object, give),
              inanimate (Object),
              recipient (Ind_obj, give),
              human (Ind_obj) ] ).
verb(bite,
       [animate (Subject),
              agent (Subject, Action),
              act_of_biting (Action),
              object (Object, Action),
              animate (Object),
              instrument (teeth, Action),
              part_of (teeth, Subject) ] ).
```

Logic programming also offers a powerful medium for building grammars as well as representations for semantic meanings. We next build recursive descent parsers in Prolog, and then add syntactic and semantic constraints to these parsers.

8.3 A Context-Free Parser in Prolog

Consider the subset of English grammar rules below. These rules are "declarative" in the sense that they simply define relationships among parts of speech. With this subset of rules a large number of simple sentences can be judged as well formed or not. The "`<->`" indicate that the symbol on the left hand side can be replaced by the symbol or symbols on the right. For example, a `Sentence` can be replaced by a `NounPhrase` followed by a `VerbPhrase`.

```
Sentence <-> NounPhrase VerbPhrase
NounPhrase <-> Noun
NounPhrase <-> Article Noun
VerbPhrase <-> Verb
VerbPhrase <-> Verb NounPhrase
```

Adding some vocabulary to the grammar rules:

```
Article(a)
Article(the)
Noun(man)
Noun(dog)
Verb(likes)
Verb(bites)
```

These grammar rules have a natural fit to Prolog, for example, a `sentence` is a `nounphrase` followed by a `verbphrase`:

```
sentence(Start, End) :-
     nounphrase(Start, Rest), verbphrase(Rest, End).
```

This `sentence` Prolog rule takes two parameters, each a list; the first list, `Start`, is a sequence of words. The rule attempts to determine whether some initial part of this list is a `NounPhrase`. Any remaining tail of the `NounPhrase` list will match the second parameter and be passed to the first parameter of the `verbphrase` predicate. Any symbols that remain after the `verbphrase` check are passed back as the second argument of `sentence`. If the original list is a `sentence`, the second argument of `sentence` must be empty, `[]`.Two alternative Prolog descriptions of `nounphrase` and `verbphrase` parses follow.

Figure 8.4 is the parse tree of "the man bites the dog," with `and` constraints in the grammar reflected by `and` links in the tree.

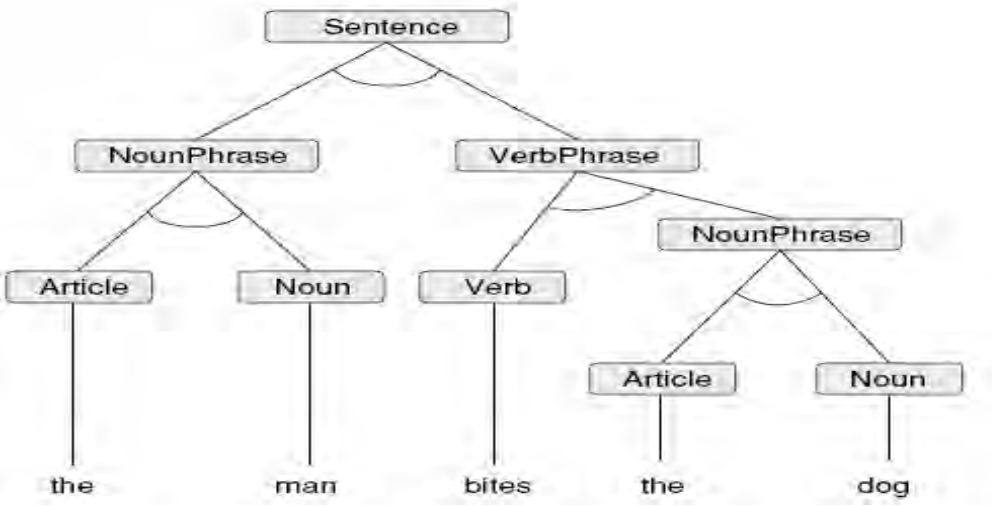

Figure 8.4. The and/or parse tree for "The man bites the dog."

To simplify our Prolog code, we present the sentence as a list: `[the, man, likes, the, dog]`. This list is broken up by, and passed to, the various grammar rules to be examined for syntactic correctness. Note how the "pattern matching" works on the list in question: pulling off the head, or the head and second element; passing on what is left over; and so on. The `utterance` predicate takes the list to be parsed as its argument and calls the `sentence` rule, initializing the second parameter of sentence to `[]`. The complete grammar is defined:

```
utterance(X) :- sentence(X, [ ]).
sentence(Start, End) :-
     nounphrase(Start, Rest), verbphrase(Rest, End).
nounphrase([Noun | End], End) :-
     noun(Noun).
nounphrase([Article, Noun | End], End) :-
     article(Article), noun(Noun).
verbphrase([Verb | End], End) :-
     verb(Verb).
```

```
verbphrase([Verb | Rest], End) :-
     verb(Verb), nounphrase(Rest, End).
article(a).
article(the).
noun(man).
noun(dog).
verb(likes).
verb(bites).
```

Example sentences may be tested for correctness:

```
?- utterance([the, man, bites, the, dog]).
Yes
?- utterance([the, man, bites, the]).
no
```

The interpreter can also fill in possible legitimate words to incomplete sentences:

```
?- utterance([the, man, likes, X]).
X = man
;
X = dog
;
no
```

Finally, the same code may be used to generate the set of all well-formed sentences using this limited dictionary and set of grammar rules:

```
?- utterance(X).
[man, likes]
;
[man, bites]
;
[man, likes, man]
;
[man, likes, dog]
etc.
```

If the user continues asking for more solutions, eventually all possible well-formed sentences that can be generated from the grammar rules and our vocabulary are returned as values for X. Note that the Prolog search is left-to-right and depth-first.

The grammar rules specify a subset of legitimate sentences of English. The Prolog grammar code represents these specifications. The interpreter is asked questions about them and the answer is a function of the specifications and the question asked. Since there are no constraints enforced across the subtrees that make up the full parse of a sentence, see Figure 8.4, the parser/generator for this grammar is said to be *context free*. In Section 8.3 we use probabilistic measures to add constraints both to particular word combinations and to the structures of the grammar.

8.4 Probabilistic Parsers in Prolog

In this section we extend the context-free grammar of Section 8.2 to include further syntactic and semantic constraints. For example, we may want some grammatical structures to be less likely than others, such as a noun by itself being less likely than an article followed by a noun. Further, we may want the sentence "The dog bites the widget" to be less likely than the sentence "The dog bites the man." Finally, if our vocabulary includes the verb like (as well as likes), we want "The man likes the dog" to be acceptable, but "The man like the dog" to fail. The parsers for Sections 8.3.1 and 8.3.2 were suggested by Professor Mark Steedman of the University of Edinburgh and transformed to the syntax of this book by Dr. Monique Morin of the University of New Mexico.

We next create two probabilistic parsers in Prolog, first a context free parser and second, a lexicalized context free parser.

Probabilistic Context-Free Parser

Our first extension is to build a *probabilistic context-free parser*. To do this, we add a probabilistic parameter, `Prob`, to each grammar rule. Note that the probability that a sentence will be a noun phrase followed by a verb phrase is 1.0, while the probability that a noun phrase is simply a noun is less than the probability of it being an article followed by a noun. These probabilities are reflected in `pr` facts that are related to each grammar rule, `r1`, `r2`, …, `r5`.

The full probability of a particular sentence, `Prob`, however, is calculated by combining a number of probabilities: that of the rule itself together with the probabilities of each of its constituents. Thus, the full probability `Prob` of `r1` is a product of the probabilities that a particular noun phrase is combined with a particular verb phrase. Further, the probability for the third rule, `r3`, will be the product of that type noun phrase occurring (`r3`) times the probabilities of the particular article and noun that make up the noun phrase. These noun/article probabilities are given in the two argument dictionary "fact" predicates. These probabilities for particular words might be determined by sampling some corpus of collected sentences. In the examples that follow we simply made-up these probabilistic measures.

```
utterance(Prob, X) :- sentence(Prob, X, [ ]).
sentence(Prob, Start, End) :-
     nounphrase(P1, Start, Rest),
     verbphrase(P2, Rest, End),
     pr(r1, P), Prob is P*P1*P2.
nounphrase(Prob, [Noun | End], End) :-
     noun(P1, Noun), pr(r2, P), Prob is P*P1.
nounphrase(Prob, [Article, Noun | End], End) :-
     article(P1, Article), noun(P2,Noun), pr(r3, P),
     Prob is P*P1*P2.
verbphrase(Prob, [Verb | End], End) :-
     verb(P1, Verb), pr(r4, P), Prob is P*P1.
```

```
verbphrase(Prob, [Verb | Rest], End) :-
     verb(P1, Verb),
     nounphrase(P2, Rest, End), pr(r5, P),
     Prob is P*P1*P2.
pr(r1, 1.0).
pr(r2, 0.3).
pr(r3, 0.7).
pr(r4, 0.2).
pr(r5, 0.8).
article(0.25, a).
article(0.75, the).
noun(0.65, man).
noun(0.35, dog).
verb(0.9, likes).
verb(0.1, bites).
```

We now run several example sentences as well as offer general patterns of sentences, i.e., sentences beginning with specific patterns of words such as "The dog bites…" Finally, we ask for all possible sentences that can be generated under these constraints.

```
?- utterance(Prob, [the, man, likes, the, dog]).
Prob = 0.0451474
Yes
?- utterance(Prob, [bites, dog])
No
?- utterance(Prob, [the, man, dog]).
No
?- utterance(Prob, [the, dog, bites, X]).
Prob = 0.0028665
X = man
;
Prob = 0.0015435
X = dog
;
No
?- utterance(Prob, [the, dog, bites, XY]).
Prob = 0.0028665
X = man
Y = [ ]
;
Prob = 0.0015435
X = dog
Y = [ ]
;
```

```
Prob = 0.00167212
X = a
Y = [man] ;
etc.
?- utterance(Prob, X).
Prob = 0.0351
X = [man, likes]
;
Prob = 0.0039
X = [man, bites]
;
Prob = 0.027378
X = [man, likes, man]
;
Prob = 0.014742
X = [man, likes, dog]
```

etc.

A Probabilistic Lexicalized Context Free Parser

We next demonstrate a *probabilistic lexicalized context-free parser*. This is a much more constrained system in which the probabilities, besides giving measures for the various grammatical structures and individual words as in the previous section, also describe the possible combinations of words (thus, it is a probabilistic *lexicalized* parser). For example, we now measure the likelihood of both noun-verb and verb-object word combinations. Constraining noun-verb combinations gives us much of the power of the context-sensitive parsing that we see next in Section 8.4, where noun-verb agreement is enforced by the constraints across the subtrees of the parse.

utterances in the language by determining a probabilistic measure for their occurring. Thus, we can determine that a possible sentence fails for syntactic or semantic reasons by seeing that it produces a very low or zero probability measure, rather than by the interpreter simply saying "no."

In the following grammar we have hard coded the probabilities of various structure and word combinations. In a real system, lexical information could be better obtained by sampling appropriate corpora with noun-verb or verb-object *bigrams*. We discuss the *n-gram* approach to language analysis in Luger (2009, Section 15.4) where the probability of word combinations was described (two words—*bigrams*, three words—*trigrams*, etc.). These probabilities are usually determined by sampling over a large collection of sentences, called a *corpus*. The result was the ability to assess the likelihood of these word combinations, e.g., to determine the probability of the verb "bite" following the noun "dogs."

In the following examples the `Prob` value is made up of the probabilities of the particular sentence structure, the probabilities of the verb-noun and verb-object combinations, and the probabilities of individual words.

```
utterance(Prob, X) :-
     sentence(Prob, Verb, Noun, X, [ ]).
sentence(Prob, Verb, Noun, Start, End) :-
     nounphrase(P1, Noun, Start, Rest),
     verbphrase(P2, Verb, Rest, End),
     pr(r1, P),      % Probability of this structure
     pr([r1, Verb, Noun], PrDep),
          % Probability of this noun/verb combo
     pr(shead, Verb, Pshead),
          % Probability this verb heads the sentence
     Prob is Pshead*P*PrDep*P1*P2.
nounphrase(Prob, Noun, [Noun | End], End) :-
     noun(P1, Noun), pr(r2, P), Prob is P*P1.
nounphrase(Prob, Noun, [Article,Noun | End], End) :-
     article(P1, Article), noun(P2,Noun), pr(r3, P),
     pr([r3, Noun, Article], PrDep),
          % Probability of art/noun combo
     Prob is P*PrDep*P1*P2.
verbphrase(Prob, Verb, [Verb | End], End) :-
     verb(P1, Verb), pr(r4, P), Prob is P*P1.
verbphrase(Prob, Verb, [Verb,Object | Rest], End) :-
     verb(P1, Verb), nounphrase(P2, Object,
          Rest, End).
     pr([r5, Verb, Object], PrDep),
          % Probability of verb/object combo
     pr(r5, P), Prob is P*PrDep*P1*P2.
pr(r1, 1.0).
pr(r2, 0.3).
pr(r3, 0.7).
pr(r4, 0.2).
pr(r5, 0.8).
article(1.0, a).
article(1.0, the).
article(1.0, these).
noun(1.0, man).
noun(1.0, dogs).
verb(1.0, likes).
verb(1.0, bite).
pr(shead, likes, 0.5).
pr(shead, bite, 0.5).
pr([r1, likes, man], 1.0).
pr([r1, likes, dogs], 0.0).
pr([r1, bite, man], 0.0).
pr([r1, bite, dogs], 1.0).
```

```
pr([r3, man, a], 0.5).
pr([r3, man, the], 0.5).
pr([r3, man, these], 0.0).
pr([r3, dogs, a], 0.0).
pr([r3, dogs, the], 0.6).
pr([r3, dogs, these], 0.4).
pr([r5, likes, man], 0.2).
pr([r5, likes, dogs], 0.8).
pr([r5, bite, man], 0.8).
pr([r5, bite, dogs], 0.2).
```

The `Prob` measure gives the likelihood of the utterance; words that aren't sentences return `No`.

```
?- utterance(Prob, [a, man, likes, these, dogs]).
Prob = 0.03136
?- utterance(Prob, [a, man, likes, a, man]).
Prob = 0.0098
?- utterance(Prob, [a, man, likes, a, man]).
Prob = 0.0098
?- utterance(Prob, [the, dogs, likes, these, man]).
Prob = 0
?- utterance(Prob, [the, dogs]).
No
?- utterance(Prob, [the, dogs, X | Y])
Prob = 0
X = likes Y = []
;
Prob = 0.042
X = bite Y = []
;
Prob = 0
X = likes Y = [man]
;
Prob = 0.04032
X = bite Y = [man]
;
Prob = 0.01008
X = bite Y = [dogs]
;
Prob = 0.04704
X = bite Y = [a, man]
Etc
?- utterance(Prob, X).
```

```
Prob = 0.03
X = [man, likes]
;
Prob = 0
X = [man, bite]
;
Prob = 0.0072
X = [man, likes, man]
;
Prob = 0.0288
X = [man, likes, dogs]
;
Prob = 0.0084
X = [man, likes, a, man]
etc
```

We next enforce many of the same syntax/semantic relationships seen in this section by imposing constraints (context sensitivity) across the subtrees of the parse. Context sensitivity can be used to constrain subtrees to support relationships within a sentence such as article-noun and noun-verb number agreement.

8.5 A Context-Sensitive Parser in Prolog

A *context-sensitive* parser addresses the issues of the previous section in a different manner. Suppose we desire to have proper noun–verb agreement enforced by the grammar rules themselves. In the dictionary entry for each word its singular or plural form can be noted as such. Then in the grammar specifications for `nounphrase` and `verbphrase` a further parameter is used to signify the `Number` of each phrase. This enforces the constraint that a singular noun has to be associated with a singular verb. Similar constraints for article–noun combinations can also be enforced. The technique we are using is constraining sentence components by enforcing variable bindings across the subtrees of the parse of the sentence (note the and links in the parse tree of Figure 8.4).

Context sensitivity increases the power of a context-free grammar considerably. These additions are made by directly extending the Prolog code of Section 8.2:

```
utterance(X) :- sentence(X, [ ]).
sentence(Start, End) :-
     nounphrase(Start, Rest, Number),
     verbphrase(Rest, End, Number).
nounphrase([Noun | End], End, Number) :-
     noun(Noun, Number).
nounphrase([Article, Noun | End], End, Number) :-
     noun(Noun, Number), article(Article, Number).
```

```
verbphrase([Verb | End], End, Number) :-
     verb(Verb, Number).
verbphrase([Verb | Rest], End, Number) :-
     verb(Verb, Number), nounphrase(Rest, End, _).
article(a, singular).
article(these, plural).
article(the, singular).
article(the, plural).
noun(man, singular).
noun(men, plural).
noun(dog, singular).
noun(dogs, plural).
verb(likes, singular).
verb(like, plural).
verb(bites, singular).
verb(bite, plural).
```

We next test some sentences. The answer to the second query is `no`, because the subject (`men`) and the verb (`likes`) do not agree in number.

```
?- utterance([the, men, like, the, dog]).
Yes
?- utterance([the, men, likes, the, dog]).
no
```

If we enter the following goal, `X` returns all verb phrases that complete the plural "`the men` …" with all verb phrases with noun–verb number agreement. The final query returns all sentences with article–noun as well as noun–verb agreement.

```
?- utterance([the, men  X]).
?- utterance(X).
```

In the context-sensitive example we use the parameters of dictionary entries to introduce more information on the meanings of each of the words that make up the sentence. This approach may be generalized to a powerful parser for natural language. More and more information may be included in the dictionary of the word components used in the sentences, implementing a knowledge base of the meaning of English words. For example, men are animate and human. Similarly, dogs may be described as animate and nonhuman. With these descriptions new rules may be added for parsing, such as "humans do not bite animate nonhumans" to eliminate sentences such as [the, man, bites, the, dog]. We add these constraints in the following section.

8.6 A Recursive Descent Semantic Net Parser

We next extend the set of context-sensitive grammar rules to include some possibilities of semantic consistency. We do this by matching case frames,

Section 8.1, for the verbs of sentences to semantic descriptions of subjects and objects. After each match, we constrain these semantic net subgraphs to be consistent with each other. We do this by performing graph operations, such as `join` and `restrict`, to each piece of the graph as it is returned up the parse tree.

We first present the grammar rules where the top-level `utterance`, returns not just a sentence but also a `Sentence_graph`. Each component of the grammar, e.g., `nounphrase` and `verbphrase`, call `join` to merge together the constraints of their respective graphs.

```
utterance(X, Sentence_graph) :-
     sentence(X, [ ], Sentence_graph).
sentence(Start, End, Sentence_graph) :-
     nounphrase(Start, Rest, Subject_graph),
     verbphrase(Rest, End, Predicate_graph),
     join([agent(Subject_graph)], Predicate_graph,
          Sentence_graph).
nounphrase([Noun | End], End, Noun_phrase_graph) :-
     noun(Noun, Noun_phrase_graph).
nounphrase([Article, Noun | End], End,
          Noun_phrase_graph) :-
     article(Article),
     noun(Noun, Noun_phrase_graph).
verbphrase([Verb | End], End, Verb_phrase_graph) :-
     verb(Verb, Verb_phrase_graph).
verbphrase([Verb | Rest], End, Verb_phrase_graph) :-
     verb(Verb, Verb_graph),
     nounphrase(Rest, End, Noun_phrase_graph),
     join([object(Noun_phrase_graph)], Verb_graph,
          Verb_phrase_graph).
```

We next present the graph `join` and `restriction` operations. These are meta-predicates since their domain is other Prolog structures. These utilities propagate constraints across pieces of semantic nets they combine.

```
join(X, X, X).
join(A, B, C) :-
     isframe(A), isframe(B), !,
     join_frames(A, B, C, not_joined).
join(A, B, C) :-
     isframe(A), is_slot((B), !,
     join_slot_to_frame(B, A, C).
join(A, B, C) :-
     isframe(B), is_slot(A), !,
     join_slot_to_frame(A, B, C).
join(A, B, C) :-
     is_slot(A), is_slot(B), !,
     join_slots(A, B, C).
```

`join_frames` recursively matches each slot (property) of the first frame to matching slots of the second frame. `join_slot_to_frame` takes a slot and a frame and searches the frame for matching slots. `join_slots,` once slots are matched, unites the two slots, taking the type hierarchy into account:

```
join_frames([A | B], C, D, OK) :-
     join_slot_to_frame(A, C, E) , !,
     join_frames(B, E, D, ok).
join_frames([ A | B], C, [A | D], OK) :-
     join_frames(B, C, D, OK), !.
join_frames([], A, A, ok).
join_slot_to_frame(A, [B | C], [D | C]) :-
     join_slots(A, B, D).
join_slot_to_frame(A, [B | C], [B | D]) :-
     join_slot_to_frame(A, C, D).
join_slots(A, B, D) :-
     functor(A, FA, _), functor(B, FB, _),
     match_with_inheritance(FA, FB, FN),
     arg(1, A, Value_a), arg(1, B, Value_b),
     join(Value_a, Value_b, New_value),
     D =.. [FN | [New_value]].
isframe([_ | _]).
isframe([ ]).
is_slot(A) :- functor(A, _, 1).
```

Finally, we create dictionary entries, the inheritance hierarchy, and verb case frames. In this example, we use a simple hierarchy that lists all valid specializations; the third argument to `match_with_inheritance` is the common specialization of the first two. A more realistic approach might maintain a graph of the hierarchies and search it for common specializations. Implementation of this is left as an exercise.

```
match_with_inheritance(X, X, X).
match_with_inheritance(dog, animate, dog).
match_with_inheritance(animate, dog, dog).
match_with_inheritance(man, animate, man).
match_with_inheritance(animate, man, man).
article(a).
article(the).
noun(fido, [dog(fido)]).
noun(man, [man(X)]).
noun(dog, [dog(X)]).
verb(likes, [action([liking(X)]),
         agent([animate(X)]), object(animate(Y)])]).
verb(bites, [action([biting(Y)]),
         agent([dog(X)]), object(animate(Z)])]).
```

We now parse several sentences and print out their `Sentence_graph`:

```
?- utterance([the, man, likes, the, dog], X).
X = [action([liking(_54)]), agent([man(_23)]),
        object([dog(_52)])].
?- utterance([fido, likes, the, man], X).
X = [action([liking(_62)]), agent([dog(fido)]),
        object([man(_70)])].
?- utterance([the, man, bites, fido], Z).
no
```

The first sentence states that some man, with name unknown, likes an unnamed dog. The last sentence, although it was syntactically correct, did not meet the semantic constraints, where a dog had to be the agent of bites. In the second sentence, a particular dog, Fido, likes an unnamed man. Next we ask whether Fido can bite an unnamed man:

```
?- utterance([fido, bites, the, man], X).
X = [action([biting(_12)]), agent([dog(fido)]),
        object([man(_17)])].
```

This parser may be extended in many interesting directions, for instance, by adding adjectives, adverbs, and prepositional phrases, or by allowing compound sentences. These additions must be both matched and constrained as they are merged into the sentence graph for the full sentence. Each dictionary item may also have multiple meanings that are only accepted as they meet the general requirements of the sentence. In the next chapter we present the Earley parser for language structures.

Exercises

1. Create a predicate calculus and a Prolog representation for the Conceptual Graph presented in Figure 8.2, "Mary gave John the book." Take this same example and create a general Prolog rule, "`X` gave `Y` the `Z`" along with a number of constraints, such as "`object(Z)`." Also create a number of Prolog facts, such as "`object(book)`" and show how this conceptual graph can be constrained by using the Prolog interpreter on your simple program.

2. Figure 8.3 presents case frames for the verbs `like` and `bite`. Write Prolog specifications that captures the constraints of these representations. Add other related fact and rules in Prolog and then use the Prolog interpreter to instantiate the constraints that are implicit in these two verb case frames.

3. Create a predicate calculus and a Prolog representation for the two Conceptual Graphs presented in Figure 8.5.

4. Describe an algorithm that could be used to impose graph constraints across the structures of Figure 8.5. You will have to address the nesting issue to handle sentences like "Mary believes that John does not like soup."

5. Create Prolog case frames, similar to those of Section 8.1 for five other verbs, including like, trade, and pardon.

6. Write the Prolog code for a subset of English grammar rules, as in the context-free and context-sensitive parsers in Sections 8.2 and 8.4, adding:

Adjectives and adverbs that modify verbs and nouns, respectively.

Prepositional phrases. (Can you do this with a recursive call?)

Compound sentences (two sentences joined by a conjunction).

7. Extend the stochastic context-free parser of Section 8.3 to include probabilities for the new sentence structures of Exercise 8. Explore obtaining probabilities for these sentence structures from a treebank for natural language processing. Examples may be found on the www.

8. Add probabilities for more word pair relationships as in the lexicalized context-free parser of Section 8.3.2. Explore the possibility of obtaining the probabilistic *bigram* values for the noun–verb, verb–object, and other word pairs from actual *corpus linguistics*. These may be found on the www.

9. Many of the simple natural language parsers presented in Chapter 8 will accept grammatically correct sentences that may not have a commonsense meaning, such as "the man bites the dog." These sentences may be eliminated from the grammar by augmenting the parser to include some notion of what is semantically plausible. Design a small "semantic network" (Section 2.4.1) in Prolog to allow you to reason about some aspect of the possible interpretations of the English grammar rules, such as when it is reasonable for the man to bite a dog.

10. Rework the semantic net parser of Section 14.3.2 to support richer class hierarchies. Specifically, rewrite `match_with_inheritance` so that instead of enumerating the common specializations of two items, it computes this by searching a type hierarchy.

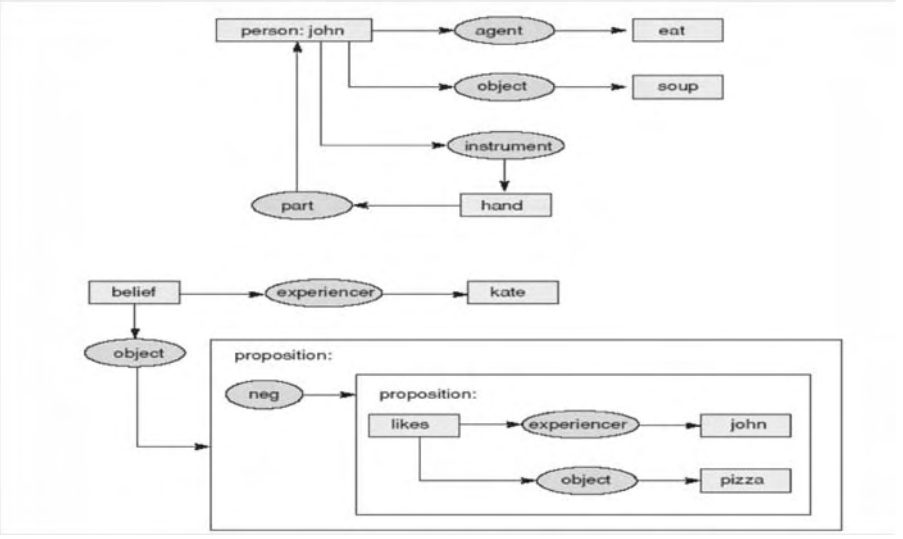

Figure 8.5. Conceptual Graphs to be translated into predicate calculus and into Prolog.

9 Dynamic Programming and the Earley Parser

Chapter Objectives

Language parsing with dynamic programming technique
Memoization of subparses
Retaining partial solutions (parses) for reuse
- The *chart* as medium for storage and reuse
  - Indexes for word list (sentence)
  - States reflect components of parse
  - Dot reflects extent parsing right side of grammar rule
- Lists of states make up components of chart
  - Chart linked to word list

Prolog implementation of an Earley parser
- Context free parser
  - Deterministic
- Chart supports multiple parse trees

Forwards development of chart composes components of successful parse
Backwards search of chart produces possible parses of word list
Earley parser important use of meta-interpreter technology.

Chapter Contents

9.1 Dynamic Programming Revisited

The dynamic programming (DP) approach to problem solving was originally proposed by Richard Bellman (1956). The idea is straightforward: when addressing a large complex problem that can be broken down into multiple subproblems, save partial solutions as they are generated so that they can be reused later in the solution process for the full problem. This "save and reuse of partial solutions" is sometimes called *memoizing* the subproblem solutions for later reuse.

There are many examples of dynamic programming in pattern matching technology, for example, it has been used in determining a difference measure between two strings of bits or characters. The overall difference between the strings will be a function of the differences between its specific components. An example of this is a spell checker going to its dictionary and suggesting words that are "close" to your misspelled word. The spell checker determines "closeness" by calculating a difference measure between your word and words that it has in its dictionary. This difference is often calculated, using some form of the DP algorithm, as a

function of the differences between the characters in each word. Examples of the DB comparison of character strings are found in Luger (2009, Section 4.1.2).

A further example of the use of DP is for recognizing words in speech understanding as a function of the possible phonemes from an input stream. As phonemes are recognized (with associated probabilities), the most appropriate word is often a function of the combined conjoined probabilistic measures of the individual phones. The DP Viterbi algorithm can be used for this task (Luger 2009, Section 13.1).

In this section, we present the Earley parser, a use of dynamic programming to build a context-free parser that recognizes strings of words as components of syntactically correct sentences. The presentation and Prolog code of this chapter is based on the efforts of University of New Mexico graduate student Stan Lee. The pseudo-code of Section 9.2 is adapted from that of Jurafsky and Martin (2008).

9.2 The Earley Parser

The parsing algorithms of Chapter 8 are based on a recursive, depth-first, and left-to-right search of possible acceptable syntactic structures. This search approach can mean that many of the possible acceptable partial parses of the first (left-most) components of the string are repeatedly regenerated. This revisiting of early partial solutions within the full parse structure is the result of later backtracking requirements of the search and can become exponentially expensive and costly in large parses. Dynamic programming provides an efficient alternative where partial parses, once generated, are saved for reuse in the overall final parse of a string of words. The first DP-based parser was created by Earley (1970).

Memoization And Dotted Pairs

In parsing with Earley's algorithm the memoization of partial solutions (partial parses) is done with a data structure called a *chart*. This is why the various alternative forms of the Earley approach to parsing are sometimes called *chart parsing*. The chart is generated through the use of *dotted grammar rules*.

The dotted grammar rule provides a representation that indicates, in the chart, the state of the parsing process at any given time. Every dotted rule falls into one of three categories, depending on whether the dot's position is at the beginning, somewhere in the middle, or at the end of the right hand side, RHS, of the grammar rule. We refer to these three categories as the *initial*, *partial,* or *completed* parsing stages, respectively:

```
Initial prediction:  Symbol  →  @  RHS_unseen
Partial parse:  Symbol  →  RHS_seen  @  RHS_unseen
Completed parse:  Symbol  →  RHS_seen  @
```

In addition, there is a natural correspondence between states containing different dotted rules and the edges of the parse tree(s) produced by the parse. Consider the following very simple grammar, where terminal symbols are surrounded by quotes, as in `"mary"`:

```
Sentence  →  Noun  Verb
Noun  →  "mary"
Verb  →  "runs"
```

As we perform a top-down, left-to-right parse of this sentence, the following sequence of states is produced:

`Sentence → • Noun Verb`		*predict: Noun followed by Verb*
`Noun → • mary`		*predict:* `mary`
`Noun → mary •`		*scanned:* `mary`
`Sentence → Noun • Verb`		*completed: Noun; predict: Verb*
`Verb → • runs`		*predict:* `runs`
`Verb → runs •`		*scanned:* `runs`
`Sentence → Noun Verb •`		*completed: Verb, completed: sentence*

Note that the *scanning* and *completing* procedures deterministically produce a result. The *prediction* procedure describes the possible parsing rules that can apply to the current situation. Scanning and prediction creates the states in the parse tree of Figure 9.1.

Earley's algorithm operates by generating top-down and left-to-right predictions of how to parse a given input. Each prediction is recorded as a *state* containing all the relevant information about the prediction, where the key component of each state is a dotted rule. (A second component will be introduced in the next section.) All of the predictions generated after examining a particular word of the input are collectively referred to as the *state set*. For a given input sentence with `n` words, w_1 to w_n, a total `n + 1` state sets are generated: [S_0, S_1, …, S_n]. The initial state set, S_0, contains those predictions that are made before examining any input words, S_1 contains predictions made after examining W_1, and so on.

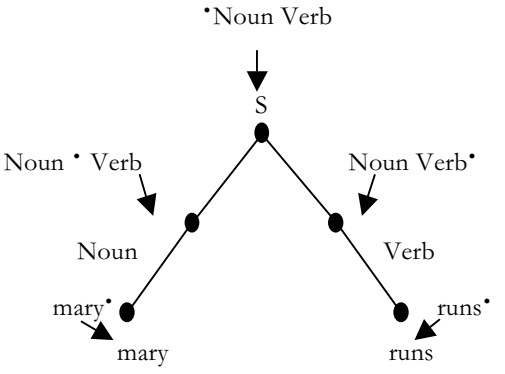

Figure 9.1 The relationship of dotted rules to the generation of a parse tree.

We refer to the entire collection of state sets as the *chart* produced by the parser. Figure 9.1 illustrates the relationship between state set generation and the examination of input words.

At this point we need to pause to get our terminology straight. Although, traditionally, the sets of states that make up each component of the parse are called *state sets*, the *order* of the generation of these states is important. Thus we call each component of the chart the *state list*, and describe it as `[State1, State2, …, Staten]`. This also works well with the Prolog implementation, Section 9.3, where the state lists will be maintained as Prolog lists. Finally, we describe each state of the state list as a sequence of specific symbols enclosed by brackets, for example, (`$ → • S`).

We now consider Earley's algorithm parsing the simple sentence `mary runs`, using the grammar above. The algorithm begins by creating a dummy start state, (`$ → • S`), that is the first member of state list S_0. This state represents the prediction that the input string can be parsed as a sentence, and it is inserted into S_0 prior to examining any input words. A successful parse produces a final state list S_n, which is S_2 in this example, that contains the state (`$ → S •`).

Beginning with S_0, the parser executes a loop in which each state, S_i, in the current state list is examined *in order* and used to generate new states. Each new state is generated by one of three procedures that are called the *predictor*, *scanner*, and *completer*. The appropriate procedure is determined by the dotted rule in state `S`, specifically by the grammar symbol (if any) following the dot in the rule.

In our example, the first state to be examined contains the rule (`$ → • S`). Since the dot is followed by the symbol `S`, this state is "expecting" to see an instance of `S` occur next in the input. As S is a nonterminal symbol of the grammar, the predictor procedure generates all states corresponding to a possible parse of `S`. In this case, as there is only one alternative for `S`, namely that `S → Noun Verb`, only one state, (`S → • Noun Verb`), is added to S_0. As this state is expecting a part of speech, denoted by the nonterminal symbol `Noun` following the dot, the algorithm examines the next input word to verify that prediction. This is done by the scanner procedure, and since the next word matches the prediction, `mary` is indeed a `Noun`, the scanner generates a new state recording the match: (`Noun → mary •`). Since this state depends on input word W_1, it becomes the first state in state list S_1 rather than being added to S_0. At this point the chart, containing two state lists, looks as follows, where after each state we name the procedure that generated it:

S_0:	[($ → • S),	dummy start state
	(S → • Noun Verb)]	predictor
S_1:	[(Noun → mary •)]	scanner

Each state in the list of states S_0 has now been processed, so the algorithm moves to S_1 and considers the state (`Noun → mary •`). Since this is a completed state, the completer procedure is applied. For each state expecting a `Noun`, that is, has the `• Noun` pattern, the completer

generates a new state that records the discovery of a **Noun** by advancing the dot over the **Noun** symbol. In this case, the completer produces the state (**S → • Noun Verb)** in S_0 and generates the new state (**S → Noun • Verb)** in the list S_1. This state is expecting a part of speech, which causes the scanner to examine the next input word W_2. As W_2 is a **Verb**, the Scanner generates the state (**Verb → runs •)** and adds it to S_2, resulting in the following chart:

S_0:	[($ → • S),	start
	(S → • Noun Verb)]	predictor
S_1:	[(Noun → mary •),	scanner
	(S → Noun • Verb)]	completer
S_2:	[(Verb → runs •)]	scanner

Processing the new state S_2, the completer advances the dot in (**S → Noun • Verb)** to produce (**S → Noun Verb •)**, from which the completer generates the state (**$ → S •)** signifying a successful parse of a sentence. The final chart for **mary runs**, with three state lists, is:

S_0:	[($ → • S),	start
	(S → • Noun Verb)]	predictor
S_1:	[(Noun → mary •),	scanner
	(S → Noun • Verb)]	completer
S_2:	[(Verb → runs •),	scanner
	(S → Noun Verb •),	completer
	($ → S •)]	completer

Earley Pseudocode

To represent computationally the state lists produced by the dotted pair rules above, we create indices to show how much of the right hand side of a grammar rule has been parsed. We first describe this representation and then offer pseudo-code for implementing it within the Earley algorithm. Each state in the state list is augmented with an index indicating how far the input stream has been processed. Thus, we extend each state description to a (*dotted rule [i, j]*) representation where the *[i, j]* pair denotes how much of right hand side, RHS, of the grammar rule has been seen or parsed to the present time. For the right hand side of a parsed rule that includes zero or more seen and unseen components indicated by the •, we have (**A → Seen • Unseen, [i,j]**), where **i** is the start of **Seen** and **j** is the position of • in the word sequence.

We now add indices to the parsing states discussed earlier for the sentence **mary runs**:

($ → • S, [0, 0])
produced by predictor, **i = j = 0,** nothing yet parsed

(Noun → mary •, [0,1])
scanner sees **word[1]** between word indices **0** and **1**

`(S → Noun • Verb, [0,1])`
completer has seen `Noun (mary)` between chart `0` and `1`

`(S → Noun Verb •, [0,2])`
completer has seen sentence `S` between chart `0` and `2`

Thus, the state indexing pattern shows the results produced by each of the three state generators using the dotted rules along with the word index W_i.

To summarize, the three procedures for generating the states of the state list are: *predictor* generating states with index `[j, j]` going into `chart[j]`, *scanner* considering word W_{j+1} to generate states indexed by `[j, j+1]` into `chart[j+1]`, and *completer* operating on rules with index `[i, j]`, `i < j`, adding a state entry to `chart[j]`. Note that a state from the dotted-rule, `[i, j]` always goes into the state list `chart[j]`. Thus, the state lists include `chart[0], ..., chart[n]` for a sentence of `n` words.

Now that we have presented the indexing scheme for representing the chart, we give the pseudo-code for the Earley parser. In Section 9.2.3 we use this code to parse an example sentence and in Section 9.3 we implement this algorithm in Prolog. We replace the "•" symbol with "@" as this symbol will be used for the dot in the Prolog code of Section 9.3.

```
function EARLEY-PARSE(words, grammar) returns chart
     chart := empty
     ADDTOCHART((S → @ S, [0, 0]), chart[0])
          % dummy start state

     for i from 0 to LENGTH(words) do
          for each state in chart[i] do
               if rule_rhs(state) = … @ A …
                    and A is not a part of speech
               then PREDICTOR(state)
               else if rule_rhs(state) = … @ L …
                    % L is part of speech
                    then SCANNER(state)
                    else     OMPLETER(state)
                         % rule_rhs = RHS @
          end
     end

procedure PREDICTOR((A → … @ B …, [i,  j]))
    for each (B → RHS) in grammar do
          ADDTOCHART((B → @ RHS, [j,  j]), chart[j])
end

procedure SCANNER((A → … @ L …, [i,  j]))
    if (L → word[j]) is_in grammar
    then ADDTOCHART((L → word[j] @ , [j,  j + 1]),
         chart[j + 1])
end
```

```
procedure COMPLETER((B → …  @, [j, k]))
    for each (A → … @ B …, [i, j]) in chart[j] do
        ADDTOCHART((A → … B @ …, [i, k]), chart[k])
end
procedure ADDTOCHART(state, state-list)
    if state is not in state-list
    then ADDTOEND(state, state-lis
end
```

Earley Example

Our first example, the Earley parse of the sentence "Mary runs," was intended to be simple but illustrative, with the detailed presentation of the state lists and their indices. We now produce a solution, along with the details of the chart that is generated, for a more complex sentence, "John called Mary from Denver". This sentence is ambiguous (Did John use a phone while he was in Denver to call, or did John call that Mary that was from Denver). We present the two different parses of this sentence in Figure 9.2 and describe how they may both be recovered from the chart produced by parsing the sentence in an exercise. This retrieval process is typical of the dynamic programming paradigm where the parsing is done in a *forward* left-to-right fashion and then particular parses are retrieved from the chart by moving *backward* through the completed chart.

The following set of grammar rules is sufficient for parsing the sentence:

```
S  →  NP VP
NP  →  NP PP
NP  →  Noun
VP  →  Verb NP
VP  →  VP PP
PP  →  Prep NP
Noun  →  "john"
Noun  →  "mary"
Noun  →  "denver"
Verb  →  "called"
Prep  →  "from"
```

In Figure 9.2 we present two parse trees for the word string `john called mary from denver`. Figure 9.2a shows `john called (mary from denver)`, where Mary is from Denver, and in Figure 9.2b `john (called mary) (from denver)`, where John is calling from Denver. We now use the pseudo-code of the function EARLEY-PARSE to address this string. It is essential that the algorithm not allow any state to be placed in any state list more than one time, although the same state may be generated with different predictor/scanner applications:

1. Insert start state ($ → @ S, [0,0]) into chart[0]
2. Processing state-list S_0 = chart[0] for (i = 0):
   The *predictor* procedure produces within chart[0]:

```
(S → @ NP VP, [0,0]) ==>
      (NP → @ NP PP, [0,0])

(S → @ NP VP, [0,0]) ==>
      (NP → @ Noun, [0,0])
```

3. Verifying that the next word word[*i* + 1] = word[1] or "john" is a Noun:

   The *scanner* procedure initializes chart[1] by producing

```
(NP → @ Noun, [0,0]) ==>
      (Noun → john @, [0,1])
```

4. Processing S_1 = chart[1] shows the typical start of a new state list, with the *scanner* procedure processing the next word in the word string, and the algorithm then calling *completer*.

   The *completer* procedure adds the following states to chart[1]:

```
(NP → Noun @, [0,1])
(S → NP @ VP, [0,1])              from x1
(NP → NP @ PP, [0,1])             from x2
```

5. The completer procedure ends for S_1 as no more states have "dots" to advance, calling *predictor*:

   The *predictor* procedure generates states based on all newly-advanced dots:

```
(VP → @ Verb NP, [1,1])           from x1
(VP → @ VP PP, [1,1])        also from x1
(PP → @ Prep NP, [1,1])           from x2
```

6. Verifying that the next word, word[*i* + 1] = word[2] or "called" is a Verb:

   The *scanner* procedure initializes chart[2] by producing:

```
(VP → @ Verb NP, [1,1]) ==>
      (Verb → called @, [1,2])
```

Step 6 (above) initializes `chart[2]` by scanning `word[2]` in the word string; the *completer* and *predictor* procedures then finish state list 2.

The function EARLEY-PARSE continues through the generation of `chart[5]` as seen in the full chart listing produced next. In the full listing we have annotated each state by the procedure that generated it. It should also be noted that several partial parses, indicated by *, are generated for the chart that are not used in the final parses of the sentence. Note also that the fifth and sixth states in the state list of `chart[1]`, indicated by **, which predict two different types of VP beginning at index 1, are instrumental in producing the two different parses of the string of words, as presented in Figure 9.2.

```
chart[0]:
   [($ → @ S, [0,0])                start state
   (S → @ NP VP, [0,0])             predictor
   (NP → @ NP PP, [0,0])*           predictor
   (NP → @ Noun, [0,0])]            predictor

chart[1]:
   [(Noun → john @, [0,1])          scanner
   (NP → Noun @, [0,1])             completer
   (S → NP @ VP, [0,1])             completer
   (NP → NP @ PP, [0,1])*           completer
   (VP → @ Verb NP, [1,1])**        predictor
   (VP → @ VP PP, [1,1])**          predictor
   (PP → @ Prep NP, [1,1])*]        predictor

chart[2]:
   [(Verb → called @, [1,2])        scanner
   (VP → Verb @ NP, [1,2])          completer
   (NP → @ NP PP, [2,2])            predictor
   (NP → @ Noun, [2,2])[            predictor

chart[3]:
   [(Noun → mary @, [2,3])          scanner
   (NP → Noun @, [2,3])             completer
   (VP → Verb NP @, [1,3])          completer
   (NP → NP @ PP, [2,3])            completer
   (S → NP VP @, [0,3])*            completer
   (VP → VP @ PP, [1,3])            completer
   (PP → @ Prep NP, [3,3])          predictor
   ($ → S @, [0,3])*]               completer

chart[4]:
   [(Prep → from @, [3,4])          scanner
   (PP → Prep @ NP, [3,4])          completer
   (NP → @ NP PP, [4,4])*           predictor
   (NP → @ Noun, [4,4])]            predictor

chart[5]:
   [(Noun → denver @, [4,5])        scanner
   (NP → Noun @, [4,5])             completer
   (PP → Prep NP @, [3,5])          completer
   (NP → NP @ PP, [4,5])*           completer
   (NP → NP PP @, [2,5])            completer
   (VP → VP PP @, [1,5])            completer
   (PP → @ Prep NP, [5,5])*         predictor
   (VP → Verb NP @, [1,5])          completer
   (NP → NP @ PP, [2,5])*           completer
   (S → NP VP @, [0,5])             completer
   (VP → VP @ PP, [1,5])*           completer
   ($ → S @, [0,5])]                completer
```

The complete chart generated by the EARLEY-PARSE algorithm contains 39 states separated into six different state lists, charts 0 – 5. The final state list contains the success state `($ → S @, [0,5])` showing that the string containing five words has been parsed and is indeed a sentence. As pointed out earlier, there are states in the chart, ten indicated by *, that are not part of either of the final parse trees, as seen in Figure 9.2.

9.3 The Earley Parser in Prolog

Finally, we present the Earley parser in Prolog. Our Prolog code, designed by Stan Lee, a graduate student in Computer Science at the University of New Mexico, is a direct implementation of the EARLEY-PARSE pseudo-code given in Section 9.2.2. When looking at the three procedures that follow - scanner, predictor, and completer – it is important to note that similarity.

The code begins with initialization, including reading in the word string, parsing, and writing out the chart after it is created:

```
go :- go(s).
go(NT) :
     input(Words),
     earley(NT, Words, Chart),
     writesln(Chart).
```

The `earley` predicate first generates the start state, `StartS` for the parser and then calls the state generator `state_gen` which produces the chart, `Chart`. `state_gen` checks first if the wordlist is exhausted and terminates if it is, next it checks if the current state list is completed and if it is, begins working on the next state, otherwise it continues to process the current state list:

```
earley(NonTerminal, Words, Chart) :-
     StartS = s($, [@, NonTerminal], [0,0]),
     initial_parser_state(Words, StartS, PS),
     state_gen(PS, Chart).
state_gen(PS, Chart) :-          %Si = [], Words = []
     final_state_set_done(PS, Chart)
state_gen(PS, Chart) :-        %Si = [], Words not []
     current_state_set_done(PS, NextPS),
     state_gen(NextPS, Chart).
state_gen(PS, Chart) :-                %Si = [S|Rest]
     current_state_rhs(S, RHS, PS, PS2),
                                      %PS2[Si] = Rest
     (
     append(_, [@, A|_], RHS),
     rule(A, _) ->            %A not a part of speech
     predictor(S, A, PS2, NextPS)
     ;
     append(_, [@, L|_], RHS),
     lex_rule(L, _) ->          %L is part of speech
     scanner(S, L, PS2, NextPS)
     ;
```

```
completer(S, PS2, NextPS) %S is completed state
),
state_gen(NextPS, Chart).
```

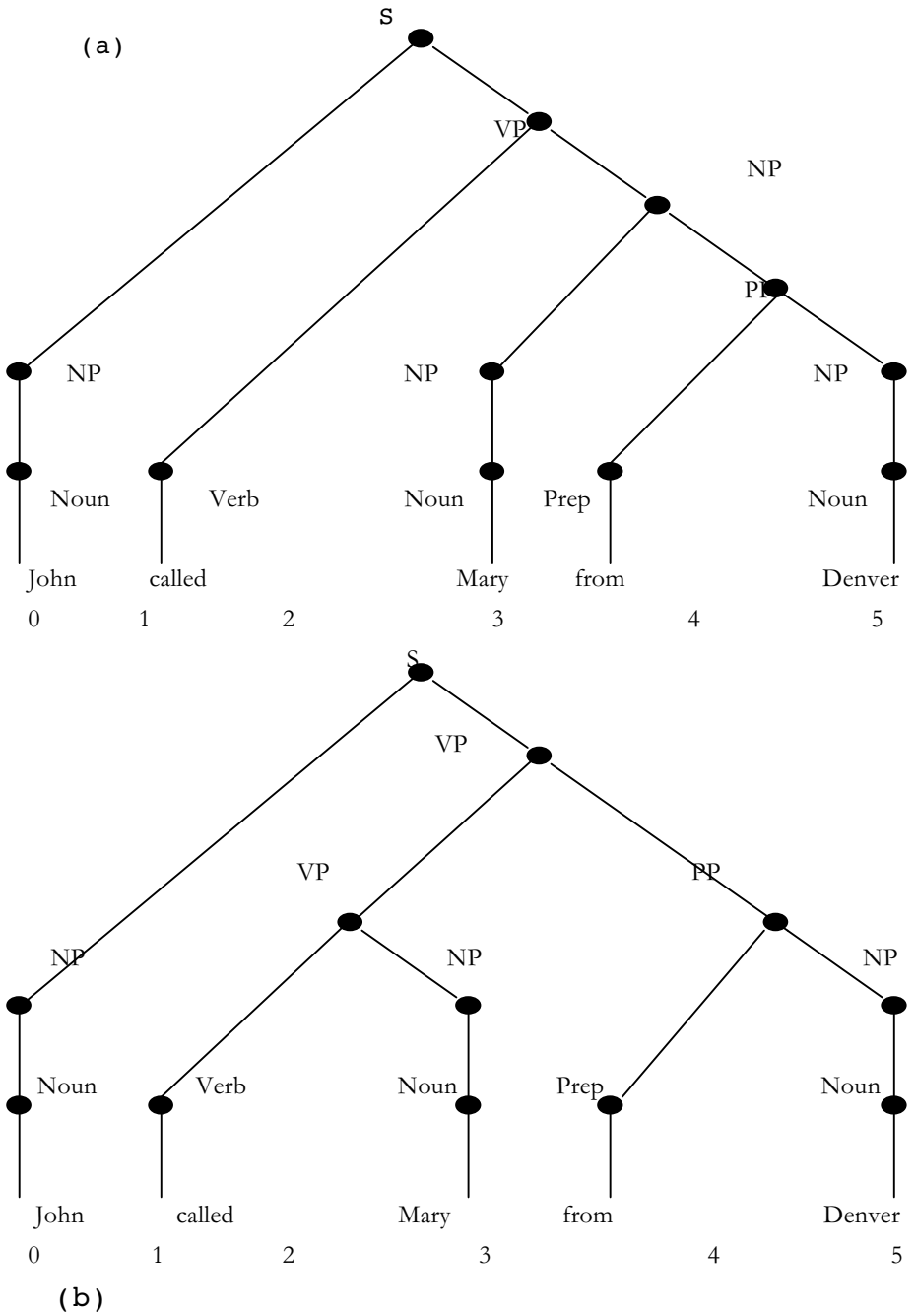

Figure 9.2. Two different parse trees for the word string representing the sentence "John called Mary from Denver". The index scheme for the word string is below it.

We next present the `predictor` procedure. This procedure takes a dotted rule `A -> ... @ B ...` and predicts a new entry into the state list for the symbol `B` in the grammar:

```
predictor(S, B, PS, NewPS) :-
     S = s(_, _, [I,J]),
     Findall
          (
          s(B, [@ | RHS], [J,J]),
          rule(B, RHS),
          NewStates
          ),
     add_to_chart(NewStates, PS, NewPS).
```

The `scanner` procedure considers the next word in the input string. If it is a part of speech, `Lex`, `scanner` creates a new state list and enters that part of speech, for example, the state (`Noun → denver @, [4,5]`), that begins `chart[5]` in Section 9.2.3. The `scanner` procedure prepares the way for the `completer` and `predictor` procedures. If the next word in the input stream is not the predicted part of speech, it leaves the chart unchanged:

```
scanner(S, Lex, PS, NewPS) :-
     S = s(_, _, [I,J]),
     next_input(Word, J, J1, PS),
     lex_rule(Lex, [Word]), !,
     add_to_chart( [s(Lex, [Word,@], [J,J1])], PS,
          NewPS).
scanner(_, _, PS, PS).
```

Finally, the `completer` procedure takes a completed state `S` that has recognized a pattern `B`, and adds a new state to the list for each preceding state that is looking for that pattern.

```
completer(S, PS, NewPS) :-
     S = s(B, _, [J,K]),
     Findall
          (
          s(A, BdotRHS, [I,K]),
          (
          in_chart( s(A, DotBRHS, [I,J]), PS),
          append(X, [@, B|Y], DotBRHS),
          append(X, [B, @|Y], BdotRHS % adv dot over B
          ),
          NewStates
          ),
     add_to_chart(NewStates, PS, NewPS).
```

We next describe the utility predicates that support the three main procedures just presented. The most important of these are predicates for maintaining the state of the parser itself. The parser-state, `PS`, is represented by a structure `ps` with five arguments: `PS = ps(Words, I, Si, SNext, Chart)`. The first argument of `ps` is the current string of words maintained as a list and the second argument, `I`, is the current index of `Words`. `Si` and `SNext` are the current and next state

lists, and `Chart` is the current chart. Thus, `Si` and `SNext` are always subsets of the current `Chart`. Notice that the "assignment" that creates the next state-list is done with unification (`=`).

The `PS` utilities perform initial and final state checks, determine if the current state list is complete, extract components of the current state and get the next input value. Parsing is finished when the `Word` list and the current state list `Si`, the first and third arguments of `PS`, are both empty. If `Si` is empty but the `Word` list is not, then the next state list becomes the new current state list and the parser moves to the next index as is seen in the `current_state_set_done` predicate:

```
initial_parser_state(Words, StartState, InitPS) :-
     InitPS = ps(Words, 0, [StartState], [],
          [StartState]).
final_state_set_done( ps([], _, [], _, FinalChart),
          FinalChart).
current_state_set_done( ps([_|Words], I, [], SNext,
          Chart), ps(   Words,  J, SNext, [],
          Chart)) :-
     J is I+1.

current_state_rhs(S, RHS, ps(Words, I, [S|Si],
          SNext, Chart), ps(Words, I,    Si,  SNext,
          Chart)) :-
S = s(_, RHS, _).
```

In the final predicate, `S` is the first state of the current state list (the third argument of `ps`, maintained as a list). This is removed, and the patterns of the right hand side of the current dotted grammar rule, `RHS`, are isolated for interpretation. The current state list `Si` is the tail of the previous list.

More utilities: The next element of `Words` in `PS` is between the current and next indices. The chart is maintained by checking to see if states are already in its state lists. Finally, there are predicates for adding states to the current chart.

```
next_input(Word, I, I1, ps([Word|_], I, _, _, _)) :-
     I1 is I+1.
add_to_chart([], PS, PS).
add_to_chart([S|States], PS, NewPS) :-
     in_chart(S, PS),!,
     add_to_chart(States, PS, NewPS).
add_to_chart([S|States], PS, NewPS) :-
     add_to_state_set(S, PS, NextPS),
     add_to_chart(States, NextPS, NewPS).
in_chart(S, ps(_, _, _, _, Chart)) :-
     member(S, Chart).
```

```
add_to_state_set(S, PS, NewPS) :-
     PS = ps(Words, I, Si, SNext, Chart),
     S = s(_, _, [_,J]),
     add_to_end(S, Chart, NewChart),
     (
     I == J ->                                %S is not a scan
state
        add_to_end(S, Si, NewSi),
        NewPS = ps(Words, I, NewSi, SNext, NewChart)
     ;
        add_to_end(S, SNext, NewSNext),
        NewPS = ps(Words, I, Si, NewSNext, NewChart)
     ).
add_to_end(X, List, NewList) :-
     append(List, [X], NewList).
```

The `add_to_state_set` predicate, first places the new state in the new version of the chart, `NewChart`. It then checks whether the current word list index `I` is the same as the second index `J` of the pair of indices of the state being added to the state list, testing whether `I == J`. When this is true, that state is added to the end (made the last element) of the current state list S_i. Otherwise, the new state was generated by the `scanner` procedure after reading the next word in the `input` word list. This new state will begin a new state list, `SNext`.

Finally, we present the output of the Prolog `go` and `earley` predicates running on the word list "John called Mary from Denver":

```
?- listing([input, rule, lex_rule]).
input([john, called, mary, from, denver]).
rule(s, [np, vp]).
rule(np, [np, pp]).
rule(np, [noun]).
rule(vp, [verb, np]).
rule(vp, [vp, pp]).
rule(pp, [prep, np]).
lex_rule(noun, [john]).
lex_rule(noun, [mary]).
lex_rule(noun, [denver]).
lex_rule(verb, [called]).
lex_rule(prep, [from]).
?- go.
s($, [@, s], [0, 0])
s(s, [@, np, vp], [0, 0])
s(np, [@, np, pp], [0, 0])
s(np, [@, noun], [0, 0])
s(noun, [john, @], [0, 1])
s(np, [noun, @], [0, 1])
s(s, [np, @, vp], [0, 1])
s(np, [np, @, pp], [0, 1])
```

```
s(vp, [@, verb, np], [1, 1])
s(vp, [@, vp, pp], [1, 1])
s(pp, [@, prep, np], [1, 1])
s(verb, [called, @], [1, 2])
s(vp, [verb, @, np], [1, 2])
s(np, [@, np, pp], [2, 2])
s(np, [@, noun], [2, 2])
s(noun, [mary, @], [2, 3])
s(np, [noun, @], [2, 3])
s(vp, [verb, np, @], [1, 3])
s(np, [np, @, pp], [2, 3])
s(s, [np, vp, @], [0, 3])
s(vp, [vp, @, pp], [1, 3])
s(pp, [@, prep, np], [3, 3])
s($, [s, @], [0, 3])
s(prep, [from, @], [3, 4])
s(pp, [prep, @, np], [3, 4])
s(np, [@, np, pp], [4, 4])
s(np, [@, noun], [4, 4])
s(noun, [denver, @], [4, 5])
s(np, [noun, @], [4, 5])
s(pp, [prep, np, @], [3, 5])
s(np, [np, @, pp], [4, 5])
s(np, [np, pp, @], [2, 5])
s(vp, [vp, pp, @], [1, 5])
s(pp, [@, prep, np], [5, 5])
s(vp, [verb, np, @], [1, 5])
s(np, [np, @, pp], [2, 5])
s(s, [np, vp, @], [0, 5])
s(vp, [vp, @, pp], [1, 5])
s($, [s, @], [0, 5])
Yes
?-
```

We present the Earley parser again in Java, Chapter 30. Although the control procedures in Java are almost identical to those just presented in Prolog, it is interesting to compare the representational differences between declarative and an object-oriented languages.

Next, in the final chapter of Part I, we discuss important features of Prolog and declarative programming. We present Lisp and functional programming in Part III.

Exercises

1. Describe the role of the dot within the right hand side of the grammar rules as they are processed by the Earley parser. How is the location of the dot changed as the parse proceeds? What does it mean when we say that the same right hand side of a grammar rule can have dots at different locations?

2. In the Earley parser the input word list and the states in the state lists have indices that are related. Explain how the indices for the states of the state list are created.

3. Describe in your own words the roles of the `predictor`, `completer`, and `scanner` procedures in the algorithm for Earley

parsing. What order are these procedures called in when parsing a sentence, and why is that ordering important? Explain your answers to the order of procedure invocation in detail.

4. Augment the Earley Prolog parser to consider the sentence "John saw the burglar with the telescope". Create two different possible pare trees from interpreting this string and comment on how the different possible parses are retrieved them from the chart.

5. Create an 8 – 10 word sentence of your own and send it to the Earley parser. Produce the chart as it changes with each additional word of the sentence that is scanned.

6. Create a grammar that includes adjectives and adverbs in its list of rules. What changes are needed for the Earley parser to handle these new rules? Test the Early parser with sentences that contain adjectives and adverbs.

7. In the case of "John called Mary from Denver" the parser produced two parse trees. Analyze Figure 9.4 and show which components of the full parse are shared between both trees and where the critical differences are.

8. Analyze the complexity of the Earley algorithm. What was the cost of the two parses that we considered in detail in this chapter? What are the worst- and best-case complexity limits? What type sentences force the worst case? Alternatively, what types of sentences are optimal?

10 Prolog: Final Thoughts

Chapter Objectives

Prolog and declarative representations
- Facts
- Rules

The `append` example

Prolog referenced to automated reasoning systems
- Lack of *occurs check*
- No *unique names* or *closed world*

Prolog semantics
- Pattern-matching

Left-to-right depth-first search search

Backtracking on variable bindings

References offered for Prolog extensions

Chapter Contents

10.1 Prolog: Towards a Declarative Semantics

We have now finishing our nine-chapter presentation of Prolog. To summarize and conclude we describe again the design philosophy supporting this language paradigm, look at how this influenced the history of its development, summarize the main language idioms we used in building our AI applications programs, and mention several modern extensions of this declarative approach to programming.

Prolog was first designed and used at the University of Marseilles in the south of France in the early 1970s. The first Prolog interpreter was intended to analyze French using *metamorphosis grammars* (Colmerauer 1975). From Marseilles, the language development moved on to the University of Edinburgh in Scotland, where at the Artificial Intelligence Department, Fernando Pereira and David Warren (1980) created *definite clause grammars*. In fact, because of the declarative nature of Prolog and the flexibility of pattern-driven control, tasks in Natural Language Processing, NLP, (Luger 2009, Chapter 15) have always offered a major application domain (see Chapters 8 and 9). Veronica Dahl (1977), Dahl and McCord (1983), Michael McCord (1982, 1986), and John Sowa (Sowa 1984, Walker et al. 1987) have all contributed to this research.

Besides NLP, Prolog has supported many research tasks including the development of early expert systems (Bundy et al. 1979). Building AI representations such as semantic nets, frames, and objects has always been an important task for Prolog (see especially *Knowledge Systems and Prolog* by

Adrian Walker, Michael McCord, John Sowa, and Walter Wilson, 1987, and *Prolog: A Relational Language and Its Applications* by John Malpas 1987).

In the remainder of this chapter we discuss briefly declarative programming, how Prolog relates to theorem proving, and describe again the Prolog idioms presented in Part II.

In traditional computing languages such as FORTRAN, C, and Java the logic for the problem's specification and the control for executing the solution algorithm are inextricably mixed together. A program in these languages is simply a sequence of things *to be done* to achieve an answer. This is the accepted notion of *applicative* or *procedural* languages. Prolog, however, separates the logic or specification for a problem application from the execution or control of the use of that specification. In artificial intelligence programs, there are many reasons for this separation, as has been evident throughout Part II.

Prolog presents an alternative approach to computing. A program, as we have seen, consists of a set of specifications or declarations *of what is true* in a problem domain. The Prolog interpreter, taking a question from the user, determines whether it is true or false with respect to the set of specifications, If the query is true, Prolog will return a set of variable bindings (a model, see 10.2) under which the query is true.

As an example of the *declarative/nonprocedural* nature of Prolog, consider `append`:

```
append([ ], L, L).
append([X | T], L, [X | NL]) :- append(T, L, NL).
```

`append` is nonprocedural in that it defines a relationship between lists rather than a series of operations for joining two lists. Consequently, different queries will cause it to compute different aspects of this relationship. We can understand `append` by tracing its execution in joining two lists together. If the following call is made, the response is:

```
?- append([a, b, c], [d, e], Y).
Y = [a, b, c, d, e]
```

The execution of append is not tail recursive, in that the local variable values are accessed after the recursive call has succeeded. In this case, X is placed on the head of the list ([X | NL]) after the recursive call has finished. This requires that a record of each call be kept on the Prolog stack. For purposes of reference in the following trace:

```
1. is append([ ], L, L).
2. is append([X | T], L, [X | NL]) :-
          append(T, L, NL).

?- append([a, b, c], [d, e], Y).
   try match 1, fail [a, b, c] /= [ ]
   match 2, X is a, T is [b, c], L is [d, e],
        call append([b, c], [d, e], NL)
        try match 1, fail [b, c] /= [ ]
        match 2, X is b, T is [c], L is [d, e],
             call append([c], [d, e], NL)
```

```
                try match 1, fail [c]  [ ]
                match 2, X is c, T is [ ], L is [d, e],
                     call append([ ], [d, e], NL)
                    match 1, L is [d, e]
                   yes
                yes, N is [d, e], [X | NL] is [c, d, e]
   yes, NL is [c, d, e], [X | NL] is [b, c, d, e]
yes, NL is [b, c, d, e],
      [X | NL] is [a, b, c, d, e]
Y = [a, b, c, d, e], yes
```

In most Prolog programs, the parameters of the predicates seem to be intended as either "input" or "output"; most definitions assume that certain parameters be bound in the call and others unbound. This need not be so. In fact, there is no commitment at all to parameters being input or output! Prolog code is simply a set of specifications of what is true, a statement of the logic of the situation. Thus, `append` specifies a relationship between three lists, such that the third list is the catenation of the first onto the front of the second.

To demonstrate this we can give `append` a different set of goals:

```
?- append([a, b], [c], [a, b, c]).
Yes
?- append([a], [c], [a, b, c]).
No
?- append(X, [b, c], [a, b, c]).
X = [a]
?- append(X, Y, [a, b, c]).
X = [ ]
Y = [a, b, c]
;
X = [a]
Y = [b, c]
;
X = [a, b]
Y = [c]
;
X = [a, b, c]
Y = [ ]
;
no
```

In the last query, Prolog returns all the lists `X` and `Y` that, when appended together, give `[a,b,c]`, four pairs of lists in all. As mentioned above, `append` gives a statement of the logic of a relationship that exists among three lists. What the interpreter produces depends on the query.

The notion of solving a problem based on a set of specifications for relationships in a problem domain area, coupled with the action of a theorem prover, is exciting and important. As seen in Part II, it is a

valuable tool in areas as diverse as natural language understanding, databases, expert systems, and machine learning. How the Prolog interpreter works cannot be fully understood without the concepts of resolution theorem proving, especially the Horn clause refutation process, which is presented in Luger (2009, Section 14.2 and Section 14.3) where Prolog is presented as an instance of a resolution refutation system. In Section 10.2 we briefly summarize these issues.

10.2 Prolog and Automated Reasoning

Prolog's declarative semantics, with the interpreter determining the truth or falsity of queries has much of the feel of an automated reasoning system or *theorem prover* (Luger 2009, Chapter 14). In fact, Prolog is not a theorem prover, as it lacks several important features that would make it both *sound* (only producing mathematically correct responses) and *complete* (able to produce all correct responses consistent with a problem's specifications). Many of these features are not implemented in current versions of Prolog. In fact, most are omitted to make Prolog a more efficient programming tool, even when this omission costs Prolog any claim of mathematical soundness.

In this section we will list several of the key features of automated reasoning systems that Prolog lacks. First is the *occurs check*. Prolog does not determine whether any expression in the language contains a subset of itself. For example, the test whether `foo(X) = foo(foo(X))` will make most Prolog environments get seriously weird. It turns out that the systematic check of whether any Prolog expression contains a subset of itself is computationally costly and as a result is ignored.

A second limitation on Prolog is the order constraint. The Prolog inference system (interpreter) performs a left-to-right depth-first goal reduction on its specifications. This requires that the programmer order these specifications appropriately. For example, the termination of a recursive call must be presented before the recursive expression, otherwise the recursion will never terminate. The programmer can also organize goals in the order in which she wishes the interpreter to see them. This can help create an efficient program but does not support a truly declarative specification language where non-deterministic goal reduction is a critical component. Finally, the use of the cut, `!,` allows the programmer to further limit the set of models that the interpreter can compute. Again this limitation promotes efficiency but it is at the cost of a mathematically complete system.

A third limitation of Prolog is that there is no *unique name* constraint or *closed world* assumption. Unique names means that each atom in the prolog world must have one and only one "name" or value; otherwise there must exist a set of deterministic predicates that can reduce an atom to its unique (canonical) form. In mathematics, for example, 1, cannot be 1 + 0, 0 + 1, or 0 + 1 + 0, etc. There must be some predicate that can reduce all of these expressions to one canonical form for that atom.

Further, the closed world assumption, requires that all the atoms in a

domain must be specified; the interpreter cannot return `no` because some atom was ignored or misspelled. These requirements in a theorem proving environment address the *negation as failure* result that can be so frustrating to a Prolog programmer. Negation as failure describes the situation where the interpreter returns `no` and this indicates either that the query is false or that the program's specifications are incorrect. When a true theorem prover responds `no` then the query is false.

Even though the Prolog interpreter is not a theorem prover, the intelligent programmer can utilize many aspects of its declarative semantics to build a set of clean representational specifications; these are then addressed by an efficient interpreter. For more discussion of Prolog as theorem proving see Luger (2009, Section 14.3).

10.3 Prolog Idioms and Extensions

We now summarize several of the Prolog programming idioms we have used in Part II of this presentation. We consider idioms from three perspectives, from the lowest level of command and specification instructions, from a middle level of creating program language modules such as abstract data types, and from the most general level of using meta-predicates to make new interpreters within Prolog that are able to operate on specific sets of Prolog expressions.

From the lowest level of designing control and building specifications, we mention four different idioms that were used throughout Part II as critical components for constructing Prolog programs. The first idiom is *unification* or *pattern matching*. Unification offers a unique power for variable binding found only in high-level languages. It is particularly powerful for pattern matching across sets of predicate calculus specifications. Unification offers an efficient implementation of the `if/then/else` constructs of lower level languages: if the pattern matches, perform the associated action, otherwise consider the next pattern. It is also an important and simplifying tool for designing meta-interpreters, such as the production system (Section 4.2). Production rules can be ordered and presented as a set of patterns to be matched by unification) that will then trigger specific behaviors. An algorithm for unification can be found in Luger (2009, Section 2.3). It is interesting to note that unification, a constituent of Prolog, is explicitly programmed into Lisp (Chapter 15) and Java (Chapter 32 and 33) to support AI programming techniques in these languages.

A second idiom of Prolog is the use of *assignment*. Assignment is related to unification in that many variables, especially those in predicate calculus form, are assigned values through unification. However, when a variable is to have some value based on an explicit functional calculation, the `is` operator must be used. Understanding the specific roles of assignment, evaluation, and pattern matching is important for Prolog use.

The primary control construct for Prolog is *recursion*, the third idiom we mention. Recursion works with unification to evaluate patterns in much the same way as `for`, `repeat/until`, or `while` constructs are used in lower level languages. Since many of AIs problem solving tasks consist in

searching indeterminate sized trees or graphs, the naturalness of recursion makes it an important idiom: until specific criteria are met continue search over specifications. Of course the lower-level control constructs of `for`, `repeat`, etc., could be built into Prolog, but the idioms for these constructs is recursion coupled with unification.

Finally, at the predicate creation level of the program, the *ordering* of predicate specifications is important for Prolog. The issue is to utilize the built in depth-first left-to-right goal reduction of the Prolog interpreter. Understanding the action of the interpreter has important implications for using the order idiom. Along with order of specifications for efficient search, of course, is understanding and using wisely the predicate cut, !.

At the middle level of program design, where specifications are clustered to have systematic program level effects, we mention several idioms. These were grouped together in our presentation in Section 3.3 under the construct *abstract data types (ADTs)*. Abstract data types, such as *set*, *stack*, *queue*, and *priority queue* were developed in Chapter 3. The abstractions allow the program designer to use the higher-level constructs of queue, stack, etc. directly in problem solving. We then used these control abstract data types to design the search algorithms presented in Chapter 4. They were also later used in the machine learning and natural language chapters of Part II. For our Prolog chapters these idioms offer a natural way to express constructs related to graph search.

Finally, at that abstract level where the programmer is directly designing interpreters we described and used the *meta-predicate* idioms. Meta-predicates are built into the Prolog environment to assist the program designer with tools that manipulate other Prolog predicates, as described in Section 5.1. We demonstrated in Section 5.2 how the meta-predicate idioms can be used to enforce type constraints within a Prolog programming environment.

The most important use of meta-predicates, however, is to design meta-interpreters as we did in the remaining chapters (6 – 9) of Part II. Our meta-interpreters were collected sets of predicates that were used to interpret other sets of predicate specifications. Example meta-interpreters included a Prolog interpreter written in Prolog and a production system interpreter, Exshell, for building rule-based expert systems. The meta-interpreter is the most powerful use of our idioms, because at this level of abstraction and encapsulation our interpreters are implementing specific design patterns.

There are many additional software tools for declarative and logic programming available. An extension of Prolog's declarative semantics into a true resolution-based theorem-proving environment can be found in Otter (McCune and Wos 1997). Otter, originally produced at Argonne National Laboratories, is a complete automated reasoning system based on resolution refutation that addresses many of the shortcomings of Prolog mentioned in Section 10.2, e.g., the occurs check. A current version of Otter includes Isabelle, written in ML, (Paulson 1989), Tau (Halcomb and Schulz 2005), and Vampire (Robinson and Voronkov 2001). These automated reasoning systems are in the public domain and downloadable.

Ciao Prolog is a modern version of Prolog created in Spain (Mera et al. 2007, Hermenegildo et al. 2007). Ciao offers a complete Prolog system, but its novel modular design allows both restricting and extending the language. As a result, it allows working with fully declarative subsets of Prolog and also to extend these subsets both syntactically and semantically. Most importantly, these restrictions and extensions can be activated separately on each program module so that several extensions can coexist in the same application for different modules. Ciao also supports (through such extensions) programming with functions, higher-order (with predicate abstractions), constraints, and objects, as well as feature terms (records), persistence, several control rules (breadth-first search, iterative deepening), concurrency (threads/engines), a good base for distributed execution (agents), and parallel execution. Libraries also support WWW programming, sockets, external interfaces (C, Java, TclTk, relational databases, etc.).

Ehud Shapiro and his colleagues have researched the parallel execution of Prolog specifications. This is an important extension of the power to be gained by extending the built in depth-first search with backtracking traditional Prolog interpreter with parallel execution. For example, if a declarative goal has a number of or based goals to satisfy, these can be checked in parallel (Shapiro 1987).

Constraint logic programming is a declarative specification language where relations between variables can be stated in the form of constraints. Constraints differ from the common primitives of other programming languages in that they do not specify a step or sequence of steps to execute but rather the properties or requirements of the solution to be found. The constraints used in constraint programming are of various kinds, including constraint satisfaction problems. Constraints are often embedded within a programming language or provided via separate software libraries (O'Sullivan 2003, Krzysztof and Wallace 2007).

Recent research has also extended traditional logic programming by adding distributions to declarative specifications (Pless and Luger 2003, Chakrabarti et al. 2005, Sakhanenko et al. 2007). This is a natural extension, in that declarative specifications do not be need to be seen as deterministic, but may be more realistically cast as probabilistic.

There is ongoing interest in logic-based or pure declarative programming environments other than Prolog. *The Gödel Programming Language*, by Hill and Lloyd (1994), presents the Gödel language and Somogyi, Henderson, and Conway (1995) describe Mercury. Gödel and Mercury are two relatively new declarative logic-programming environments.

Finally, Prolog is a general-purpose language, and, because of space limitations, we have been unable to present a number of its important features and control constructs. We recommend that the interested reader pursue some of the many excellent texts available including *Programming in Prolog* (Clocksin and Mellish 2003), *Computing with Logic* (Maier and Warren 1988), *The Art of Prolog* (Sterling and Shapiro 1986), *The Craft of Prolog* (O'Keefe 1990), *Techniques of Prolog Programming* (VanLe 1993), *Mastering Prolog* (Lucas 1996), or *Advanced Prolog: Techniques and Examples* (Ross 1989), *Knowledge Systems through Prolog* (King 1991), and *Natural Language Processing in Prolog* (Gazdar and Mellish 1989).

In Part III we present the philosophy and idioms of functional programming, using the Lisp language. Part IV then presents object-oriented design and programming with Java, and Part V offers our summary. As the reader covers the different parts of this book it can be seen how the different languages are utilized to address many of the same problems, while the idioms of one programming paradigm may or may not be suitable to another.

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.

11 S-expressions, the Syntax of Lisp

Chapter Objectives

The Lisp, s-expression introduced
    Basic syntactic unit for the language
Structures defined recursively
The list as data or function
    `quote`
    `eval`
Creating new functions in Lisp:
    `defun`
Control structures in Lisp
    Functions
        `cond`
        `if`
    Predicates
        `and`
        `or`
        `not`

Chapter Contents

11.1 Introduction to Symbol Expressions

The S-expression

The syntactic elements of the Lisp programming language are *symbolic expressions*, also known as *s-expressions*. Both programs and data are represented as s-expressions: an s-expression may be either an *atom* or a *list*. Lisp atoms are the basic syntactic units of the language and include both numbers and symbols. Symbolic atoms are composed of letters, numbers, and the non-alphanumeric characters.

Examples of Lisp atoms include:

```
3.1416
100
hyphenated-name
*some-global*
nil
```

A *list* is a sequence of either atoms or other lists separated by blanks and enclosed in parentheses. Examples of lists include:

```
(1 2 3 4)
(george kate james joyce)
(a (b c) (d (e f)))
()
```

Note that lists may be elements of lists. This nesting may be arbitrarily

deep and allows us to create symbol structures of any desired form and complexity. The empty list, "`()`", plays a special role in the construction and manipulation of Lisp data structures and is given the special name `nil`. `nil` is the only s-expression that is considered to be both an atom and a list. Lists are extremely flexible tools for constructing representational structures. For example, we can use lists to represent expressions in the predicate calculus:

```
(on block-1 table)
(likes bill X)
(and (likes george kate) (likes bill merry))
```

We use this syntax to represent predicate calculus expressions in the unification algorithm of this chapter. The next two examples suggest ways in which lists may be used to implement the data structures needed in a database application.

```
((2467 (lovelace ada) programmer)
 (3592 (babbage charles) computer-designer))
((key-1 value-1) (key-2 value-2) (key-3 value-3))
```

An important feature of Lisp is its use of Lisp syntax to represent programs as well as data. For example, the lists,

```
(* 7 9)
(– (+ 3 4) 7)
```

may be interpreted as arithmetic expressions in a prefix notation. This is exactly how Lisp treats these expressions, with `(* 7 9)` representing the product of 7 and 9. When Lisp is invoked, the user enters an interactive dialogue with the Lisp interpreter. The interpreter prints a prompt, in our examples "`>`", reads the user input, attempts to evaluate that input, and, if successful, prints the result. For example:

```
> (* 7 9)
63
>
```

Here, the user enters `(* 7 9)` and the Lisp interpreter responds with 63, i.e., the *value* associated with that expression. Lisp then prints another prompt and waits for more user input. This cycle is known as the *read-eval-print* loop and is the heart of the Lisp interpreter.

When given a list, the Lisp evaluator attempts to interpret the first element of the list as the name of a function and the remaining elements as its arguments. Thus, the s-expression `(f x y)` is equivalent to the more traditional looking mathematical function notation `f(x,y)`. The value printed by Lisp is the result of *applying* the function to its arguments. Lisp expressions that may be meaningfully evaluated are called *forms*. If the user enters an expression that may not be correctly evaluated, Lisp prints an error message and allows the user to trace and correct the problem. A sample Lisp session appears below:

```
> (+ 14 5)
19
> (+ 1 2 3 4)
10
> (– (+ 3 4) 7)
0
```

```
> (* (+ 2 5) (— 7 (/ 21 7)))
28
> (= (+ 2 3) 5)
t
> (> (* 5 6) (+ 4 5))
t
> (a b c)
Error: invalid function: a
```

Several of the examples above have arguments that are themselves lists, for example the expression `(— (+ 3 4) 7)`. This indicates the composition of functions, in this case "subtract 7 from the *result* of adding 3 to 4". The word "result" is emphasized here to indicate that the function—is not passed the s-expression "`(+ 3 4)`" as an argument but rather the result of *evaluating* that expression.

In evaluating a function, Lisp first evaluates its arguments and then applies the function indicated by the first element of the expression to the results of these evaluations. If the arguments are themselves function expressions, Lisp applies this rule recursively to their evaluation. Thus, Lisp allows nested function calls of arbitrary depth. It is important to remember that, by default, Lisp evaluates everything. Lisp uses the convention that numbers always evaluate to themselves. If, for example, `5` is typed into the Lisp interpreter, Lisp will respond with `5`. Symbols, such as `x`, may have a value *bound* to them. If a symbol is bound, the binding is returned when the symbol is evaluated (one way in which symbols become bound is in a function call; see Section 13.2). If a symbol is unbound, it is an error to evaluate that symbol.

For example, in evaluating the expression `(+ (* 2 3) (* 3 5))`, Lisp first evaluates the arguments, `(* 2 3)` and `(* 3 5)`. In evaluating `(* 2 3)`, Lisp evaluates the arguments `2` and `3`, which return their respective arithmetic values; these values are multiplied to yield `6`. Similarly, `(* 3 5)` evaluates to `15`. These results are then passed to the top-level addition, which is evaluated, returning `21`. A diagram of this evaluation appears in Figure 11.1.

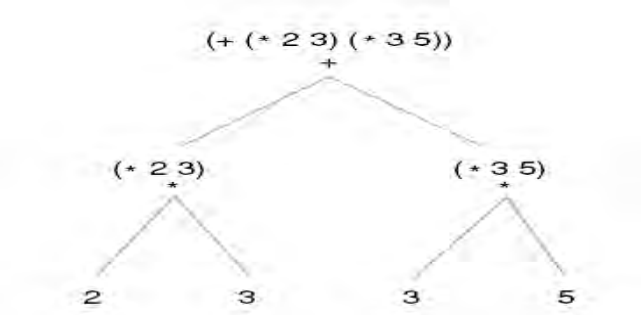

Figure 11.1. Tree representation of the evaluation of a simple Lisp function

In addition to arithmetic operations, Lisp includes a large number of functions that operate on lists. These include functions to construct and combine lists, to access elements of lists, and to test various properties. For example, `list` takes any number of arguments and constructs a list of those elements. `nth` takes a number and a list as arguments and returns

the indicated element of the list. By convention, `nth` begins counting with `0`. Examples of these and other list manipulation functions include:

```
> (list 1 2 3 4 5)
(1 2 3 4 5)
> (nth 0 '(a b c d))
a
> (nth 2 (list 1 2 3 4 5))
3
> (nth 2 '((a 1) (b 2) (c 3) (d 4)))
(c 3)
> (length '(a b c d))
4
> (member 7 '(1 2 3 4 5))
nil
> (null ( ))
t
```

S-expressions Defined

DEFINITION

S-EXPRESSION

An *s-expression* is defined recursively:

An *atom* is an s-expression.

If s_1, s_2, …, s_n are s-expressions, then so is the list $(s_1\ s_2\ \ldots\ s_n)$.

A *list* is a non-atomic s-expression.

A form is an s-expression that is intended to be evaluated. If it is a list, the first element is treated as the function name and the subsequent elements are evaluated to obtain the function arguments.

In evaluating an s-expression:

If the s-expression is a number, return the value of the number.

If the s-expression is an atomic symbol, return the value bound to that symbol; if it is not bound, it is an error.

If the s-expression is a list, evaluate the second through the last arguments and apply the function indicated by the first argument to the results.

Lisp represents both programs and data as s-expressions. Not only does this simplify the syntax of the language but also, when combined with the ability to control the evaluation of s-expressions, it makes it easy to write programs that treat other Lisp programs as data. This simplifies the implementation of interpreters in Lisp.

11.2 Control of Lisp Evaluation

Using `quote` and `eval`

In the previous section, several of the examples included list arguments preceded by a single quotation mark: `'`. The `'`, which can also be represented by the function `quote`, is a special function which does not evaluate its argument but prevents evaluation, often because its argument is to be treated as data rather than as an evaluable form.

When evaluating an s-expression, Lisp will first try to evaluate all of its arguments. If the interpreter is given the expression `(nth 0 (a b c d))`, it will first try to evaluate the argument `(a b c d)`. This attempted evaluation will result in an error, because `a`, the first element of this s-expression, does not represent any known Lisp function. To prevent this, Lisp provides the user with the built-in function `quote`. `quote` takes one argument and returns that argument without evaluating it. For example:

```
> (quote (a b c))
(a b c)
> (quote (+ 1 3))
(+ 1 3)
```

Because `quote` is used so often, Lisp allows it to be abbreviated by a single quotation mark. Thus, the preceding examples could be written:

```
> '(a b c)
(a b c)
> '(+ 1 3)
(+ 1 3)
```

In general, `quote` is used to prevent the evaluation of arguments to a function when these arguments are intended to be treated as data rather than evaluable forms. In the earlier examples of simple arithmetic, `quote` was not needed, because numbers always evaluate to themselves. Consider the effect of quote in the following calls to the `list` function:

```
> (list (+ 1 2) (+ 3 4))
(3 7)
> (list '(+ 1 2) '(+ 3 4))
((+ 1 2) (+ 3 4))
```

In the first example, the arguments are not quoted; they are therefore evaluated and passed to `list` according to the default evaluation scheme. In the second example, `quote` prevents this evaluation, with the s-expressions themselves being passed as arguments to `list`. Even though `(+ 1 2)` is a meaningful Lisp form, `quote` prevents its evaluation. The ability to prevent evaluation of programs and manipulate them as data is an important feature of Lisp.

As a complement to `quote`, Lisp also provides a function, `eval`, that allows the programmer to evaluate an s-expression at will. `eval` takes one s-expression as an argument: this argument is evaluated as is usual for arguments to functions; however, the result is then evaluated *again* and this final result is returned as the value of `eval`. Examples of the behavior of `eval` and `quote`:

```
> (quote (+ 2 3))
(+ 2 3)
> (eval (quote (+ 2 3)))   ;eval undoes the effect of quote
5
> (list '* 2 5)    ;this constructs an evaluable s-expression
(* 2 5)
> (eval (list '* 2 5)) ;this constructs and evaluates the s-expression
10
```

The `eval` function is precisely what is used in the ordinary evaluation of s-expressions. By making `quote` and `eval` available to the programmer, Lisp greatly simplifies the development of *meta-interpreters*: variations on the standard Lisp interpreter that define alternative or extended behaviors for the Lisp language. This important programming methodology is illustrated in the "infix-interpreter" of Section 15.2 and the design of an expert system shell in Section 17.2.

11.3 Programming in Lisp: Creating New Functions

Using defun

Common Lisp includes a large number of built-in functions, including:

- A full range of arithmetic functions, supporting integer, rational, real and complex arithmetic.
- A variety of looping and program control functions.
- List manipulation and other data structuring functions.
- Input/output functions.
- Forms for the control of function evaluation.
- Functions for the control of the environment and operating system.

Lisp includes too many functions to list in this chapter; for a more detailed discussion, consult a specialized Lisp text, the manual for your particular implementation, or see Chapter 20.

In Lisp, we program by defining new functions, constructing programs from this already rich repertoire of built-in functions. These new functions are defined using `defun`, which is short for *define function*. Once a function is defined it may be used in the same fashion as functions that are built into the language.

Suppose, for example, the user would like to define a function called `square` that takes a single argument and returns the square of that argument. `square` may be created by having Lisp evaluate the following expression:

```
(defun square (x)
     (* x x))
```

The first argument to `defun` is the name of the function being defined; the second is a list of the formal parameters for that function, which must all be symbolic atoms; the remaining arguments are zero or more s-expressions, which constitute the body of the new function, the Lisp code that actually defines its behavior. Unlike most Lisp functions, `defun` does not evaluate its arguments; instead, it uses them as specifications to create a new function. As with all Lisp functions, however, `defun` returns a value, although the value returned is simply the name of the new function.

The important result of evaluating a `defun` is the side effect of creating a new function and adding it to the Lisp environment. In the above example, `square` is defined as a function that takes one argument and returns the result of multiplying that argument by itself. Once a function is defined, it must be called with the same number of arguments, or "actual parameters," as there are formal parameters specified in the `defun`. When a function is called, the actual parameters are bound to the formal parameters. The body of the function is then evaluated with these bindings. For example, the call

`(square 5)` causes `5` to be bound to the formal parameter `x` in the body of the definition. When the body `(* x x)` is evaluated, Lisp first evaluates the arguments to the function. Because `x` is bound to `5` by the call, this leads to the evaluation of `(* 5 5)`.

More concisely, the syntax of a `defun` expression is:

```
(defun <function name>
    (<formal parameters>) <function body>)
```

In this definition, descriptions of the elements of a form are enclosed in angle brackets: `< >`. We use this notational convention throughout this text to define Lisp forms. Note that the formal parameters in a `defun` are enclosed in a list.

A newly defined function may be used just like any built-in function. Suppose, for example, that we need a function to compute the length of the hypotenuse of a right triangle given the lengths of the other two sides. This function may be defined according to the Pythagorean theorem, using the previously defined `square` function along with the built-in function `sqrt`. We have added a number of comments to this sample code. Lisp supports "end of line comments": it ignores all text from the first "`;`" to the end of the same line.

```
(defun hypotenuse (x y)          ;the length of the hypotenuse is
   (sqrt (+ (square x)           ; the square root of the sum of
      (square y))))              ;the squares of the other sides.
```

This example is typical in that most Lisp programs are built up of relatively small functions, each performing a single well-defined task. Once defined, these functions are used to implement higher-level functions until the desired "top-level" behavior has been defined.

11.4 Program Control in Lisp: Conditionals and Predicates

Using cond

Lisp branching is also based on function evaluation: control functions perform tests and, depending on the results, selectively evaluate alternative forms. Consider, for example, the following definition of the `absolute-value` function (note that Lisp actually has a built-in function, `abs`, that computes absolute value):

```
(defun absolute-value (x)
     (cond ((< x 0) (– x))          ;if x < 0, return –x
           ((>= x 0) x)))           ;else return x
```

This example uses the function, `cond`, to implement a conditional branch. `cond` takes as arguments a number of *condition–action* pairs:

```
(cond (< condition1 > < action1 >)
    (< condition2 > < action2 >)
        …
    (< conditionn > < actionn >))
```

Conditions and actions may be arbitrary s-expressions, and each pair is enclosed in parentheses. Like `defun`, `cond` does not evaluate all of its arguments. Instead, it evaluates the conditions in order until one of them returns a non-nil value. When this occurs, it evaluates the associated action

and returns this result as the value of the **cond** expression. None of the other actions and none of the subsequent conditions are evaluated. If all of the conditions evaluate to **nil**, **cond** returns **nil**.

An alternative definition of **absolute-value** is:

```
(defun absolute-value (x)
    (cond ((< x 0) (– x))          ;if x < 0, return –x
          (t x)))                  ;else, return x
```

This version notes that the second condition, **(>= x 0)**, is always true if the first is false. The "**t**" atom in the final condition of the **cond** statement is a Lisp atom that roughly corresponds to "true." By convention, **t** always evaluates to itself; this causes the last action to be evaluated if all preceding conditions return **nil**. This construct is extremely useful, as it provides a way of giving a **cond** statement a default action that is evaluated if and only if all preceding conditions fail.

Lisp Predicates

Although any evaluable s-expressions may be used as the conditions of a **cond**, generally these are a particular kind of Lisp function called a *predicate*. A predicate is simply a function that returns a value of either true or false depending on whether or not its arguments possess some property. The most obvious examples of predicates are the relational operators typically used in arithmetic such as **=**, **>**, and **>=**. Here are some examples of arithmetic predicates in Lisp:

```
> (= 9 (+ 4 5))
t
> (>= 17 4)
t
> (< 8 (+ 4 2))
nil
> (oddp 3)              ;oddp tests whether or not its argument is odd
t
> (minusp 6)             ;minusp tests whether its argument < 0
nil
> (numberp 17)      ;numberp tests whether its argument is numeric
t
> (numberp nil)
nil
> (zerop 0)      ;zerop is true if its argument = 0, nil otherwise
t
> (plusp 10)                  ;plusp is true if its argument > 0
t
> (plusp –2)
nil
```

Note that the predicates in the above examples do not return "true" or "false" but rather **t** or **nil**. Lisp is defined so that a predicate may return **nil** to indicate "false" and anything other than **nil** (not necessarily **t**) to indicate "true." An example of a function that uses this feature is the **member** predicate. **member** takes two arguments, the second of which must be a list. If the first argument is a member of the second, **member** returns the suffix of the second argument, containing the first argument as its initial element; if it is not, member returns **nil**. For example:

```
> (member 3 '(1 2 3 4 5))
(3 4 5)
```

One rationale for this convention is that it allows a predicate to return a value that, in the "true" case, may be of use in further processing. It also allows any Lisp function to be used as a condition in a **cond** form.

As an alternative to **cond**, the **if** form takes three arguments. The first is a test. **if** evaluates the test; if it returns a non-nil value, the **if** form evaluates its second argument and returns the result, otherwise it returns the result of evaluating the third argument. In cases involving a two-way branch, the **if** construct generally provides cleaner, more readable code than **cond**. For example, **absolute-value** could be defined using the **if** form:

```
(defun absolute-value (x)
      (if (< x 0) (– x) x))
```

In addition to **if** and **cond**, Lisp offers a wide selection of alternative control constructs, including iterative constructs such as do and while loops. Although these functions provide Lisp programmers with a wide range of control structures that fit almost any situation and programming style, we will not discuss them in this section; the reader is referred to a more specialized Lisp text for this information.

One of the more interesting program control techniques in Lisp involves the use of the logical connectives **and**, **or**, and **not**. **not** takes one argument and returns **t** if its argument is **nil** and **nil** otherwise. Both **and** and **or** may take any number of arguments and behave as you would expect from the definitions of the corresponding logical operators. It is important to note, however, that **and** and **or** are based on *conditional evaluation*.

In evaluating an **and** form, Lisp evaluates its arguments in left-to-right order, stopping when any one of the arguments evaluates to **nil** or the last argument has been evaluated. Upon completion, the **and** form returns the value of the last argument evaluated. It therefore returns non-**nil** only if all its arguments return non-**nil**. Similarly, the **or** form evaluates its arguments only until a non-**nil** value is encountered, returning this value as a result. Both functions may leave some of their arguments unevaluated, as may be seen by the behavior of the **print** statements in the following example. In addition to printing its argument, in some Lisp environments **print** returns a value of **nil** on completion.

```
> (and (oddp 2) (print "eval second statement"))
nil
> (and (oddp 3) (print "eval second statement"))
eval second statement
> (or (oddp 3) (print "eval second statement"))
t
> (or (oddp 2) (print "eval second statement"))
eval second statement
```

Because **(oddp 2)** evaluates to **nil** in the first expressions, the **and** simply returns **nil** without evaluating the **print** form. In the second expression, however, **(oddp 3)** evaluates to **t** and the **and** form then

evaluates the `print`. A similar analysis may be applied to the `or` examples. It is important to be aware of this behavior, particularly if some of the arguments are forms whose evaluations have side effects, such as the `print` function. The conditional evaluation of logical connectives makes them useful in controlling the flow of execution of Lisp programs. For example, an `or` form may be used to try alternative solutions to a problem, evaluating them in order until one of them returns a non-`nil` result.

Exercises

1. Which of the following are legitimate s-expressions? If any is not, explain why it isn't.

```
(geo rge fred john)
(a b (c d (e f (g h)))
(3 + 5)
(quote (eval (+ 2 3)))
(or (oddp 4) (* 4 5 6)
```

2. Create a small database in Lisp for some application, such as for professional contacts. Have at least five fields in the data-tuples where at least one of the fields is itself a list of items. Create and test your own assess functions on this database.

3. Create a `cond` form that uses `and` and `or` that will test the items in the database created in exercise 2. Use these forms to test for properties of the data-tuples, such as to print out the name of a male person that makes more than a certain amount of money.

4. Create a function called `my-member` that performs the function of the `member` example that was presented in Section 11.4.

12 Lists and Recursive Search

Chapter Objectives

Lisp processing of arbitrary symbol structures
- Building blocks for data structures
- Designing accessors

The symbol list as building block
- `car`
- `cdr`
- `cons`

Recursion as basis of list processing
- `cdr` recursion
- `car-cdr` recursion

The tree as representative of the structure of a list

Chapter Contents

12.1 Functions, Lists, and Symbolic Computing

Symbolic Computing

Although the Chapter 11 introduced Lisp syntax and demonstrated a few useful Lisp functions, it did so in the context of simple arithmetic examples. The real power of Lisp is in symbolic computing and is based on the use of lists to construct arbitrarily complex data structures of symbolic and numeric atoms, along with the forms needed for manipulating them. We illustrate the ease with which Lisp handles symbolic data structures, as well as the naturalness of data abstraction techniques in Lisp, with a simple database example. Our database application requires the manipulation of employee records containing name, salary, and employee number fields.

These records are represented as lists, with the name, salary, and number fields as the first, second, and third elements of a list. Using `nth`, it is possible to define access functions for the various fields of a data record. For example:

```
(defun name-field (record)
     (nth 0 record))
```

will have the behavior:

```
> (name-field '((Ada Lovelace) 45000.00 38519))
(Ada Lovelace)
```

Similarly, the functions `salary-field` and `number-field` may be defined to access the appropriate fields of a data record. Because a name is itself a list containing two elements, a first name and a last name, it is useful to define functions that take a name as argument and return either the first or last name as a result.

```
(defun first-name (name)
     (nth 0 name))
```

will have the behavior:

```
> (first-name (name-field '((Ada Lovelace) 45000.00
338519)))
Ada
```

In addition to accessing individual fields of a data record, it is also necessary to implement functions to create and modify data records. These are defined using the built-in Lisp function `list`. `list` takes any number of arguments, evaluates them, and returns a list containing those values as its elements. For example:

```
> (list 1 2 3 4)
(1 2 3 4)
> (list '(Ada Lovelace) 45000.00 338519)
((Ada Lovelace) 45000.00 338519)
```

As the second of these examples suggests, `list` may be used to define a constructor for records in the database:

```
(defun build-record (name salary emp-number)
     (list name salary emp-number))
```

will have the behavior:

```
> (build-record '(Alan Turing) 50000.00 135772)
((Alan Turing) 50000.00 135772)
```

Now, using `build-record` and the access functions, we may construct functions that return a modified copy of a record. For example `replace-salary` will behave:

```
(defun replace-salary-field (record new-salary)
(build-record (name-field record)
     new-salary
     (number-field record)))
> (replace-salary-field '((Ada Lovelace) 45000.00
338519) 50000.00)
((Ada Lovelace) 50000.00 338519)
```

Note that this function does not actually update the record itself but produces a modified copy of the record. This updated version may be saved by binding it to a global variable using `setf` (Section 13.1). Although Lisp provides forms that allow a particular element in a list to be modified in the original structure (i.e., without making a copy), good Lisp programming style generally avoids their use, and they are not covered in this text. For Lisp applications involving all but extremely large structures, modifications are generally done by creating a new copy of the structure.

In the above examples, we created an abstract data type for employee records. The various access and update functions defined in this section implement a specialized language appropriate to the meaning of the records, freeing the programmer from concerns about the actual list structures being used to implement the records. This simplifies the

development of higher-level code, as well as making that code much easier to maintain and understand.

Generally, AI programs manipulate large amounts of varied knowledge about problem domains. The data structures used to represent this knowledge, such as objects and semantic networks, are complex, and humans generally find it easier to relate to this knowledge in terms of its meaning rather than the particular syntax of its internal representation. Therefore, data abstraction techniques, always good practice in computer science, are essential tools for the AI programmer. Because of the ease with which Lisp supports the definition of new functions, it is an ideal language for data abstraction.

12.2 Lists as Recursive Structures

Car/cdr Recursion

In the previous section, we used `nth` and `list` to implement access functions for records in a simple "employee" database. Because all employee records were of a determinate length (three elements), these two functions were sufficient to access the fields of records. However, these functions are not adequate for performing operations on lists of unknown length, such as searching through an unspecified number of employee records. To do this, we must be able to scan a list iteratively or recursively, terminating when certain conditions are met (e.g., the desired record is found) or the list is exhausted. In this section we introduce list operations, along with the use of recursion to create list-processing functions.

The basic functions for accessing the components of lists are `car` and `cdr`. `car` takes a single argument, which must be a list, and returns the first element of that list. `cdr` also takes a single argument, which must also be a list, and returns that list with its first argument removed:

```
> (car '(a b c))        ;note that the list is quoted
a
> (cdr '(a b c))
(b c)
> (car '((a b) (c d)))          ;the first element of
(a b)                           ;a list may be a list
> (cdr '((a b) (c d)))
((c d))
> (car (cdr '(a b c d)))
b
```

The way in which `car` and `cdr` operate suggests a recursive approach to manipulating list structures. *To perform an operation on each of the elements of a list:*

> If the list is empty, quit.
>
> Otherwise, operate on the first element and recurse on the remainder of the list.

Using this scheme, we can define a number of useful list-handling functions. For example, Common Lisp includes the predicates `member`, which determines whether one s-expression is a member of a list, and `length`, which determines the length of a list. We define our own

versions of these functions: **my-member** takes two arguments, an arbitrary s-expression and a list, **my-list**. It returns **nil** if the s-expression is not a member of **my-list**; otherwise it returns the list containing the s-expression as its first element:

```
(defun my-member (element my-list)
   (cond ((null my-list) nil)
      ((equal element (car my-list)) my-list)
      (t (my-member element (cdr my-list)))))
```

my-member has the behavior:

```
> (my-member 4 '(1 2 3 4 5 6))
(4 5 6)
> (my-member 5 '(a b c d))
nil
```

Similarly, we may define our own versions of **length** and **nth**:

```
(defun my-length (my-list)
     (cond ((null my-list) 0)
           (t (+ (my-length (cdr my-list)) 1))))
(defun my-nth (n my-list)
     (cond ((zerop n) (car my-list))
     ; zerop tests if argument is zero
           (t (my-nth (- n 1) (cdr my-list)))))
```

It is interesting to note that these examples, though presented here to illustrate the use of **car** and **cdr**, reflect the historical development of Lisp. Early versions of the language did not include as many built-in functions as Common Lisp does; programmers defined their own functions for checking list membership, length, etc. Over time, the most generally useful of these functions have been incorporated into the language standard. As an easily extensible language, Common Lisp makes it easy for programmers to create and use their own library of reusable functions.

In addition to the functions **car** and **cdr**, Lisp provides a number of functions for constructing lists. One of these, **list**, which takes as arguments any number of s-expressions, evaluates them, and returns a list of the results, was introduced in Section 10.1. A more primitive list constructor is the function **cons**, that takes two s-expressions as arguments, evaluates them, and returns a list whose **car** is the value of the first argument and whose **cdr** is the value of the second:

```
> (cons 1 '(2 3 4))
(1 2 3 4)
> (cons '(a b) '(c d e))
((a b) c d e)
```

cons bears an inverse relationship to **car** and **cdr** in that the **car** of the value returned by a **cons** form is always the first argument to the **cons**, and the **cdr** of the value returned by a **cons** form is always the second argument to that form:

```
> (car (cons 1 '(2 3 4)))
1
> (cdr (cons 1 '(2 3 4)))
(2 3 4)
```

An example of the use of **cons** is seen in the definition of the function **filter-negatives**, which takes a list of numbers as an argument and returns that list with any negative numbers removed. **filter-negatives** recursively examines each element of the list; if the first element is negative, it is discarded and the function returns the result of filtering the negative numbers from the **cdr** of the list. If the first element of the list is positive, it is "**cons**ed" onto the result of **filter-negatives** from the rest of the list:

```
(defun filter-negatives (number-list)
      (cond ((null number-list) nil)
          ((plusp (car number-list))
              (cons (car number-list)
              (filter-negatives
                    (cdr number-list))))
      (t (filter-negatives (cdr number-list)))))
```

This function behaves:

```
> (filter-negatives '(1 –1 2 –2 3 –4))
(1 2 3)
```

This example is typical of the way **cons** is often used in recursive functions on lists. **car** and **cdr** tear lists apart and "drive" the recursion; **cons** selectively constructs the result of the processing as the recursion "unwinds." Another example of this use of **cons** is in redefining the built-in function **append**:

```
(defun my-append (list1 list2)
      (cond ((null list1) list2)
              (t (cons (car list1)
              (my-append (cdr list1) list2)))))
```

which yields the behavior:

```
> (my-append '(1 2 3) '(4 5 6))
(1 2 3 4 5 6)
```

Note that the same recursive scheme is used in the definitions of **my-append**, **my-length**, and **my-member**. Each definition uses the **car** function to remove (and process) the first element of the list, followed by a recursive call on the shortened (tail of the) list; the recursion "bottoms out" on the empty list. As the recursion unwinds, the **cons** function reassembles the solution. This particular scheme is known as ***cdr*** *recursion*, because it uses the **cdr** function to linearly scan and process the elements of a list.

12.3 Nested Lists, Structure, and car/cdr Recursion

Car/cdr Recursion and Nested Structure

Although both `cons` and `append` may be used to combine smaller lists into a single list, it is important to note the difference between these two functions. If `cons` is called with two lists as arguments, it makes the first of these a new first element of the second list, whereas `append` returns a list whose elements are the elements of the two arguments:

```
> (cons '(1 2) '(3 4))
((1 2) 3 4)
> (append '(1 2) '(3 4))
(1 2 3 4)
```

The lists `(1 2 3 4)` and `((1 2) 3 4)` have fundamentally different structures. This difference may be noted graphically by exploiting the isomorphism between lists and trees. The simplest way to map lists onto trees is to create an unlabeled node for each list, with descendants equal to the elements of that list. This rule is applied recursively to the elements of the list that are themselves lists; elements that are atoms are mapped onto leaf nodes of the tree. Thus, the two lists mentioned above generate the different tree structures illustrated in Figure 12.1.

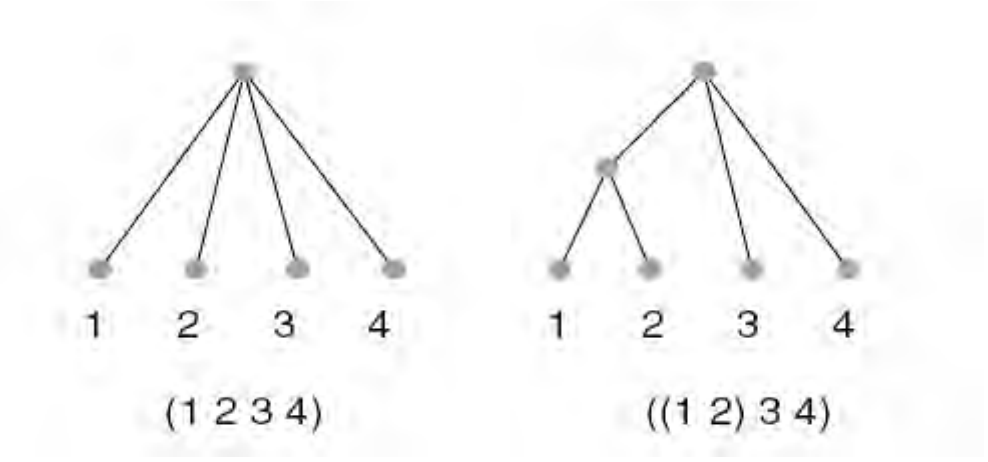

Figure 12.1. Mapping lists onto trees showing structural differences.

This example illustrates the representational power of lists, particularly as a means of representing any tree structure such as a search tree or a parse tree (Figure 16.1). In addition, nested lists provide a way of hierarchically structuring complex data. In the employee records example of Section 12.1, the name field was itself a list consisting of a first name and a last name. This list could be treated as a single entity or its individual components could be accessed.

The simple `cdr`-recursive scheme discussed in the previous section is not sufficient to implement all manipulations on nested lists, because it does not distinguish between items that are lists and those that are simple atoms. Suppose, for example, that the length function defined in Section 12.2 is applied to a nested list structure:

```
> (length '((1 2) 3 (1 (4 (5)))))
3
```

In this example, `length` returns `3` because the list has 3 elements, `(1 2)`, `3`, and `(1 (4 (5)))`. This is, of course, the correct and desired behavior for a `length` function.

On the other hand, if we want the function to count the number of *atoms* in the list, we need a different recursive scheme, one that, in addition to scanning along the elements of the list, "opens up" non-atomic list elements and recursively applies itself to the task of counting their atoms. We define this function, called `count-atoms`, and observe its behavior:

```
(defun count-atoms (my-list)
      (cond ((null my-list) 0)
            ((atom my-list) 1)
            (t (+ (count-atoms (car my-list))
                  (count-atoms
                 (cdr my-list))))))
> (count-atoms '((1 2) 3 (((4 5 (6))))))
6
```

The above definition is an example of `car-cdr` recursion. Instead of just recurring on the `cdr` of the list, `count-atoms` also recurs on the `car` of its argument, with the `+` function combining the two components into an answer. Recursion halts when it encounters an `atom` or empty list (`null`). One way of thinking of this scheme is that it adds a second dimension to simple `cdr` recursion, that of "going down into" each of the list elements. Compare the diagrams of calls to `length` and `count-atoms` in Figure 12.2. Note the similarity of `car-cdr` recursion and the recursive definition of s-expressions given in Section 11.1.1.

Another example of the use of `car-cdr` recursion is in the definition of the function `flatten`. `flatten` takes as argument a list of arbitrary structure and returns a list that consists of the same atoms in the same order but with all the atoms at the same level. Note the similarity between the definition of `flatten` and that of `count-atoms`: both use `car-cdr` recursion to tear apart lists and drive the recursion, both terminate when the argument is either `null` or an `atom`, and both use a second function (`append` or `+`) to construct an answer from the results of the recursive calls.

```
(defun flatten (lst)
      (cond ((null lst) nil)
            ((atom lst) (list lst))
            (t (append (flatten (car lst))
                       (flatten (cdr lst))))))
```

Examples of the behavior of `flatten` include:

```
> (flatten '(a (b c) (((d) e f))))
(a b c d e f)
> (flatten '(a b c)); already flattened
(a b c)
```

```
> (flatten '(1 (2 3) (4 (5 6) 7)))
(1 2 3 4 5 6 7)
```

`car-cdr` recursion is the basis of our implementation of unification in Section 15.2. In Chapter 13, we introduce variables and design algorithms for search.

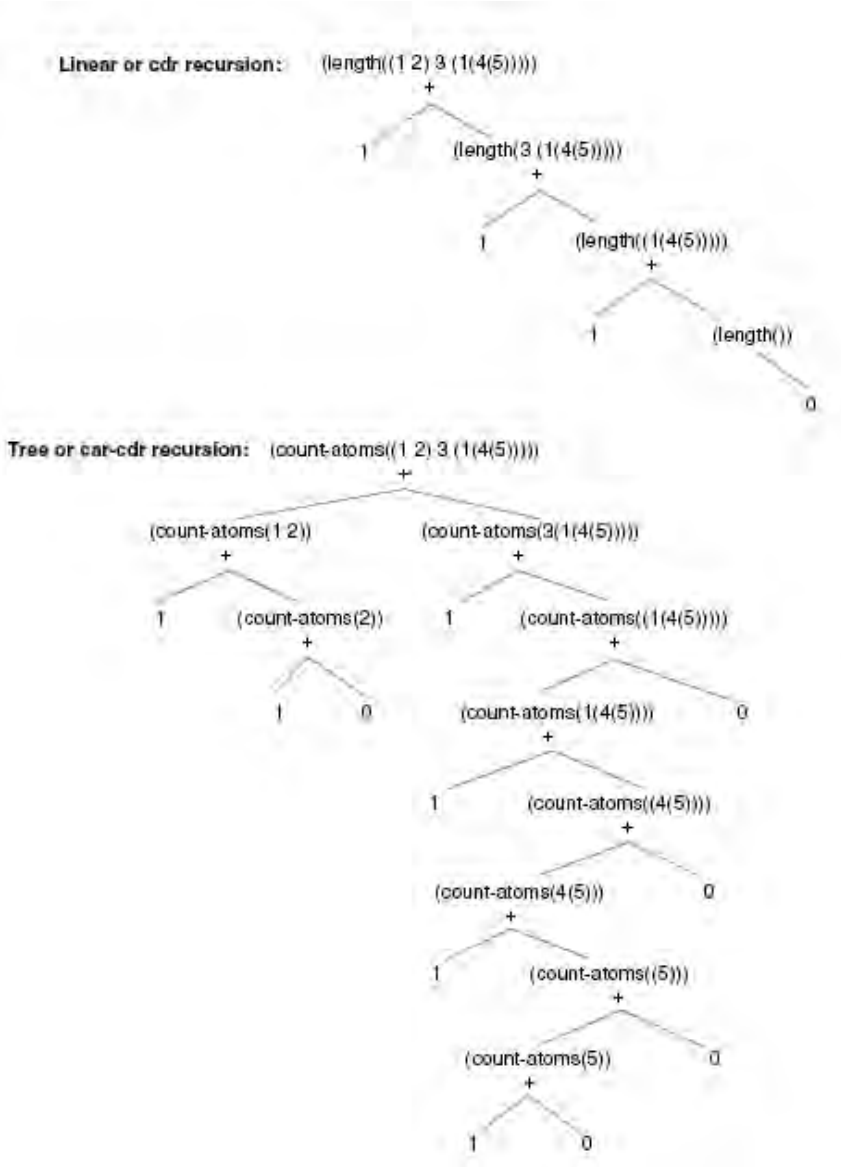

Figure 12.2. Tree representations of linear and tree-recursive functions.

Exercises

1. Create trees, similar to those of Figure 12.1, which show the structures of the following lists.

```
(+ 4 (* 5 (+ 6 7 8)))
(+ (* (+ 4 5) 6 7 8))
(+ (* (+ 4 (* 5 6)) 7) 8)
```

2. Write a recursive Lisp function that will reverse the elements of a list. (Do not use the built-in `reverse` function.) What is the complexity of your implementation? It is possible to reverse a list in linear time; can you do so?

3. Write a Lisp function that will take a list nested to any depth and print the mirror image of that list. For instance, the function should have the behavior:

```
> (mirror '((a b) (c (d e))))
(((e d) c) (b a))
```

Note that the mirroring operation operates at all levels of the list's representation.

4. Consider the database example of section 12.1. Write a function, `find`, to return all records that have a given value particular value for a particular field. To make this more interesting, allow users to specify the fields to be searched by name. For example, evaluating the expression:

```
(find 'salary-field '50000.00
   '(((Alan Turing) 50000.00 135772)
   ((Ada Lovelace) 45000.00 338519)))
```

should return:

```
((Alan Turing) 50000.00 135772)
```

5. The Towers of Hanoi problem is based on the following legend:

> In a Far Eastern monastery, there is a puzzle consisting of three diamond needles and 64 gold disks. The disks are of graduated sizes. Initially, the disks are all stacked on a single needle in decreasing order of size. The monks are attempting to move all the disks to another needle under the following rules:
>
> Only one disk may be moved at a time.
>
> No disk can ever rest on a smaller disk.

Legend has it that when the task has been completed, the universe will end. Write a Lisp program to solve this problem. For safety's sake (and to write a program that will finish in your lifetime) do not attempt the full 64-disk problem. Four or five disks is more reasonable.

6. Write a compiler for arithmetic expressions of the form:

```
(op operand1 operand2)
```

where `op` is either `+`, `–`, `*`, or `/` and the operands are either numbers or nested expressions. An example is `(* (+ 3 6) (– 7 9))`. Assume that the target machine has instructions:

```
(move value register)
(add register-1 register-2)
(subtract register-1 register-2)
(times register-1 register-2)
(divide register-1 register-2)
```

All the arithmetic operations will leave the result in the first register argument. To simplify, assume an unlimited number of registers. Your compiler should take an arithmetic expression and return a list of these machine operations.

13 Variables, Datatypes, and Search

Chapter Objectives

Variables introduced
- Basic support for type systems

Fundamental to search

Creating and binding variables
- `set`
- `let`

Depth-first search in Lisp:
- Use of production system architecture
- Backtracking supports search of all options

Chapter Contents

13.1 Variables and Datatypes

We begin this chapter by demonstrating the creation of variables using `set` and `let` and discussing the scope of their bindings. We then introduce datatypes and discuss run-time type checking. We finish this chapter with a sample search, where we use variables for state and recursion for generation of the state-space graph.

Binding Variables Using Set

Lisp is based on the theory of recursive functions; early Lisp was the first example of a functional or *applicative* programming language. An important aspect of purely functional languages is the lack of any side effects as a result of function execution. This means that the value returned by a function call depends only on the function definition and the value of the parameters in the call. Although Lisp is based on mathematical functions, it is possible to define Lisp forms that violate this property. Consider the following Lisp interaction:

```
> (f 4)
5
> (f 4)
6
> (f 4)
7
```

Note that `f` does not behave as a true function in that its output is not determined solely by its actual parameter: each time it is called with `4`, it returns a different value. This implies that f is retaining its state, and that each execution of the function creates a side effect that influences the behavior of future calls. `f` is implemented using a Lisp built-in function called `set`:

```
(defun f (x)
      (set 'inc (+ inc 1))
      (+ x inc))
```

set takes two arguments. The first must evaluate to a symbol; the second may be an arbitrary s-expression. **set** evaluates the second argument and assigns this value to the symbol defined by the first argument. In the above example, if **inc** is first set to **0** by the call **(set 'inc 0)**, each subsequent evaluation will increment its parameter by one.

set requires that its first argument evaluate to a symbol. In many cases, the first argument is simply a quoted symbol. Because this is done so often, Lisp provides an alternative form, **setq**, which does not evaluate its first argument. Instead, **setq** requires that the first argument be a symbol. For example, the following forms are equivalent:

```
> (set 'x 0)
0
> (setq x 0)
0
```

Although this use of **set** makes it possible to create Lisp objects that are not pure functions in the mathematical sense, the ability to bind a value to a variable in the global environment is a useful feature. Many programming tasks are most naturally implemented using this ability to define objects whose state persists across function calls. The classic example of this is the "seed" in a random number generator: each call to the function changes and saves the value of the seed. Similarly, it would be natural for a database program (such as was described in Section 11.3) to store the database by binding it to a variable in the global environment.

So far, we have seen two ways of giving a value to a symbol: explicitly, by assignment using **set** or **setq**, or implicitly, when a function call binds the calling parameters to the formal parameters in the definition. In the examples seen so far, all variables in a function body were either *bound* or *free*. A bound variable is one that appears as a formal parameter in the definition of the function, while a free variable is one that appears in the body of the function but is not a formal parameter. When a function is called, any bindings that a bound variable may have in the global environment are saved and the variable is rebound to the calling parameter. After the function has completed execution, the original bindings are restored. Thus, setting the value of a bound variable inside a function body has no effect on the global bindings of that variable, as seen in the Lisp interaction:

```
> (defun foo (x)
      (setq x (+ x 1))    ;increment bound variable x
      x)                            ;return its value.
foo
> (setq y 1)
1
```

```
> (foo y)
2
> y                ;note that value of y is unchanged.
1
```

In the example that began this section, `x` was bound in the function `f`, whereas `inc` was free in that function. As we demonstrated in the example, free variables in a function definition are the primary source of side effects in functions.

An interesting alternative to `set` and `setq` is the generalized assignment function, `setf`. Instead of assigning a value to a symbol, `setf` evaluates its first argument to obtain a memory location and places the value of the second argument in that location. When binding a value to a symbol, `setf` behaves like `setq`:

```
> (setq x 0)
0
> (setf x 0)
0
```

However, because we may call `setf` with any form that corresponds to a memory location, it allows a more general semantics. For example, if we make the first argument to `setf` a call to the `car` function, `setf` will replace the first element of that list. If the first argument to `setf` is a call to the `cdr` function, `setf` will replace the tail of that list. For example:

```
> (setf x '(a b c))            ;x is bound to a list.
(a b c)
> x                         ;The value of x is a list.
(a b c)
> (setf (car x) 1)    ;car of x is a memory location.
1
> x                  ;setf changed value of car of x.
(1 b c)
> (setf (cdr x) '(2 3))
(2 3)
> x                  ;Note that x now has a new tail.
(1 2 3)
```

We may call `setf` with most Lisp forms that correspond to a memory location; these include symbols and functions such as `car`, `cdr`, and `nth`. Thus, `setf` allows the program designer great flexibility in creating, manipulating, and even replacing different components of Lisp data structures.

Defining Local Variables Using `let`

`let` is a useful function for explicitly controlling the binding of variables. `let` allows the creation of local variables.

As an example, consider a function to compute the roots of a quadratic equation. The function quad-roots will take as arguments the three parameters `a`, `b`, and `c` of the equation $ax^2 + bx + c = 0$ and

return a list of the two roots of the equation. These roots will be calculated from the formula:

$$x = \frac{-b +/- sqrt(b2 - 4ac)}{2a}$$

For example:

```
> (quad-roots 1 2 1)
(-1.0 -1.0)
> (quad-roots 1 6 8)
(-2.0 -4.0)
```

In computing `quad-roots`, the value of

```
sqrt(b2 - 4ac)
```

is used twice. For reasons of efficiency, as well as elegance, we should compute this value only once, saving it in a variable for use in computing the two roots. Based on this idea, an initial implementation of `quad-roots` might be:

```
(defun quad-roots-1 (a b c)
      (setq temp (sqrt (- (* b b) (* 4 a c))))
           (list (/ (+ (- b) temp) (* 2 a))
                 (/ (- (- b) temp) (* 2 a))))
```

Note that the above implementation assumes that the equation does not have imaginary roots, as attempting to take the square root of a negative number would cause the `sqrt` function to halt with an error condition. Modifying the code to handle this case is straightforward and not relevant to this discussion.

Although, with this exception, the code is correct, evaluation of the function body will have the side effect of setting the value of `temp` in the global environment:

```
> (quad-roots-1 1 2 1)
(-1.0 -1.0)
> temp
0.0
```

It is much more desirable to make `temp` local to the function `quad-roots`, thereby eliminating this side effect. This can be done through the use of a `let` block. A `let` expression has the syntax:

```
(let (<local-variables>) <expressions>)
```

where the elements of (`<local-variables>`) are either symbolic atoms or pairs of the form:

```
(<symbol> <expression>)
```

When a `let` form (or *block* as it is usually called) is evaluated, it establishes a local environment consisting of all of the symbols in (`<local-variables>`). If a symbol is the first element of a pair, the second element is evaluated and the symbol is bound to this result; symbols that are not included in pairs are bound to `nil`. If any of these symbols are already bound in the global environment, these global bindings are saved and restored when the `let` block terminates.

After these local bindings are established, the `<expressions>` are evaluated in order within this environment. When the `let` statement terminates, it returns the value of the last expression evaluated within the block. The behavior of the `let` block is illustrated by the following example:

```
> (setq a 0)
0
> (let ((a 3) b)
      (setq b 4)
      (+ a b))
7
> a
0
> b
ERROR – b is not bound at top level.
```

In this example, before the `let` block is executed, `a` is bound to `0` and `b` is unbound at the top-level environment. When the `let` is evaluated, `a` is bound to `3` and `b` is bound to `nil`. The `setq` assigns `b` to `4`, and the sum of `a` and `b` is returned by the `let` statement. Upon termination of the `let`, `a` and `b` are restored to their previous values, including the unbound status of `b`.

Using the `let` statement, `quad-roots` can be implemented with no global side effects:

```
(defun quad-roots-2 (a b c)
      (let (temp) (setq temp (sqrt (– (* b b)
                                      (* 4 a c))))
            (list (/ (+ (–b) temp) (* 2 a))
                  (/ (– (– b) temp) (* 2 a))))))
```

Alternatively, `temp` may be bound when it is declared in the `let` statement, giving a somewhat more concise implementation of `quad-roots`. In this final version, the denominator of the formula, `2a`, is also computed once and saved in a local variable, `denom`:

```
(defun quad-roots-3 (a b c)
      (let ((temp (sqrt (–. (* b b) (* 4 a c))))
            (denom (* 2 a)))
                  (list (/ (+ (– b) temp) denom)
                        (/ (– (– b) temp) denom))))
```

In addition to avoiding side effects, `quad-roots-3` is the most efficient of the three versions, because it does not recompute values unnecessarily.

Data Types in Common Lisp

Lisp provides a number of built-in data types. These include integers, floating-point numbers, strings, and characters. Lisp also includes such structured types as arrays, hash tables, sets, and structures. All of these types include the appropriate operations on the type and predicates for testing whether an object is an instance of the type. For example, lists are supported by such functions as `listp`, which identifies an object as a list;

`null`, which identifies the empty list, and constructors and accessors such as `list`, `nth`, `car`, and `cdr`.

However, unlike such strongly typed languages as C or Pascal, where all expressions can be checked for type consistency before run time, in Lisp it is the data objects that are typed, rather than variables. Any Lisp symbol may bind to any object. This provides the programmer with the power of typing but also with a great deal of flexibility in manipulating objects of different or even unknown types. For example, we may bind any object to any variable at run time. This means that we may define data structures such as frames, without fully specifying the types of the values stored in them. To support this flexibility, Lisp implements run-time type checking. So if we bind a value to a symbol, and try to use this value in an erroneous fashion at run time, the Lisp interpreter will detect an error:

```
> (setq x 'a)
a
> (+ x 2)
> > Error: a is not a valid argument to +.
> > While executing: +
```

Users may implement their own type checking using either built-in or user-defined type predicates. This allows the detection and management of type errors as needed.

The preceding pages are not a complete description of Lisp. Instead, they are intended to call the reader's attention to interesting features of the language that will be of use in implementing AI data structures and algorithms. These features include:

- The naturalness with which Lisp supports a data abstraction approach to programming.
- The use of lists to create symbolic data structures.
- The use of `cond` and recursion to control program flow.
- The recursive nature of list structures and the recursive schemes involved in their manipulation.
- The use of `quote` and `eval` to control function evaluation
- The use of `set` and `let` to control variable bindings and side effects.

The remainder of the Lisp section builds on these ideas to demonstrate the use of Lisp for typical AI programming tasks such as pattern matching and the design of graph search algorithms. We begin with a simple example, the Farmer, Wolf, Goat, and Cabbage problem, where an abstract datatype is used to describe states of the world.

13.2 Search: The Farmer, Wolf, Goat, and Cabbage Problem

A Functional Approach to Search

To introduce graph search programming in Lisp, we next represent and solve the farmer, wolf, goat, and cabbage problem:

> A farmer with his wolf, goat, and cabbage come to the edge of a river they wish to cross. There is a boat at the river's edge, but, of course, only the farmer can row it. The boat also can carry only two things (including the rower) at a time. If the wolf is ever left alone with the goat, the wolf will eat the goat; similarly, if the goat is left alone with the cabbage, the goat will eat the cabbage. Devise a sequence of crossings of the river so that all four characters arrive safely on the other side of the river.

This problem was first presented in Prolog in Section 4.2. The Lisp version searches the same space and has structural similarities to the Prolog solution; however, it differs in ways that reflect Lisp's imperative/functional orientation. The Lisp solution searches the state space in a depth-first fashion using a list of visited states to avoid loops.

The heart of the program is a set of functions that define states of the world as an abstract data type. These functions hide the internals of state representation from higher-level components of the program. States are represented as lists of four elements, where each element denotes the location of the farmer, wolf, goat, or cabbage, respectively. Thus, `(e w e w)` represents the state in which the farmer (the first element) and the goat (the third element) are on the east bank and the wolf and cabbage are on the west. The basic functions defining the state data type will be a constructor, `make-state`, which takes as arguments the locations of the farmer, wolf, goat, and cabbage and returns a state, and four access functions, `farmer-side`, `wolf-side`, `goat-side`, and `cabbage-side`, which take a state and return the location of an individual. These functions are defined:

```
(defun make-state (f w g c) (list f w g c))
(defun farmer-side (state)
     (nth 0 state))
(defun wolf-side (state)
     (nth 1 state))
(defun goat-side (state)
     (nth 2 state))
(defun cabbage-side (state)
     (nth 3 state))
```

The rest of the program is built on these state access and construction functions. In particular, they are used to implement the four possible actions the farmer may take: rowing across the river alone or with either of the wolf, goat, or cabbage.

Each move uses the access functions to tear a state apart into its components. A function called `opposite` (to be defined shortly) determines the new location of the individuals that cross the river, and `make-state` reassembles

these into the new state. For example, the function `farmer-takes-self` may be defined:

```
(defun farmer-takes-self (state)
     (make-state (opposite (farmer-side state))
           (wolf-side state)
           (goat-side state)
           (cabbage-side state)))
```

Note that `farmer-takes-self` returns the new state, regardless of whether it is safe or not. A state is unsafe if the farmer has left the goat alone with the cabbage or left the wolf alone with the goat. The program must find a solution path that does not contain any unsafe states. Although this "safe" check may be done at a number of different stages in the execution of the program, our approach is to perform it in the move functions. This is implemented by using a function called `safe`, which we also define shortly. `safe` has the following behavior:

```
> (safe '(w w w w)) ;safe state, return unchanged
(w w w w)
> (safe '(e w w e)) ;wolf eats goat, return nil
nil
> (safe '(w w e e)) ;goat eats cabbage, return nil
nil
```

`safe` is used in each move-across-the-river function to filter out the unsafe states. Thus, any move that moves to an unsafe state will return `nil` instead of that state. The recursive `path` algorithm can check for this `nil` and use it to prune that state. In a sense, we are using `safe` to implement a *production system* style condition-check prior to determining if a move rule can be applied. For a detailed discussion of the production system pattern for computation see Luger (2009, Chapter 6). Using `safe`, we next present a final definition for the four move functions:

```
(defun farmer-takes-self (state)
     (safe
          (make-state (opposite (farmer-side state))
                (wolf-side state)
                (goat-side state)
                (cabbage-side state))))
(defun farmer-takes-wolf (state)
     (cond ((equal (farmer-side state)
               (wolf-side state))
            (safe  (make-state
              (opposite (farmer-side state))
              (opposite (wolf-side state))
              (goat-side state)
              (cabbage-side state))))
          (t nil)))
```

```
(defun farmer-takes-goat (state)
     (cond ((equal (farmer-side state)
                   (goat-side state))
            (safe (make-state
              (opposite (farmer-side state))
               (wolf-side state)
               (opposite (goat-side state))
               (cabbage-side state))))
           (t nil)))
(defun farmer-takes-cabbage (state)
     (cond ((equal (farmer-side state)
                   (cabbage-side state))
            (safe (make-state
              (opposite (farmer-side state))
              (wolf-side state)
              (goat-side state)
               (opposite
                  (cabbage-side state)))))
           (t nil)))
```

Note that the last three move functions include a conditional test to determine whether the farmer and the prospective passenger are on the same side of the river. If they are not, the functions return `nil`. The move definitions use the `state` manipulation functions already presented and a function `opposite`, which, for any given side, returns the other side of the river:

```
(defun opposite (side)
     (cond ((equal side 'e) 'w)
           ((equal side 'w) 'e)))
```

Lisp provides a number of different predicates for equality. The most stringent, `eq`, is true only if its arguments evaluate to the same object, i.e., point to *the same memory location*. `equal` is less strict: it requires that its arguments be syntactically identical, as in:

```
> (setq l1 '(1 2 3))
(1 2 3)
> (setq l2 '(1 2 3))
(1 2 3)
> (equal l1 l2)
t
> (eq l1 l2)
nil
> (setq l3 l1)
(1 2 3)
> (eq l1 l3)
t
```

We define safe using a `cond` to check for the two unsafe conditions: (1) the farmer on the opposite bank from the wolf and the goat and (2) the farmer on the opposite bank from the goat and the cabbage. If the state is `safe`, it is returned unchanged; otherwise, `safe` returns `nil`:

```
(defun safe (state)
      (cond ((and (equal (goat-side state)
                         (wolf-side state))
                  (not (equal (farmer-side state)
                         (wolf-side state))))
             nil)
            ((and (equal (goat-side state)
                         (cabbage-side state))
                  (not (equal (farmer-side state)
                         (goat-side state))))
             nil)
            (t state)))
```

`path` implements the backtracking search of the state space. It takes as arguments a `state` and a `goal` and first checks to see whether they are `equal`, indicating a successful termination of the search. If they are not `equal`, `path` generates all four of the neighboring states in the state space graph, calling itself recursively on each of these neighboring states in turn to try to find a path from them to a goal. Translating this simple definition directly into Lisp yields:

```
(defun path (state goal)
      (cond ((equal state goal) 'success)
            (t (or
              (path (farmer-takes-self state) goal)
              (path (farmer-takes-wolf state) goal)
              (path (farmer-takes-goat state) goal)
              (path (farmer-takes-cabbage state)
                  goal)))))
```

This version of the path function is a simple translation of the recursive path algorithm from English into Lisp and has several "bugs" that need to be corrected. It does, however, capture the essential structure of the algorithm and should be examined before continuing to correct the bugs. The first test in the `cond` statement is necessary for a successful completion of the search algorithm. When the `equal state goal` pattern matches, the recursion stops and the atom `success` is returned. Otherwise, `path` generates the four descendant nodes of the search graph and then calls itself on each of the nodes in turn.

In particular, note the use of the `or` form to control evaluation of its arguments. Recall that an `or` evaluates its arguments in turn until one of them returns a non-`nil` value. When this occurs, the `or` terminates without evaluating the other arguments and returns this non-`nil` value as a result. Thus, the `or` not only is used as a logical operator but also provides a way of

controlling branching within the space to be searched. The `or` form is used here instead of a `cond` because the value that is being tested and the value that should be returned if the test is non-`nil` are the same.

One problem with using this definition to change the problem state is that a move function may return a value of `nil` if the move may not be made or if it leads to an unsafe state. To prevent `path` from attempting to generate the children of a `nil` state, it must first check whether the current state is `nil`. If it is, `path` should return `nil`.

The other issue that needs to be addressed in the implementation of `path` is that of detecting potential loops in the search space. If the above implementation of `path` is run, the farmer will soon find himself going back and forth alone between the two banks of the river; that is, the algorithm will be stuck in an infinite loop between identical states, both of which it has already visited.

To prevent this looping from happening, `path` is given a third parameter, `been-list`, a list of all the states that have already been visited. Each time that `path` is called recursively on a new state of the world, the parent state will be added to `been-list`. `path` uses the `member` predicate to make sure the current `state` is not a member of `been-list`, i.e., that it has not already been visited. This is accomplished by checking the current problem state for membership in `been-list` before generating its descendants. `path` is now defined:

```
(defun path (state goal been-list)
      (cond ((null state) nil)
            ((equal state goal)
                (reverse (cons state been-list)))
            ((not (member state been-list
                          :test #'equal))
            (or (path (farmer-takes-self state) goal
                      (cons state been-list))
                (path (farmer-takes-wolf state) goal
                      (cons state been-list))
                (path (farmer-takes-goat state) goal
                      (cons state been-list))
                (path (farmer-takes-cabbage state)
                      goal
                      (cons state been-list))))))
```

In the above implementation, `member` is a Common Lisp built-in function that behaves in essentially the same way as the `my-member` function defined in Section 12.2. The only difference is the inclusion of `:test #'equal` in the argument list. Unlike our "home-grown" member function, the Common Lisp built-in form allows the programmer to specify the function that is used in testing for membership. This wrinkle increases the flexibility of the function and should not cause too much concern in this discussion.

Rather than having the function return just the atom `success`, it is better to have it return the actual solution path. Because the series of states on the

solution path is already contained in the `been-list`, this list is returned instead. Because the `goal` is not already on `been-list`, it is `cons`ed onto the list. Also, because the list is constructed in reverse order (with the start state as the last element), the list is reversed (constructed in reverse order using another Lisp built-in function, `reverse`) prior to being returned.

Finally, because the `been-list` parameter should be kept "hidden" from the user, a top-level calling function may be written that takes as arguments a `start` and a `goal` state and calls `path` with a `nil` value of `been-list`:

```
(defun solve-fwgc (state goal)
      (path state goal nil))
```

Finally, let us compare our Lisp version of the farmer, wolf, goat, and cabbage problem with the Prolog solution presented in Section 4.2. Not only does the Lisp program solve the same problem, but it also searches exactly the same state space as the Prolog version. This underscores the point that the state space conceptualization of a problem is independent of the implementation of a program for searching that space. Because both programs search the same space, the two implementations have strong similarities; the differences tend to be subtle but provide an interesting contrast between declarative and procedural programming styles.

States in the Prolog version are represented using a predicate, `state(e,e,e,e)`, and the Lisp implementation uses a list. These two representations are more than syntactic variations on one another. The Lisp representation of `state` is defined not only by its list syntax but also by the access and move functions that constitute the abstract data type "state." In the Prolog version, states are patterns; their meaning is determined by the way in which they match other patterns in the Prolog rules.

The Lisp version of `path` is slightly longer than the Prolog version. One reason for this is that the Lisp version must implement a search strategy, whereas the Prolog version takes advantage of Prolog's built-in search algorithm. The control algorithm is explicit in the Lisp version but is implicit in the Prolog version. Because Prolog is built on declarative representation and theorem-proving techniques, the Prolog program is more concise and has a flavor of describing the problem domain, without directly implementing the search algorithm. The price paid for this conciseness is that much of the program's behavior is hidden, determined by Prolog's built-in inference strategies. Programmers may also feel more pressure to make the problem solution conform to Prolog's representational formalism and search strategies. Lisp, on the other hand, allows greater flexibility for the programmer. The price paid here is that the programmer cannot draw on a built-in representation or search strategy and must implement this explicitly.

In Chapter 14 we present higher-level functions, that is, functions that can take other functions as arguments. This gives the Lisp language much of the representational flexibility that meta-predicates (Chapter 5) give to Prolog.

Exercises

1. Write a random number generator in Lisp. This function must maintain a global variable, seed, and return a different random number each time the function is called. For a description of a reasonable random number algorithm, consult any basic algorithms text.

2. Create an "inventory supply" database. Build type checks for a set of six useful queries on these data tuples. Compare your results with the Prolog approach to this same problem as seen in Chapter 5. 2.

3. Write the functions `initialize`, `push`, `top`, `pop`, and `list-stack` to maintain a global stack. These functions should behave:

```
> (initialize)
nil
> (push 'foo)
foo
> (push 'bar)
bar
> (top)
bar
> (list-stack)
(bar foo)
> (pop)
bar
> (list-stack)
(foo)
> (pop)
foo
> (list-stack)
()
```

4. Sets may be represented using lists. Note that these lists should not contain any duplicate elements. Write your own Lisp implementations of the set operations of union, intersection, and set difference. (Do not use Common Lisp's built-in versions of these functions.)

5. Solve the Water Jug problem, using a production system architecture similar to the Farmer, Wolf, Goat, and Cabbage problem presented in Section 13.2.

> There are two jugs, one holding 3 gallons and the other 5 gallons of water. A number of things that can be done with the jugs: they can be filled, emptied, and dumped one into the other either until the poured-into jug is full or until the poured-out-of jug is empty. Devise a sequence of actions that will produce 4 gallons of water in the larger jug. (Hint: only integer values of water are used.)

6. Implement a depth-first backtracking solution (such as was used to solve the farmer, wolf, goat, and cabbage problem in Section 13.2) to the

missionary and cannibal problem:

> Three missionaries and three cannibals come to the bank of a river they wish to cross. There is a boat that will hold only two people, and any of the group can row it. If there are ever more missionaries than cannibals on any side of the river the cannibals will get converted. Devise a series of moves to get everyone across the river with no conversions.

14 Higher-Order Functions and Flexible Search

Chapter Objectives

Lisp higher-order functions
Lisp functions
- `map`
- `filter`

Lisp functions as arguments of functions
- `funcall`
- `apply`

Designing search algorithms in Lisp
- General production system framework
- Use of open and closed lists

Algorithms designed in Lisp
- Depth-first search
- Breadth-first search
- Best first search
- Programmer implements typing as needed

Chapter Contents

14.1 Higher-Order Functions and Abstraction

One of the most powerful techniques that Lisp and other functional programming languages provide is the ability to define functions that take other functions as parameters or return them as results. These functions are called *higher-order functions* and are an important tool for procedural abstraction.

Maps and Filters

A *filter* is a function that applies a test to the elements of a list, eliminating those that fail the test. `filter-negatives`, presented in Section 12.2, was an example of a filter. *Maps* takes a list of data objects and applies a function to each one, returning a list of the results. This idea may be further generalized through the development of general maps and filters that take as arguments both lists and the functions or tests that are to be applied to their elements.

To begin with an example, recall the function `filter-negatives` from Section 12.2. This function took as its argument a list of numbers and returned that list with all negative values deleted. Similarly, we can define a function to filter out all the even numbers in a list. Because these two functions differ *only* in the name of the predicate used to filter elements from the list, it is natural to think of generalizing them into a single function that takes the filtering predicate as a second parameter:

```
(defun filter-evens (number-list)
   (cond ((null number-list) nil)
         ((oddp (car number-list))
               (cons (car number-list)
                     (filter-evens
                      (cdr number-list))))
         (t (filter-evens (cdr number-list)))))
```

This combination of function applications may be defined using a Lisp form called **funcall**, which takes as arguments a function and a series of arguments and applies that function to those arguments:

```
(defun filter (list-of-elements test)
     (cond ((null list-of-elements) nil)
           ((funcall test (car list-of-elements))
             (cons (car list-of-elements)
                   (filter (cdr list-of-elements)
                      test)))
           (t (filter (cdr list-of-elements)
                      test))))
```

The function, **filter**, applies the test to the first element of the list. If the test returns non-**nil**, it **cons**es the element onto the result of **filter** applied to the **cdr** of the list; otherwise, it just returns the filtered **cdr**. This function may be used with different predicates passed in as parameters to perform a variety of filtering tasks:

```
> (filter '(1 3 -9 5 -2 -7 6) #'plusp)
                    ;Filter out all negative numbers.
(1 3 5 6)
> (filter '(1 2 3 4 5 6 7 8 9) #'evenp)
                          ;Filter out all odd numbers.
(2 4 6 8)
> (filter '(1 a b 3 c 4 7 d) #'numberp)
                          ;Filter out all non-numbers.
(1 3 4 7)
```

When a function is passed as a parameter, as in the above examples, it should be preceded by a **#'** instead of just **'**. The purpose of this convention is to flag arguments that are functions so that they may be given appropriate treatment by the Lisp interpreter. In particular, when a function is passed as an argument in Common Lisp, the bindings of its free variables (if any) must be retained. This combination of function definition and bindings of free variables is called a *lexical closure*; the **#'** informs Lisp that the lexical closure must be constructed and passed with the function. More formally, **funcall** is defined:

```
(funcall <function> <arg1> <arg2> … <argn>)
```

In this definition, **<function>** is a Lisp function and **<arg$_1$> … <arg$_n$>** are zero or more arguments to the function. The result of

evaluating a `funcall` is the same as the result of evaluating `<function>` with the specified arguments as actual parameters.

`apply` is a similar function that performs the same task as `funcall` but requires that its arguments be in a list. Except for this syntactic difference, `apply` and `funcall` behave the same; the programmer can choose the function that seems more convenient for a given application. These two functions are similar to `eval` in that all three of them allow the user to specify that the function evaluation should take place. The difference is that `eval` requires its argument to be an s-expression that is evaluated; `funcall` and `apply` take a function and its arguments as separate parameters. Examples of the behavior of these functions include:

```
> (funcall #'plus 2 3)
5
> (apply #'plus '(2 3))
5
> (eval '(plus 2 3))
5
> (funcall #'car '(a b c))
a
> (apply #'car '((a b c)))
a
```

Another important class of higher-order functions consists of mapping functions, functions that will apply a given function to all the elements of a list. Using `funcall`, we define the simple mapping function `map-simple`, which returns a list of the results of applying a functional to all the elements of a list. It has the behavior:

```
(defun map-simple (func list)
     (cond ((null list) nil)
           (t (cons (funcall func (car list))
                    (map-simple func (cdr list))))))
> (map-simple #'1+ '(1 2 3 4 5 6))
(2 3 4 5 6 7)
> (map-simple #'listp '(1 2 (3 4) 5 (6 7 8)))
(nil nil t nil t)
```

`map-simple` is a simplified version of a Lisp built-in function `mapcar`, that allows more than one argument list, so that functions of more than one argument can be applied to corresponding elements of several lists:

```
> (mapcar #'1+ '(1 2 3 4 5 6))  ;Same as map-simple.
(2 3 4 5 6 7)
> (mapcar #'+ '(1 2 3 4) '(5 6 7 8))
(6 8 10 12)
> (mapcar #'max '(3 9 1 7) '(2 5 6 8))
(3 9 6 8)
```

`mapcar` is only one of many mapping functions provided by Lisp, as well as only one of many higher-order functions built into the language.

Functional Arguments and Lambda Expressions

In the preceding examples, function arguments were passed by their name and applied to a series of arguments. This requires that the functions be previously defined in the global environment. Frequently, however, it is desirable to pass a function definition directly, without first defining the function globally. This is made possible through the `lambda` expression.

Essentially, the `lambda` expression allows us to separate a function definition from the function name. The origin of lambda expressions is in the *lambda calculus*, a mathematical model of computation that provides (among other things) a particularly thoughtful treatment of this distinction between an object and its name. The syntax of a `lambda` expression is similar to the function definition in a `defun`, except that the function name is replaced by the term `lambda`. That is:

```
(lambda (<formal-parameters>) <body>)
```

`Lambda` expressions may be used in place of a function name in a `funcall` or `apply`. The `funcall` will execute the body of the `lambda` expression with the arguments bound to the parameters of the `funcall`. As with named functions, the number of formal parameters and the number of actual parameters must be the same. For example:

```
> (funcall #'(lambda (x) (* x x)) 4)
16
```

Here, `x` is bound to `4` and the body of the `lambda` expression is then evaluated. The result, the square of `4`, is returned by `funcall`. Other examples of the use of `lambda` expressions with `funcall` and `apply` include:

```
> (apply #'(lambda (x y) (+ (* x x) y)) '(2 3))
7
> (funcall #'(lambda (x) (append x x)) '(a b c))
(a b c a b c)
> (funcall #'(lambda (x1 x2)
      (append (reverse x1) x2)) '(a b c) '(d e f))
(c b a d e f)
```

`Lambda` expressions may be used in a higher-order function such as `mapcar` in place of the names of globally defined functions. For example:

```
> (mapcar #'(lambda (x) (* x x)) '(1 2 3 4 5))
(1 4 9 16 25)
> (mapcar #'(lambda (x) (* x 2)) '(1 2 3 4 5))
(2 4 6 8 10)
> (mapcar #'(lambda (x) (and (> x 0) (< x 10)))
          '(1 24 5 -9 8 23))
(t nil t nil t nil)
```

Without `lambda` expressions the programmer must define every function in the global environment using a `defun`, even though that function may be used only once. `Lambda` expressions free the programmer from this

necessity: for example, if it is desired to square each element in a list, the `lambda` form is passed to `mapcar` as the first of the above examples illustrates. It is not necessary to define a squaring function first.

14.2 Search Strategies in Lisp

The use of higher-order functions provides Lisp with a powerful tool for procedural abstraction. In this section, we use this abstraction technique to implement general algorithms for breadth-first, depth-first, and best-first search. These algorithms implement the search algorithms using the open list – the current state list – and the closed list – the already visited states – to manage search through the state space, see Luger (2009, Chapters 3 and 4) and Chapter 4 of this book for similar search algorithms in Prolog.

Breadth-First and Depth-First Search

The Lisp implementation of breadth-first search maintains the open list as a first-in-first-out (FIFO) structure. We will define open and closed as global variables. This is done for several reasons: first to demonstrate the use of global structures in Lisp; second, to contrast the Lisp solution with that in Prolog; and third, it can be argued that since the primary task of this program is to solve a search problem, the state of the search may be represented globally. Finally, since open and closed may be large, their use as global variables seems justified. General arguments of efficiency for the local versus the global approach often depend on the implementation details of a particular language. Global variables in Common Lisp are written to begin and end with `*`. Breadth-first search may be defined:

```
(defun breadth-first ( )
     (cond ((null *open*) nil)
           (t (let ((state (car *open*)))
              (cond ((equal state *goal*) 'success)
                    (t (setq *closed* (cons state
                                            *closed*))
                       (setq *open* (append
                                     (cdr *open*)
                          generate-descendants
                                state *moves*)))
                       (breadth-first)))))))
(defun run-breadth (start goal)
     (setq *open* (list start))
     (setq *closed* nil)
     (setq *goal* goal)
     (breadth-first))
```

In our implementation, the `*open*` list is tested: if it is `nil`, the algorithm returns `nil`, indicating failure as there are no more states to evaluste; If `*open*` is not `nil`, it examines the first element of `*open*`. If this is `equal` to the `goal`, the algorithm halts and returns `success`; otherwise, it calls `generate-descendants` to produce the children of the current `state`, adds them to the `*open*` list, and recurs. `run-breadth` is an initialization function that sets the initial values of `*open*`, `*closed*`, and `*goal*`. `generate-descendants` is passed both the `state`

and *moves* as parameters. *moves* is a list of the functions that generate moves. In the farmer, wolf, goat, and cabbage problem, assuming the move definitions of Section 13.2, *moves* would be:

```
(setq *moves*
      '(farmer-takes-self farmer-takes-wolf
        farmer-takes-goat farmer-takes-cabbage))
```

generate-descendants takes a state and returns a list of its children. In addition to generating child states, it disallows duplicates in the list of children and eliminates any children that are already in the *open* or *closed* list. In addition to the state, generate-descendants is given a list of moves; these may be the names of defined functions, or they may be lambda definitions. generate-descendants uses a let block to save the result of a move in the local variable child. We define generate-descendants:

```
(defun generate-descendants (state moves)
      (cond ((null moves) nil)
            (t (let ((child (funcall (car moves)
                                     state))
                     (rest (generate-descendants state
                                    (cdr moves))))
                 (cond ((null child) rest)
                       ((member child rest :test
                                    #'equal) rest)
                       ((member child *open* :test
                                    #'equal) rest)
                       ((member child *closed* :test
                                    #'equal) rest)
                       (t (cons child rest)))))))
```

As first noted in Section 13.2, the calls to the member function use an additional parameter, :test #'equal. The member function allows the user to specify any test for membership. This allows us to use predicates of arbitrary complexity and semantics to test membership. Though Lisp does not require that we specify the test, the default comparison is the predicate eq. eq requires that two objects be identical, which means they have the same location in memory; we are using a weaker comparison, equal, that only requires that the objects have the same value. By binding the global variable *moves* to an appropriate set of move functions, the search algorithm just presented may be used to search any state space graph in a breadth-first fashion.

One difficulty that remains with this implementation is its inability to print the list of states along the path from a start to a goal. Although all the states that lead to the goal are present in the closed list when the algorithm halts, these are mixed with all other states from earlier levels of the search space. We can solve this problem by recording both the state and its parent, and reconstructing the solution path from this information. For example, if the state (e e e e) generates the state (w e w e), a record of both states, ((w e w e) (e e e e)), is placed on

open. Later, after the children of the state have been generated, the same (<state> <parent>) pair is placed on *closed*.

When the current state equals the goal, the ancestor information is used to build the path from the goal to the start state by going back to successive parents. This augmented version of breadth-first search begins by defining state records as an abstract data type:

```
(defun build-record (state parent)
            (list state parent))
(defun get-state (state-tuple) (nth 0 state-tuple))
(defun get-parent (state-tuple) (nth 1 state-tuple))
(defun retrieve-by-state (state list)
     (cond ((null list) nil)
           ((equal state (get-state (car list)))
                     (car list))
           (t (retrieve-by-state state
                     (cdr list)))))
```

build-record constructs a (<state> <parent>) pair. get-state and get-parent access the appropriate fields of a record. retrieve-by-state takes a state and a list of state records and returns the record whose state field matches that state.

build-solution uses retrieve-by-state to chain back from state to parent, constructing a list of successive states that led to a goal. When initializing *open*, we will give the starting state a parent of nil; build-solution stops when passed a null state.

```
(defun build-solution (state)
     (cond ((null state) nil)
         (t (cons state (build-solution (get-parent
           (retrieve-by-state state *closed*)))))))
```

The remainder of the algorithm is similar to the breadth-first search of Section 3.2:

```
(defun run-breadth (start goal)
     (setq *open* (list (build-record start nil)))
     (setq *closed* nil)
     (setq *goal* goal)
     (breadth-first))
(defun breadth-first ( )
     (cond ((null *open*) nil)
           (t (let ((state (car *open*)))
              (setq *closed* (cons state *closed*))
                (cond ((equal (get-state state)
                              *goal*)
                       (build-solution *goal*))
                      (t (setq *open* (append (cdr
                              *open*)
                         (generate-descendants
```

```
                                    (get-state state)
                                    *moves*)))
                              (breadth-first)))))))
(defun generate-descendants (state moves)
      (cond ((null moves) nil)
            (t (let ((child (funcall
                              (car moves) state))
                     (rest (generate-descendants
                           state (cdr moves))))
               (cond ((null child) rest)
                  ((retrieve-by-state child rest)
                          rest)
                  ((retrieve-by-state child *open*)
                          rest)
                  ((retrieve-by-state child
                          *closed*) rest)
                  (t (cons (build-record child
                                                state)
                           rest)))))))
```

Depth-first search is implemented by modifying breadth-first search to maintain `*open*` as a stack. This simply involves reversing the order of the arguments to `append`.

Best-First Search

Best-first search may be implemented through straightforward modifications to the breadth-first search algorithm. Specifically, the heuristic evaluation is saved along with each state. The tuples on `*open*` are then sorted according to this evaluation. The data type definitions for state records are an extension of those used in breadth-first search:

```
(defun build-record (state parent depth weight)
   (list state parent depth weight))
(defun get-state (state-tuple) (nth 0 state-tuple))
(defun get-parent (state-tuple) (nth 1 state-tuple))
(defun get-depth (state-tuple) (nth 2 state-tuple))
(defun get-weight (state-tuple) (nth 3 state-tuple))
(defun retrieve-by-state (state list)
     (cond ((null list) nil)
           ((equal state (get-state (car list)))
                 (car list))
           (t (retrieve-by-state state
                       (cdr list)))))
```

`best-first` and `generate-descendants` are defined:

```
(defun best-first ( )
      (cond ((null *open*) nil)
            (t (let ((state (car *open*)))
               (setq *closed* (cons state *closed*))
```

```
            (cond ((equal (get-state state)
                                        *goal*)
                       (build-solution *goal*))
                  (t (setq *open*
                       (insert-by-weight
                          (generate-descendants
                                (get-state state)
                             (+ 1 (get-depth
                                state))
                                *moves*)
                              (cdr *open*)))
            (best-first)))))))

(defun generate-descendants (state depth moves)
   (cond ((null moves) nil)
      (t (let ((child (funcall (car moves) state))
               (rest (generate-descendants state
                           depth (cdr moves))))
            (cond ((null child) rest)
               ((retrieve-by-state child rest)
                    rest)
               ((retrieve-by-state child *open*)
                    rest)
               ((retrieve-by-state child *closed*)
                    rest)
               (t (cons (build-record child state
                           depth (+ depth (heuristic
                                       child)))
                        rest)))))))
```

The only differences between `best-first` and `breadth-first` search are the use of `insert-by-weight` to sort the records on `*open*` by their heuristic weights and the computation of search depth and heuristic weights in `generate-descendants`.

Completion of `best-first` requires a definition of `insert-by-weight`. This function takes an unsorted list of state records and inserts them, one at a time, into their appropriate positions in `*open*`. It also requires a problem-specific definition of a function `heuristic`. This function takes a state and, using the global `*goal*`, computes a heuristic weight for that state. We leave the creation of these functions as an exercise for the reader.

Exercises

1. Create a type check that prevents the member check predicate (that checks whether an item is a member of a list of items) from crashing when called on `member(a, a)`. Will this "fix" address the `append(nil, 6, 6)` anomaly that is described in Chapter 10? Test it and determine your success.

2. Implement `build-solution` and `eliminate-duplicates` for the breadth-first search algorithm of Section 14.2.

3. Create a depth-first, a breadth-first, and best first search for the Water Jugs problem (Chapter 13, number 5). This will require you to create a heuristic measure for the Water Jugs problem, as well as create an `insert-by-weight` function for maintaining the priority queue.

4. Create a depth-first, a breadth-first, and best first search for the Missionaries and Cannibals problem (Chapter 13, number 6). This will require you to create a heuristic measure for the Missionaries and Cannibals problem, as well as create an `insert-by-weight` function for maintaining the priority queue.

5. Write a Lisp program to solve the 8-queens problem. (This problem is to find a way to place eight queens on a chessboard so that no queen may capture any other through a single move, i.e., no two queens are on the same row, column, or diagonal.) Do depth-first, breadth-first, and best-first solutions to this problem.

6. Write a Lisp program to solve the full 8 x 8 version of the Knight's Tour problem. This problem asks you to find a path from any square to any other square on the chessboard, using only the knight. Do a depth-first, breadth-first, and best-first solutions for this problem.

15 Unification and Embedded Languages in Lisp

Chapter Objectives

Pattern matching in Lisp:
- Database examples

Full unification as required for Predicate Calculus problem solving
- Needed for applying inference rules
- General structure mapping
- Recursive for embedded structures

Building interpreters and embedded languages
- Example: `read-eval-print` loop

Example: infix interpreter

Chapter Contents

15.1 Pattern Matching: Introduction

In Chapter 15 we first design an algorithm for matching patterns in general list structures. This is the basis for the unify function which supports full pattern matching and the return of sets of unifying substitutions for matching patterns in predicate calculus expressions. We will see this as the basis for interpreters for logic programming and rule-based systems in Lisp, presented in Chapters 16 and 17.

Pattern Matching in Lisp

Pattern matching is an important AI methodology that has already been discussed in the Prolog chapters and in the presentation of production systems. In this section we implement a recursive pattern matcher and use it to build a pattern-directed retrieval function for a simple database.

The heart of this retrieval system is a function called `match`, which takes as arguments two s-expressions and returns `t` if the expressions match. Matching requires that both expressions have the same *structure* as well as having identical atoms in corresponding positions. In addition, match allows the inclusion of variables, denoted by `?`, in an s-expression. Variables are allowed to match with any s-expression, either a list or an atom, but do not save bindings, as with full unification (next). Examples of the desired behavior for `match` appear below. If the examples seem reminiscent of the Prolog examples in Part II, this is because `match` is actually a simplified version of the unification algorithm that forms the heart of the Prolog environment, as well as of many other pattern-directed AI systems. We will later expand `match` into the full unification algorithm by allowing named variables and returning a list of bindings required for a match.

```
> (match '(likes bill wine) '(likes bill wine))
t
> (match '(likes bill wine) '(likes bill milk))
nil
> (match '(likes bill ?) '(likes bill wine))
t
> (match '(likes ? wine) '(likes bill ?))
t
> (match '(likes bill ?) '(likes bill (prolog lisp
smalltalk))
t
> (match '(likes ?) '(likes bill wine))
nil
```

match is used to define a function called **get-matches**, which takes as arguments two s-expressions. The first argument is a pattern to be matched against elements of the second s-expression, which must be a list. **get-matches** returns a list of the elements of the list that match the first argument. In the example below, **get-matches** is used to retrieve records from an employee database as described earlier in Part III.

Because the database is a large and relatively complex s-expression, we have bound it to the global variable ***database*** and use that variable as an argument to **get-matches**. This was done to improve readability of the examples.

```
> (setq *database* '(((lovelace ada) 50000.00 1234)
          ((turing alan) 45000.00 3927)
          ((shelley mary) 35000.00 2850)
          ((vonNeumann john) 40000.00 7955)
          ((simon herbert) 50000.00 1374)
          ((mccarthy john) 48000.00 2864)
          ((russell bertrand) 35000.00 2950))
*database*
> (get-matches '((turing alan) 45000.00 3927)
*database*)
((turing alan) 45000.00 3927)
> (get-matches '(? 50000.00 ?) *database*)
;people who make 50000
(((lovelace ada) 50000.00 1234) ((simon herbert)
50000.00 1374))
> (get-matches '((? john) ? ?) *database*)
;all people named john
(((vonNeumann john) 40000.00 7955) ((mccarthy john)
48000.00 2864))
```

We implement **get-matches** using **cdr** recursion: each step attempts to match the target pattern with the first element of the database (the **car** of the list). If the is a match, the function will **cons** it onto the list of

matches returned by the recursive call to form the answer for the pattern. `get-matches` is defined:

```
(defun get-matches (pattern database)
   (cond ((null database) ( ))
      ((match pattern (car database))
         (cons (car database)
               (get-matches pattern
                  (cdr database))))
           (t (get-matches pattern
                  (cdr database)))))
```

The heart of the system is the `match` function, a predicate that determines whether or not two s-expressions containing variables actually match. `match` is based on the idea that two lists match if and only if their respective `car`s and `cdr`s match, suggesting a `car-cdr` recursive scheme for the algorithm.

The recursion terminates when either of the arguments is atomic (this includes the empty list, `nil`, which is both an atom and a list). If both patterns are the same atom or if one of the patterns is a variable atom, `?`, which can match with anything, then termination is with a successful match; otherwise, the match will fail. Notice that if either of the patterns is a variable, the other pattern need not be atomic; variables may match with variables or with s-expressions of arbitrary complexity.

Because the handling of the terminating conditions is complex, the implementation of `match` uses a function called `match-atom` that takes two arguments, one or both of which is an `atom`, and checks to see whether the patterns match. By hiding this complexity in `match-atom` the `car-cdr` recursive structure of match is more apparent:

```
(defun match (pattern1 pattern2)
   (cond (or (atom pattern1) (atom pattern2))
             (match-atom pattern1 pattern2))
      (t (and (match (car pattern1) (car pattern2))
              (match (cdr pattern1)
                     (cdr pattern2))))))
```

The implementation of `match-atom` makes use of the fact that when it is called, at least one of the arguments is an `atom`. Because of this assumption, a simple test for equality of patterns is all that is needed to test that both patterns are the same `atom` (including both being a variable); it will fail either if the two patterns are different atoms or if one of them is nonatomic. If the first test fails, the only way match can succeed is if one of the patterns is a variable. This check constitutes the remainder of the function definition.

Finally, we define a function `variable-p` to test whether or not a pattern is a variable. Treating variables as an abstract data type now will simplify later extensions to the function, for example, the extension of the function to named variables as in Prolog.

```
(defun match-atom (pattern1 pattern2)
     (or (equal pattern1 pattern2)
         (variable-p pattern1)
         (variable-p pattern2)))
(defun variable-p (x) (equal x '?))
```

A Recursive Unification Function

We have just completed the implementation of a recursive pattern-matching algorithm that allowed the inclusion of unnamed variables in patterns. Our next step will be to extend this simple pattern matcher into the full unification algorithm. See Luger (2009, Section 2.3) for a pseudocode version of this algorithm.

The function, `unify`, allows named variables in both of the patterns to be matched, and returns a substitution list of the variable bindings required for the match. This unification function is the basis of the inference systems for logic and expert system interpreters developed later in Chapters 16 and 17.

As follows the definition of unification, patterns are either constants, variables, or list structures. We will distinguish variables from one another by their names. Named variables will be represented as lists of the form `(var <name>)`, where `<name>` is usually an atomic symbol. `(var x)`, `(var y)`, and `(var newstate)` are all examples of legal variables.

The function `unify` takes as arguments two patterns to be matched and a set of variable substitutions (bindings) to be employed in the match. Generally, this set will be empty (`nil`) when the function is first called. On a successful match, `unify` returns a (possibly empty) set of substitutions required for a successful match. If no match was possible, `unify` returns the symbol `failed`; `nil` is used to indicate an empty substitution set, i.e., a match in which no substitutions were required. An example of the behavior of `unify`, with comments, is:

```
> (unify '(p a (var x)) '(p a b) ( ))
(((var x) . b))
             ;Returns substitution of b for (var x).
> (unify '(p (var y) b) '(p a (var x)) ( ))
(((var x) . b) ((var y) . a))
                 ;Variables appear in both patterns.
> (unify '(p (var x)) '(p (q a (var y))) ( ))
(((var x) q a (var y)))
            ;Variable is bound to a complex pattern.
> (unify '(p a) '(p a) ( ))
nil
            ;nil indicates no substitution required.
> (unify '(p a) '(q a) ( ))
failed
         ;Returns atom "failed" to indicate failure.
```

We will explain the "." notation, as in `((var x) . b)`, after we present the function `unify`. `unify`, like the pattern matcher of earlier in this section, uses a `car-cdr` recursive scheme and is defined by:

```
(defun unify (pattern1 pattern2 substitution-list)
     (cond ((equal substitution-list 'failed)
              'failed)
           ((varp pattern1)
            (match-var pattern1
                    pattern2 substitution-list))
           ((varp pattern2)
            (match-var pattern2
                    pattern1 substitution-list))
           ((is-constant-p pattern1)
            (cond ((equal pattern1 pattern2)
                    substitution-list)
                  (t 'failed)))
           ((is-constant-p pattern2) 'failed)
           (t (unify (cdr pattern1)
                 (cdr pattern2)
                 (unify (car pattern1)
                   (car pattern2)
                      substitution-list)))))
```

On entering `unify`, the algorithm first checks whether the `substitution-list` is `equal` to `failed`. This could occur if a prior attempt to unify the `car`s of two patterns had failed. If this condition is met, the entire unification operation fails, and the function returns `failed`.

Next, if either pattern is a variable, the function `match-var` is called to perform further checking and possibly add a new binding to `substitution-list`. If neither pattern is a variable, `unify` tests whether either is a constant, returning the unchanged substitution list if they are the same constant, otherwise it returns `failed`.

The last item in the `cond` statement implements the tree-recursive decomposition of the problem. Because all other options have failed, the function concludes that the patterns are lists the must be unified recursively. It does this using a standard tree-recursive scheme: first, the `car`s of the patterns are unified using the bindings in `substitution-list`. The result is passed as the third argument to the call of `unify` on the `cdr`s of both patterns. This allows the variable substitutions made in matching the `car`s to be applied to other occurrences of those variables in the `cdr`s of both patterns.

`match-var`, for the case of matching a variable and a pattern, is defined:

```
(defun match-var (var pattern substitution-list)
   (cond ((equal var pattern) substitution-list)
         (t (let ((binding
                   (get-binding var substitution-list)))
              (cond (binding (unify
                              (get-binding-value binding)
                               pattern substitution-list))
                    ((occursp var pattern) 'failed)
                     (t (add-substitution var pattern
                                          substitution-list)))))))
```

match-var first checks whether the variable and the pattern are the same; unifying a variable with itself requires no added substitutions, so substitution-list is returned unchanged.

If `var` and `pattern` are not the same, `match-var` checks whether the variable is already bound. If a binding exists, `unify` is called recursively to match the value of the binding with `pattern`. Note that this binding value may be a constant, a variable, or a pattern of arbitrary complexity; requiring a call to the full unification algorithm to complete the match.

If no binding currently exists for `var`, the function calls `occursp` to test whether `var` appears in `pattern`. As explained in (Luger 2009), the occurs check is needed to prevent attempts to unify a variable with a pattern containing that variable, leading to a circular structure. For example, if `(var x)` was bound to `(p (var x))`, any attempt to apply those substitutions to a pattern would result in an infinite structure. If `var` appears in pattern, `match-var` returns `failed`; otherwise, it adds the new substitution pair to `substitution-list` using `add-substitution`.

`unify` and `match-var` are the heart of the unification algorithm. `occursp` (which performs a tree walk on a pattern to find any occurrences of the variable in that pattern), `varp`, and `is-constant-p` (which test whether their argument is a variable or a constant, respectively) appear below. Functions for handling substitution sets are discussed below.

```
(defun occursp (var pattern)
   (cond ((equal var pattern) t)
      ((or (varp pattern)
           (is-constant-p pattern))
        nil)
      (t (or (occursp var (car pattern))
             (occursp var (cdr pattern))))))

(defun is-constant-p (item)
     (atom item))
```

```
(defun varp (item)
     (and (listp item)
          (equal (length item) 2)
          (equal (car item) 'var)))
```

Sets of substitutions are represented using a built-in Lisp data type called the *association list* or *a-list*. This is the basis for the functions **add-substitutions**, **get-binding**, and **binding-value**. An association list is a list of data records, or *key/data* pairs. The **car** of each record is a *key* for its retrieval; the **cdr** of each record is called the *datum*. The datum may be a list of values or a single atom. Retrieval is implemented by the function **assoc**, which takes as arguments a key and an association list and returns the first member of the association list that has the key as its **car**. An optional third argument to **assoc** specifies the test to be used in comparing keys. The default test is the Common Lisp function **eql**, a form of equality test requiring that two arguments be the same object (i.e., either the same memory location or the same numeric value). In implementing substitution sets, we specify a less strict test, **equal**, which requires only that the arguments match syntactically (i.e., are designated by identical names). An example of **assoc**'s behavior appears next:

```
> (assoc 3 '((1 a) (2 b) (3 c) (4 d)))
(3 c)
> (assoc 'd '((a b c) (b c d e) (d e f) (c d e))
:test #'equal)
(d e f)
> (assoc 'c '((a . 1) (b . 2) (c . 3) (d . 4)) :test
#'equal)
(c . 3)
```

Note that **assoc** returns the entire record matched on the key; the datum may be retrieved from this list by the **cdr** function. Also, notice that in the last call the members of the a-list are not lists but a structure called *dotted pairs*, e.g., **(a . 1)**.

The dotted pair, or **cons** pair, is actually the fundamental constructor in Lisp. It is the result of **cons**ing one s-expression onto another; the list notation that we have used throughout the chapter is just a notational variant of dotted pairs. For example, the value returned by **(cons 1 nil)** is actually **(1 . nil)**; this is equivalent to **(1)**. Similarly, the list **(1 2 3)** may be written in dotted pair notation as **(1 . (2 . (3 . nil)))**. Although the actual effect of a **cons** is to create a dotted pair, the list notation is cleaner and is generally preferred.

If two atoms are **cons**ed together, the result is always written using dotted pair notation. The **cdr** of a dotted pair is the second element in the pair, rather than a list containing the second atom. For example:

```
> (cons 'a 'b)
(a . b)
```

```
> (car '(a . b))
a
> (cdr '(a . b))
b
```

Dotted pairs occur naturally in association lists when one atom is used as a key for retrieving another atom, as well as in other applications that require the formation and manipulation of pairs of atomic symbols. Because unifications often substitute a single atom for a variable, dotted pairs appear often in the association list returned by the unification function.

Along with `assoc`, Common Lisp defines the function `acons`, which takes as arguments a key, a datum, and an association list and returns a new association list whose first element is the result of `cons`ing the key onto the datum. For example:

```
> (acons 'a 1 nil)
((a . 1))
```

Note that when `acons` is given two atoms, it adds their `cons` to the association list:

```
> (acons 'pets '(emma jack clyde)
     '((name . bill) (hobbies music skiing movies)
         (job . programmer)))
((pets emma jack clyde) (name . bill) (hobbies music
skiing movies)(job . programmer))
```

The members of an association list may themselves be either dotted pairs or lists.

Association lists provide a convenient way to implement a variety of tables and other simple data retrieval schemes. In implementing the unification algorithm, we use association lists to represent sets of substitutions: the keys are the variables, and the data are the values of their bindings. The datum may be a simple variable or constant or a more complicated structure.

Using association lists, the substitution set functions are defined:

```
(defun get-binding (var substitution-list)
     (assoc var substitution-list :test #'equal))

(defun get-binding-value (binding) (cdr binding))

(defun add-substitution (var pattern
           substitution-list)
     (acons var pattern substitution-list))
```

This completes the implementation of the unification algorithm. We will use the unification algorithm again in Section 15.1 to implement a simple Prolog in Lisp interpreter, and again in Section 16.2 to build an expert system shell.

15.2 Interpreters and Embedded Languages

The top level of the Lisp interpreter is known as the *read-eval-print* loop. This describes the interpreter's behavior in reading, evaluating, and printing the value of s-expressions entered by the user. The `eval` function, defined in Section 11.2, is the heart of the Lisp interpreter; using `eval`, it is possible to write Lisp's top-level `read-eval-print` loop in Lisp itself. In the next example, we develop a simplified version of this loop. This version is simplified chiefly in that it does not have the error-handling abilities of the built-in loop, although Lisp does provide the functionality needed to implement such capabilities.

To write the `read-eval-print` loop, we use two more Lisp functions, `read` and `print`. `read` is a function that takes no parameters; when it is evaluated, it returns the next s-expression entered at the keyboard. `print` is a function that takes a single argument, evaluates it, and then prints that result to standard output. Another function that will prove useful is `terpri`, a function of no arguments that sends a newline character to standard output. `terpri` also returns a value of `nil` on completion. Using these functions, the `read-eval-print` loop is based on a nested s-expression:

```
(print (eval (read)))
```

When this is evaluated, the innermost s-expression, `(read)`, is evaluated first. The value returned by the `read`, the next s-expression entered by the user, is passed to `eval`, where it is evaluated. The result of this evaluation is passed to `print`, where it is sent to the display screen. To complete the loop we add a `print` expression to output the prompt, a `terpri` to output a newline after the result has been printed, and a recursive call to repeat the cycle. Thus, the final `read-eval-print` loop is defined:

```
(defun my-read-eval-print ( )
     (print ':)  ;output a prompt (":")
     (print (eval (read)))
     (terpri) ;output a newline
     (my-read-eval-print))   ;do it all again
```

This may be used "on top of" the built-in interpreter:

```
> (my-read-eval-print)
:(+ 1 2);note the alternative prompt
3
:                                                    ;etc
```

As this example illustrates, by making functions such as `quote` and `eval` available to the user, Lisp gives the programmer a high degree of control over the handling of functions. Because Lisp programs and data are both represented as s-expressions, we may write programs that perform any desired manipulations of Lisp expressions prior to evaluating them. This underlies much of Lisp's power as an imperative representation language because it allows arbitrary Lisp code to be stored, modified, and evaluated when needed. It also makes it simple to write specialized interpreters that

may extend or modify the behavior of the built-in Lisp interpreter in some desired fashion. This capability is at the heart of many Lisp-based expert systems, which read user queries and respond to them according to the expertise contained in their knowledge base.

As an example of the way in which such a specialized interpreter may be implemented in Lisp, we modify `my-read-eval-print` so that it evaluates arithmetic expressions in an infix rather than a prefix notation, as we see in the following example (note the modified prompt, `infix->`):

```
infix-> (1 + 2)
3
infix-> (7 – 2)
5
infix-> ((5 + 2) * (3 – 1))   ;Loop handles nesting.
15
```

To simplify the example, the infix interpreter handles only arithmetic expressions. A further simplification restricts the interpreter to binary operations and requires that all expressions be fully parenthesized, eliminating the need for more sophisticated parsing techniques or worries about operator precedence. However, it does allow expressions to be nested to arbitrary depth and handles Lisp's binary arithmetic operators.

We modify the previously developed `read-eval-print` loop by adding a function that translates infix expressions into prefix expressions prior to passing them on to `eval`. A first attempt at writing this function might look like:

```
(defun simple-in-to-pre (exp)
      (list (nth 1 exp)
                ;Middle element becomes first element.
      (nth 0 exp)
      ;first operand
      (nth 2 exp)
      ;second operand
```

`simple-in-to-pre` is effective in translating simple expressions; however, it is not able to correctly translate nested expressions, that is, expressions in which the operands are themselves infix expressions. To handle this situation properly, the operands must also be translated into prefix notation. Recursion is halted by testing the argument to determine whether it is a number, returning it unchanged if it is. The completed version of the `infix-to-prefix` translator is:

```
(defun in-to-pre (exp)
   (cond ((numberp exp) exp)
         (t (list (nth 1 exp)
                  (in-to-pre (nth 0 exp))
                (in-to-pre (nth 2 exp))))))
```

Using this translator, the `read-eval-print` loop may be modified to interpret infix expressions, as defined next:

```
(defun in-eval ( )
      (print 'infix->)
      (print (eval (in-to-pre (read))))
      (terpri)
      (in-eval))
```

This allows the interpretation of binary expressions in infix form:

```
> (in-eval)
infix->(2 + 2)
4
infix->((3 * 4) – 5)
7
```

In the above example, we have implemented a new language in Lisp, the language of infix arithmetic. Because of the facilities Lisp provides for symbolic computing (lists and functions for their manipulation) along with the ability to control evaluation, this was much easier to do than in many other programming languages. This example illustrates an important AI programming methodology, that of *meta-linguistic abstraction*.

Very often in AI programming, a problem is not completely understood, or the program required to solve a problem is extremely complex. *Meta-linguistic abstraction* uses the underlying programming language, in this case, Lisp, to implement a specialized, high-level language that may be more effective for solving a particular class of problems. The term "meta-linguistic abstraction" refers to our use of the base language to implement this other programming language, rather than to directly solve the problem. As we saw in Chapter 5, Prolog also gives the programmer the power to create meta-level interpreters. The power of meta-interpreters to support programming in complex domains was discussed in Part I.

Exercises

1. Newton's method for solving roots takes an estimate of the value of the root and tests it for accuracy. If the guess does not meet the required tolerance, it computes a new estimate and repeats. Pseudo-code for using Newton's method to get the square root of a number is:

```
function root-by-newtons-method (x, tolerance)
guess := 1;
repeat
      guess := 1/2(guess + x/guess)
until absolute-value(x – guess  guess) < tolerance
```

Write a recursive Lisp function to compute square roots by Newton's method.

2. Write a random number generator in Lisp. This function must maintain a global variable, seed, and return a different random number each time the function is called. For a description of a reasonable random number algorithm, consult any basic algorithms text.

3. Test the `unify` form of Section 15.1 with five different examples of your own creation.

4. Test the `occursp` form of Section 15.1 on five different examples of your own creation

5. Write a binary post-fix interpreter that takes arbitrarily complex structures in post-fix form and evaluates them. Two examples of post-fix are `(3 4 +)` and `(6 (5 4 +) *)`.

16 Logic Programming in Lisp

Chapter Objectives

A Lisp-based logic programming interpreter:
- An example of meta-linguistic abstraction

Critical components of logic interpreter
Predicate Calculus like facts and rules
Horn clause form
Queries processed by unification against facts and rules
Successful goal returns unification substitutions
Supporting technology for logic interpreter
- Streams
- Stream processing
- Stream of variables substitutions filtered through conjunctive subgoals
- `gensym` used to standardize variables apart

Exercises expanding functionality of logic interpreter
- Adding `and`, `not`
- Additions of numeric and equality relations

Chapter Contents

16.1 A Simple Logic Programming Language

Example

As an example of meta-linguistic abstraction, we develop a Lisp-based logic programming interpreter, using the unification algorithm from Section 15.2. Like Prolog, our logic programs consist of a database of facts and rules in the predicate calculus. The interpreter processes queries (or goals) by unifying them against entries in the logic database. If a goal unifies with a simple fact, it succeeds; the solution is the set of bindings generated in the match. If it matches the head of a rule, the interpreter recursively attempts to satisfy the rule premise in a depth-first fashion, using the bindings generated in matching the head. On success, the interpreter prints the original goal, with variables replaced by the solution bindings.

For simplicity's sake, this interpreter supports conjunctive goals and implications: or and not are not defined, nor are features such as arithmetic, I/O, or the usual Prolog built-in predicates. Although we do not implement full Prolog, and the exhaustive nature of the search and absence of the *cut* prevent the proper treatment of recursive predicates, the shell captures the basic behavior of the logic programming languages. The addition to the interpreter of the other features just mentioned is an interesting exercise.

Our logic programming interpreter supports Horn clauses, a subset of full predicate calculus (Luger 2009, Section 14.2). Well-formed formulae consist of terms, conjunctive expressions, and rules written in a Lisp-

oriented syntax. A compound term is a list in which the first element is a predicate name and the remaining elements are the arguments. Arguments may be either constants, variables, or other compound terms. As in the discussion of `unify`, we represent variables as lists of two elements, the word `var` followed by the name of the variable. Examples of terms include:

```
(likes bill music)
(on block (var x))
(friend bill (father robert))
```

A conjunctive expression is a list whose first element is `and` and whose subsequent arguments are either simple terms or conjunctive expressions:

```
(and (smaller david sarah) (smaller peter david))
(and (likes (var x) (var y))
   (likes (var z) (var y)))
(and (hand-empty)
   (and (on block-1 block-2)
      (on block-2 table)))
```

Implications are expressed in a syntactically sweetened form that simplifies both their writing and recognition:

```
(rule if <premise> then <conclusion>)
```

where `<premise>` is either a simple or conjunctive proposition and `<conclusion>` is always a simple proposition. Examples include:

```
(rule if (and (likes (var x) (var z))
              (likes (var y) (var z)))
     then (friend (var x) (var y)))
(rule if (and (size (var x) small)
              (color (var x) red)
              (smell (var x) fragrant))
     then (kind (var x) rose))
```

The logic database is a list of facts and rules bound to a global variable, `*assertions*`. We can define an example knowledge base of `likes` relationships by a call to `setq` (we could have used the function `defvar`):

```
(setq *assertions*
     '((likes george beer)
       (likes george kate)
       (likes george kids)
       (likes bill kids)
       (likes bill music)
       (likes bill pizza)
       (likes bill wine)
       (rule if (and (likes (var x) (var z))
                     (likes (var y) (var z)))
            then (friend (var x) (var y)))))
```

The top level of the interpreter is a function, `logic-shell`, that reads goals and attempts to satisfy them against the logic database bound to `*assertions*`. Given the above database, `logic-shell` will have the following behavior, where comments follow the `;`:

```
> (logic-shell)                          ;Prompts with a ?
?(likes bill (var x))
(likes bill kids)
(likes bill music)
(likes bill pizza)
(likes bill wine)
?(likes george kate)
(likes george kate)
?(likes george taxes)                    ;Failed query returns nothing.
?(friend bill george)
(friend bill george)      ;From (and(likes bill kids)
                                ;(likes george kids)).
?(friend bill roy)              ;roy not in knowledge base, fail.
?(friend bill (var x))
(friend bill george)       ;From (and(likes bill kids)
                                 (likes george kids)).
(friend bill bill)         ;From (and(likes bill kids)
                                 ;(likes bill kids)).
(friend bill bill)        ;From (and(likes bill music)
                                 ;(likes bill music)).
(friend bill bill)        ;From (and(likes bill pizza)
                                 ;(likes bill pizza)).
(friend bill bill)         ;From (and(likes bill wine)
                                 ;(likes bill wine)).
?quit
bye
>
```

Before discussing the implementation of the logic programming interpreter, we introduce the *stream* data type.

16.2 Streams and Stream Processing

As the preceding example suggests, even a small knowledge base can produce complex behaviors. It is necessary not only to determine the truth or falsity of a goal but also to determine the variable substitutions that make that goal be true in the knowledge base. A single goal can match with different facts, producing different substitution sets; conjunctions of goals require that all conjuncts succeed and also that the variable bindings be consistent throughout. Similarly, rules require that the substitutions formed in matching a goal with a rule conclusion be made in the rule premise when it is solved. The management of these multiple substitution sets is the major source of complexity in the interpreter. Streams help address this

complexity by focusing on the movement of a sequence of candidate variable substitutions through the constraints defined by the logic database.

A *stream* is a sequence of data objects. Perhaps the most common example of stream processing is a typical interactive program. The data from the keyboard are viewed as an endless sequence of characters, and the program is organized around reading and processing the *current* character from the input stream. Stream processing is a generalization of this idea: streams need not be produced by the user; they may also be generated and modified by functions. A *generator* is a function that produces a continuing stream of data objects. A *map function* applies some function to each of the elements of a stream. A *filter* eliminates selected elements of a stream according to the constraints of some predicate.

The solutions returned by an inference engine may be represented as a stream of different variable substitutions under which a goal follows from a knowledge base. The constraints defined by the knowledge base are used to modify and filter a stream of candidate substitutions, producing the result. Consider, for example, the conjunctive goal (using the logic database from the preceding section):

```
(and (likes bill (var z))
     (likes george (var z)))
```

The stream-oriented view regards each of the conjuncts in the expression as a *filter* for a stream of substitution sets. Each set of variable substitutions in the stream is applied to the conjunct and the result is matched against the knowledge base. If the match fails, that set of substitutions is eliminated from the stream; if it succeeds, the match may create new sets of substitutions by adding new bindings to the original substitution set.

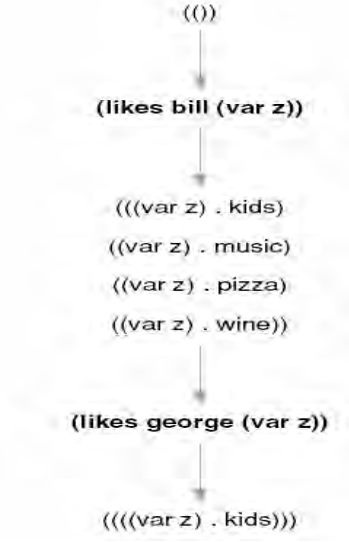

Figure 16.1 A stream of variable substitutions filtered through conjunctive subgoals.

Figure 16.1 illustrates the stream of substitutions passing through this conjunctive goal. It begins with a stream of candidate substitutions containing only the empty substitution set and grows after the first proposition matches against multiple entries in the database. It then shrinks to a single substitution set as the second conjunct eliminates substitutions that do not allow `(likes`

`george (var z))` to succeed. The resulting stream, `((((var z) . kids)))`, contains the only variable substitution that allows both subgoals in the conjunction to succeed in the knowledge base.

As this example illustrates, a goal and a single set of substitutions may generate several new substitution sets, one for each match in the knowledge base. Alternatively, a goal will eliminate a substitution set from the stream if no match is found. The stream of substitution sets may grow and shrink as it passes through a series of conjuncts.

The basis of stream processing is a set of functions to create, augment, and access the elements of a stream. We can define a simple set of stream functions using lists and the standard list manipulators. The functions that constitute a list-based implementation of the stream data type are:

```
                    ;cons-stream adds a new first element to a stream.
(defun cons-stream (element stream)
   (cons element stream))

                 ;head-stream returns the first element of the stream.
(defun head-stream (stream) (car stream))

           ;tail-stream returns the stream with first element deleted.
(defun tail-stream (stream) (cdr stream))

                   ;empty-stream-p is true if the stream is empty.
(defun empty-stream-p (stream) (null stream))

                  ;make-empty-stream creates an empty stream.
(defun make-empty-stream ( ) nil)

                        ;combine-stream appends two streams.
(defun combine-streams (stream1 stream2)
     (cond ((empty-stream-p stream1) stream2)
         (t (cons-stream (head-stream stream1)
            (combine-streams
               (tail-stream stream 1)
               stream2)))))
```

Although the implementation of streams as lists does not allow the full power of stream-based abstraction, the definition of a stream data type helps us to view the program from a data flow point of view. For many problems, such as the logic programming interpreter of Section 16.3, this provides the programmer with a powerful tool for organizing and simplifying the code. In Section 17.1 we discuss some limitations of this list-based implementation of streams and present an alternative approach using streams with delayed evaluation.

16.3 A Stream-Based Logic Programming Interpreter

We invoke the interpreter through a function called `logic-shell`, a straightforward variation of the `read-eval-print` loop discussed in Section 15.3. After printing a prompt, "`?`", it reads the next s-expression entered by the user and binds it to the symbol `goal`. If `goal` is equal to `quit`, the function halts; otherwise, it calls `solve` to generate a stream of

substitution sets that satisfy the `goal`. This stream is passed to `print-solutions`, which prints the `goal` with each of these different substitutions. The function then recurs. `logic-shell` is defined:

```
(defun logic-shell ( )
     (print '? )
     (let ((goal (read)))
          (cond ((equal goal 'quit) 'bye)
                (t (print-solutions goal
                     (solve goal nil))
                     (terpri)
                     (logic-shell)))))
```

`solve` is the heart of the interpreter. `solve` takes a goal and a set of substitutions and finds all solutions that are consistent with the knowledge base. These solutions are returned as a stream of substitution sets; if there are no matches, solve returns the empty stream. From the stream processing point of view, solve is a source, or generator, for a stream of solutions. `solve` is defined by:

```
(defun solve (goal substitutions)
     (declare (special *assertions*))
     (if (conjunctive-goal-p goal)
         (filter-through-conj-goals (body goal)
              (cons-stream substitutions
                 (make-empty-stream)))
         (infer goal substitutions *assertions*)))
```

The declaration `special` tells the Lisp compiler that `*assertions*` is a special, or global, variable and should be bound dynamically in the environment in which `solve` is called. (This special declaration is not required in many modern versions of Lisp.)

`solve` first tests whether the goal is a conjunction; if it is, `solve` calls `filter-through-conj-goals` to perform the filtering described in Section 16.2. If `goal` is not a conjunction, `solve` assumes it is a simple goal and calls `infer`, defined below, to `solve` it against the knowledge base. `solve` calls `filter-through-conj-goals` with the body of the conjunction (i.e., the sequence of conjuncts with the and operator removed) and a stream that contains only the initial substitution set. The result is a stream of substitutions representing all of the solutions for this goal. We define `filter-through-conj-goals` by:

```
(defun filter-through-conj-goals (goals
        substitution-stream)
     (if (null goals) substitution-stream
     (filter-through-conj-goals (cdr goals)
               (filter-through-goal (car goals)
                 substitution-stream))))
```

If the list of goals is empty, the function halts, returning `substitution-stream` unchanged. Otherwise, it calls `filter-`

`through-goal` to filter `substitution-stream` through the first goal on the list. It passes this result on to a recursive call to `filter-through-conj-goals` with the remainder of the goal list. Thus, the stream is passed through the goals in left-to-right order, growing or shrinking as it passes through each goal.

`filter-through-goal` takes a single goal and uses it as a filter to the stream of substitutions. This filtering is done by calling `solve` with the goal and the first set of substitutions in the `substitution-stream`. The result of this call to `solve` is a stream of substitutions resulting from matches of the goal against the knowledge base. This stream will be empty if the goal does not succeed under any of the substitutions contained in the stream, or it may contain multiple substitution sets representing alternative bindings. This stream is combined with the result of filtering the tail of the input stream through the same goal:

```
(defun filter-through-goal
          (goal substitution-stream)
      (if (empty-stream-p substitution-stream)
      (make-empty-stream)
      (combine-streams
          (solve goal
          (head-stream substitution-stream))
            (filter-through-goal goal
                (tail-stream substitution-stream)))))
```

To summarize, `filter-through-conj-goals` passes a stream of substitution sets through a sequence of goals, and `filter-through-goal` filters `substitution-stream` through a single goal. A recursive call to `solve` solves the goal under each substitution set.

Whereas `solve` handles conjunctive goals by calling `filter-through-conj-goals`, simple goals are handled by `infer`, defined next, which takes a `goal` and a set of substitutions and finds all solutions in the knowledge base, `kb`, `infer`'s third parameter, a database of logic expressions. When `solve` first calls `infer`, it passes the knowledge base contained in the global variable `*assertions*`. `infer` searches `kb` sequentially, trying the goal against each fact or rule conclusion.

The recursive implementation of `infer` builds the backward-chaining search typical of Prolog and many expert system shells. It first checks whether `kb` is empty, returning an empty stream if it is. Otherwise, it binds the first item in `kb` to the symbol assertion using a `let*` block. `let*` is like `let` except it is guaranteed to evaluate the initializations of its local variables in sequentially nested scopes, i.e., it provides an order to the binding and visibility of preceding variables. It also defines the variable match: if assertion is a rule, `let*` initializes `match` to the substitutions required to unify the goal with the conclusion of the rule; if `assertion` is a fact, `let*` binds `match` to those substitutions required to unify `assertion` with `goal`. After attempting to unify the goal with the first element of the knowledge base, `infer` tests whether the unification succeeded. If it failed to `match`, `infer` recurs, attempting to solve the

goal using the remainder of the knowledge base. If the unification succeeded and assertion is a rule, `infer` calls `solve` on the premise of the rule using the augmented set of substitutions bound to `match`. `combine-stream` joins the resulting stream of solutions to that constructed by calling `infer` on the rest of the knowledge base. If `assertion` is not a rule, it is a fact; `infer` adds the solution bound to `match` to those provided by the rest of the knowledge base. Note that once the goal unifies with a fact, it is solved; this terminates the search. We define `infer`:

```
(defun infer (goal substitutions kb)
  (if (null kb)
     (make-empty-stream)
     (let* ((assertion
             (rename-variables (car kb)))
         (match (if (rulep assertion)
            (unify goal (conclusion assertion)
                substitutions)
            (unify goal assertion substitutions))))
            (if (equal match 'failed)
               (infer goal substitutions (cdr kb))
            (if (rulep assertion)
               (combine-streams
                 (solve (premise assertion) match)
                 (infer goal substitutions
                        (cdr kb)))
               (cons-stream match
                  (infer goal substitutions
                                 (cdr kb))))))))
```

Before the first element of `kb` is bound to `assertion`, it is passed to `rename-variables` to give each variable a unique name. This prevents name conflicts between the variables in the goal and those in the knowledge base entry; e.g., if `(var x)` appears in a goal, it must be treated as a different variable than a `(var x)` that appears in the rule or fact. (This notion of standardizing variables apart is an important component of automated reasoning in general. Luger (2009, Section 14.2) demonstrates this in the context of resolution refutation systems). The simplest way to handle this is by renaming all variables in `assertion` with unique names. We define `rename-variables` at the end of this section.

This completes the implementation of the core of the logic programming interpreter. To summarize, `solve` is the top-level function and generates a stream of substitution sets (`substitution-stream`) that represent solutions to the goal using the knowledge base. `filter-through-conj-goals` solves conjunctive goals in a left-to-right order, using each goal as a filter on a stream of candidate solutions: if a goal cannot be proven true against the knowledge base using a substitution set in the

stream, `filter-through-conj-goals` eliminates those substitutions from the stream. If the goal is a simple literal, `solve` calls `infer` to generate a stream of all substitutions that make the goal succeed against the knowledge base. Like Prolog, our logic programming interpreter takes a goal and finds all variable bindings that make it true against a given knowledge base.

All that remain are functions for accessing components of knowledge base entries, managing variable substitutions, and printing solutions. `print-solutions` takes as arguments a `goal` and a `substitution-stream`. For each set of substitutions in the stream, it prints the goal with variables replaced by their bindings in the substitution set.

```
(defun print-solutions (goal substitution-stream)
    (cond ((empty-stream-p substitution-stream)
                        nil)
          (t (print (apply-substitutions goal
                    (head-stream
                   substitution-stream)))
             (terpri)
             (print-solutions goal
                (tail-stream substitution-stream)))))
```

The replacement of variables with their values under a substitution set is done by `apply-substitutions`, which does a car-cdr recursive tree walk on a pattern. If the pattern is a constant (`is-constant-p`), it is returned unchanged. If it is a variable (`varp`), `apply-substitutions` tests if it is bound. If it is unbound, the variable is returned; if it is bound, `apply-substitutions` calls itself recursively on the value of this binding. Note that the binding value may be either a constant, another variable, or a pattern of arbitrary complexity:

```
(defun apply-substitutions
          (pattern substitution-list)
   (cond ((is-constant-p pattern) pattern)
      ((varp pattern)
         (let ((binding
                     (get-binding pattern
                   substitution-list)))
           (cond (binding (apply-substitutions
                     (get-binding-value binding)
                      substitution-list))
                 (t pattern))))
        (t (cons (apply-substitutions
                 (car pattern)
                 substitution-list)
            (apply-substitutions (cdr pattern)
                 substitution-list)))))
```

`infer` renamed the variables in each knowledge base entry before matching it with a goal. This is necessary, as noted above, to prevent undesired name collisions in matches. For example, the goal `(p a (var x))` should match with the knowledge base entry `(p (var x) b)`, because the scope of each `(var x)` is restricted to a single expression. As unification is defined, however, this match will not occur. Name collisions are prevented by giving each variable in an expression a unique name. The basis of our renaming scheme is a Common Lisp built-in function called `gensym` that takes no arguments; each time it is called, it returns a unique symbol consisting of a number preceded by `#:G`. For example:

```
> (gensym)
#:G4
> (gensym)
#:G5
> (gensym)
#:G6
>
```

Our renaming scheme replaces each variable name in an expression with the result of a call to `gensym`. `rename-variables` performs certain initializations (described below) and calls `rename-rec` to make substitutions recursively in the pattern. When a variable (`varp`) is encountered, the function rename is called to return a new name. To allow multiple occurrences of a variable in a pattern to be given consistent names, each time a variable is renamed, the new name is placed in an association list bound to the special variable `*name-list*`. The special declaration makes all references to the variable dynamic and shared among these functions. Thus, each access of `*name-list*` in rename will access the instance of `*name-list*` declared in `rename-variables`. `rename-variables` initializes `*name-list*` to `nil` when it is first called on an expression. These functions are defined:

```
(defun rename-variables (assertion)
     (declare (special *name-list*))
     (setq *name-list* nil)
     (rename-rec assertion))

(defun rename-rec (exp)
     (declare (special *name-list*))
     (cond ((is-constant-p exp) exp)
           ((varp exp) (rename exp))
           (t (cons (rename-rec (car exp))
                    (rename-rec (cdr exp))))))

(defun rename (var)
     (declare (special *name-list*))
```

```
        (list 'var (or (cdr (assoc var *name-list*
                                   :test #'equal))
                       (let ((name (gensym)))
                          (setq *name-list*
                             (acons var name *name-list*))
                          name))))
```

The final functions access components of rules and goals and are self-explanatory:

```
(defun premise (rule) (nth 2 rule))

(defun conclusion (rule) (nth 4 rule))

(defun rulep (pattern)
     (and (listp pattern) (equal (nth 0 pattern)
          'rule)))

(defun conjunctive-goal-p (goal)
     (and (listp goal) (equal (car goal) 'and)))

(defun body (goal) (cdr goal))
```

In Chapter 17 we extend the ideas of Chapter 16 to delayed evaluation using lexical closures. Finally we build a goal-driven expert system shell in Lisp.

Exercises

1. Expand the logic programming interpreter to include Lisp `write` statements. This will allow rules to print messages directly to the user. Hint: modify `solve` first to examine if a goal is a `write` statement. If it is, evaluate the `write` and return a stream containing the initial substitution set.

2. Rewrite print-solutions in the logic programming interpreter so that it prints the first solution and waits for a user response (such as a carriage return) before printing the second solution.

3. Implement the general map and filter functions, `map-stream` and `filter-stream`, described in Section 16.3.

4. Expand the logic programming interpreter to include `or` and `not` relations. This will allow rules to contain more complex relationships between its premises.

5. Expand the logic programming language to include arithmetic comparisons, `=, <,` and `>`. Hint: as

in Exercise 1, modify `solve` to detect these comparisons before calling `infer`. If an expression is a comparison, replace any variables with their values and evaluate it. If it returns `nil`, `solve` should return the empty stream; if it returns non-`nil`, `solve` should return a stream containing

the initial substitution set. Assume that the expressions do not contain unbound variables.

6. For a more challenging exercise, expand the logic programming interpreter to define = so that it will function like the Prolog is operator and assign a value to an unbound variable and simply do an equality test if all elements are bound.

17 Lisp-shell: An Expert System Shell in Lisp

Chapter Objectives

This chapter defines streams (lists) and delayed evaluation with functions
- `delay`
- `force`

Stream processing based on lexical closures
- Freezes evaluation of stream
- Closures preserve variable bindings and scope

`lisp-shell` created as full expert system shell in Lisp
Unification based on stream processing
Askable list organizes user queries
- Full certainty factor system based on Stanford Certainty Factor Algebra

Demonstration of `lisp-shell` with a "plant identification" data base
Exercises on extending function of `lisp-shell`

Chapter Contents

17.1 Streams and Delayed Evaluation

Why delayed evaluation?

As we demonstrated in the implementation of `logic-shell` in Chapter 16, a stream-oriented view can help with the organization of a complex program. However, our implementation of streams as lists did not provide the full benefit of stream processing. In particular, this implementation suffers from inefficiency and an inability to handle potentially infinite data streams.

In the list implementation of streams, all of the elements must be computed before that stream (list) can be passed on to the next function. In `logic-shell` this leads to an exhaustive search of the knowledge base for each intermediate goal in the solution process. In order to produce the first solution to the top-level goal, the program must produce a list of all solutions. Even if we want only the first solution on this list, the program must still search the entire solution space. What we would really prefer is for the program to produce just the first solution by searching only that portion of the space needed to produce that solution and then to delay finding the rest of the goals until they are needed.

A second problem is the inability to process potentially infinite streams of information. Although this problem does not arise in `logic-shell`, it occurs naturally in the stream-based solution to many problems. Assume, for example, that we would like to write a function that returns a stream of the first n odd Fibonacci numbers. A straightforward implementation would use a generator to produce a stream of Fibonacci numbers, a filter to eliminate the

even-valued numbers from the stream, and an accumulator to gather these into a solution list of n elements, as in Figure 17.1. Unfortunately, the stream of Fibonacci numbers is infinite in length and we cannot decide in advance how long a list will be needed to produce the first n odd numbers.

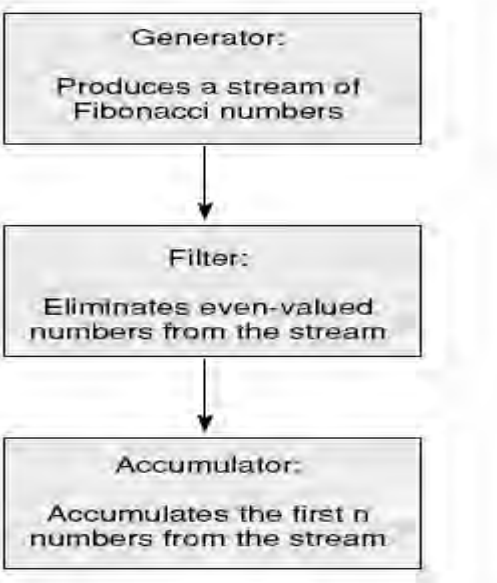

Figure 17.1. A stream implementation that finds the first n odd Fibonacci numbers.

Instead, we would like the generator to produce the stream of Fibonacci numbers one at a time and pass each number through the filter until the accumulator has gathered the n values required. This behavior more closely fits our intuitive notion of evaluating a stream than does the list-based implementation of Chapter 16. We accomplish this by use of *delayed evaluation*.

Delayed Evaluation and Function Closures

Instead of letting the generator run to completion to produce the entire stream of results, we let the function produce the first element of the stream and then freeze or delay its execution until the next element is needed. When the program needs the next element of the stream, it causes the function to resume execution and produce only that element and again delay evaluation of the rest of the stream. Thus, instead of containing the entire list of numbers, the stream consists of just two components, its first element and the frozen computation of the rest of the stream, as shown in Figure 17.2.

A list-based stream containing an indeterminate number of elements:

$(e_1\ e_2\ e_3\ e_4\ .\ .\ .)$

A stream with delayed evaluation of its tail containing two elements but capable of producing any number of elements:

$(e_1$. <delayed evaluation of rest of stream>)

Figure 17.2 A list-based and delayed evaluation of streams.

We use *function closures* to create the delayed portion of the stream that was illustrated by Figure 16.1. A closure consists of a function, along with all its variable bindings in the current environment; we may bind a closure to a variable, or pass it as a parameter, and evaluate it using `funcall`. Essentially, a closure "freezes" a function application until a later time. We can create closures using the Lisp form `function`. For example, consider the following Lisp transcript:

```
> (setq v 10)
10
> (let ((v 20)) (setq f_closure (function (lambda (
) v))))
#<COMPILED-LEXICAL-CLOSURE #x28641E>
> (funcall f_closure)
20
>
10
```

The initial `setq` binds `v` to `10` in the global environment. In the `let` block, we create a local binding of `v` to `20` and create a closure of a function that returns this value of `v`. It is interesting to note that this binding of `v` does not disappear when we exit the `let` block, because it is retained in the function closure that is bound to `f_closure`. It is a lexical binding, however, so it doesn't shadow the global binding of `v`. If we subsequently evaluate this closure, it returns `20`, the value of the local binding of `v`, even though the global `v` is still bound to `10`.

The heart of this implementation of streams is a pair of functions, `delay` and `force`. `delay` takes an expression as argument and does not evaluate it; instead it takes the unevaluated argument and returns a closure. `force` takes a function closure as argument and uses `funcall` to force its application. These functions are defined:

```
(defmacro delay (exp) '(function (lambda ( ) ,exp)))
(defun force (function-closure)
     (funcall function-closure))
```

`delay` is an example of a Lisp form called a *macro*. We cannot define `delay` using `defun` because all functions so defined evaluate their arguments before executing the body. Macros give us complete control over the evaluation of their arguments. We define macros using the `defmacro` form. When a macro is executed, it does not evaluate its arguments. Instead, it binds the unevaluated s-expressions in the call to the formal parameters and evaluates its body *twice*. The first evaluation is called a *macro-expansion*; the second evaluates the resulting form.

To define the `delay` macro, we introducethe *backquote* or `'`. Backquote prevents evaluation just like a `quote`, except that it allows us to evaluate selectively elements of the backquoted expression. Any element of a backquoted s-expression preceded by a comma is evaluated and its value inserted into the resulting expression. For example, assume we have the call `(delay (+ 2 3))`. The expression `(+ 2 3)` is not evaluated; instead it is bound to the formal parameter, `exp`. When the body of the macro is

evaluated the first time, it returns the backquoted expression with the formal parameter, `exp`, replaced by its value, the unevaluated s-expression `(+ 2 3)`. This produces the expression `(function (lambda () (+ 2 3)))`. This is evaluated again, returning a function closure.

If we later pass this closure to `force`, it will evaluate the expression `(lambda () (+ 2 3))`. This is a function that takes no arguments and whose body evaluates to `5`. Using `force` and `delay`, we can implement streams with delayed evaluation. We rewrite `cons-stream` as a macro that takes two arguments and `cons`es the value of the first onto the delayed evaluation of the second. Thus, the second argument may be a function that will return a stream of any length; it is not evaluated. We define `tail-stream` so that it forces the evaluation of the tail of a stream. These are defined:

```
(defmacro cons-stream (exp stream) '(cons ,exp
   (delay ,stream)))
(defun tail-stream (stream) (force (cdr stream)))
```

We also redefine `combine-streams` as a macro that takes two streams but does not evaluate them. Instead, it uses `delay` to create a closure for the second stream and passes this and the first stream to the function `comb-f`. `comb-f` is similar to our earlier definition of `combine-streams`, except that in the event that the first stream is empty, it forces evaluation of the second stream. If the first stream is not empty, the recursive call to `comb-f` is done using our delayed version of `cons-stream`. This freezes the recursive call in a closure for later evaluation.

```
(defmacro combine-streams (stream1 stream2)
     '(comb-f ,stream1 (delay ,stream2)))
(defun comb-f (stream1 stream2)
     (if (empty-stream-p stream1)
         (force stream2)
     (cons-stream (head-stream stream1)
        (comb-f (tail-stream stream1) stream2))))
```

If we add these definitions to the versions of `head-stream`, `make-empty-stream`, and `empty-stream-p` from Section 16.2, we have a complete stream implementation with delayed evaluation.

We can use these functions to solve our problem of producing the first `n` odd Fibonacci numbers. `fibonacci-stream` returns a stream of all the Fibonacci numbers; note that `fibonacci-stream` is a nonterminating recursive function. Delayed evaluation prevents it from looping forever; it produces the next element only when needed. `filter-odds` takes a stream of integers and eliminates the even elements of the stream. `accumulate` takes a stream and a number `n` and returns a *list* of the first `n` elements of the stream.

```
(defun fibonacci-stream (fibonacci-1 fibonacci-2)
     (cons-stream (+ fibonacci-1 fibonacci-2)
        (fibonacci-stream fibonacci-2
            (+ fibonacci-1 fibonacci-2))))
```

```
(defun filter-odds (stream)
        (cond ((evenp (head-stream stream))
               (filter-odds (tail-stream stream)))
              (t (cons-stream (head-stream stream)
               (filter-odds (tail-stream stream))))))
(defun accumulate-into-list (n stream)
         (cond ((zerop n) nil)
               (t (cons (head-stream stream)
                        (accumulate-into-list (- n 1)
                             (tail-stream stream))))))
```

To obtain a list of the first 25 odd Fibonacci numbers, we call `accumulate-into-list`:

```
(accumulate-into-list 25
          (filter-odds (fibonacci-stream 0 1)))
```

We may use these stream functions in the definition of the logic programming interpreter of Section 16.3 to improve its efficiency under certain circumstances. Assume that we would like to modify `print-solutions` so that instead of printing all solutions to a goal, it prints the first and waits for the user to ask for the additional solutions. Using our implementation of lists as streams, the algorithm would still search for all solutions before it could print out the first. Using delayed evaluation, the first solution will be the head of a stream, and the function evaluations necessary to find the additional solutions will be frozen in the tail of the stream.

In the next section we modify this logic programming interpreter to implement a Lisp-based expert system shell called `lisp-shell`. Before presenting the expert system shell, however, we mention two additional stream functions that are used in its implementation. In Section 16.3, we presented a general mapping function and a general filter for lists. These functions, `map-simple` and `filter`, can be modified to function on streams. We use `filter-stream` and `map-stream` in the next section; their implementation is an exercise.

17.2 An Expert System Shell in Lisp

The expert system shell developed in this section is an extension of the backward-chaining engine of Section 16.3. The major modifications include the use of certainty factors to manage uncertain reasoning, the ability to ask the user for unknown facts, and the use of a working memory to save user responses. This expert system shell is called `lisp-shell`.

Implementing Certainty Factors

The logic programming interpreter returned a stream of the substitution sets under which a goal logically followed from a database of logical assertions. Bindings that did not allow the goal to be satisfied using the knowledge base were either filtered from the stream or not generated in the first place. In implementing reasoning with certainty factors, however, logic truth values (`t`, `f`) are replaced by a numeric value between `-1` and `1`.

This replacement requires that the stream of solutions to a goal not only contain the variable bindings that allow the goal to be satisfied; they must also include measures of the confidence under which each solution follows from the knowledge base. Consequently, instead of processing streams of substitution sets, `lisp-shell` processes streams of pairs: a set of substitutions and a number representing the confidence in the truth of the goal under those variable substitutions.

We implement stream elements as an abstract data type: the functions for manipulating the substitution and certainty factor pairs are `subst-record`, which constructs a pair from a set of substitutions and a certainty factor; `subst-list`, which returns the set of bindings from a pair; and `subst-cf`, which returns the certainty factor.

Internally, records are represented as dotted pairs, of the form `(<substitution list>. <cf>)`. We next create functions that handle these pairs, the first returning a list of bindings, the second returning a certainty factor, and the third creating substitution-set certainty-factor pairs:

```
(defun subst-list (substitutions)
    (car substitutions))
(defun subst-cf (substitutions)
    (cdr substitutions))
(defun subst-record (substitutions cf)
    (cons substitutions cf))
```

Similarly, rules and facts are stored in the knowledge base with an attached certainty factor. Facts are represented as dotted pairs, `(<assertion>. <cf>)`, where `<assertion>` is a positive literal and `<cf>` is its certainty measure. Rules are in the format `(rule if <premise> then <conclusion> <cf>)`, where `<cf>` is the certainty factor. We next create a sample rule for the domain of recognizing different types of flowers:

```
(rule if (and (rose (var x)) (color (var x) red))
    then (kind (var x) american-beauty) 1)
```

The functions for handling rules and facts are:

```
(defun premise (rule)
    (nth 2 rule))
(defun conclusion (rule)
    (nth 4 rule))
(defun rule-cf (rule)
    (nth 5 rule))
(defun rulep (pattern)
    (and (listp pattern)
         (equal (nth 0 pattern) 'rule)))
```

```
(defun fact-pattern (fact)
    (car fact))
(defun fact-cf (fact)
    (cdr fact))
```

Using these functions, we implement the balance of the rule interpreter through a series of modifications to the logic programming interpreter first presented in Section 16.3.

Architecture of lisp-shell

`solve` is the heart of lisp-shell. `solve` does not return a solution stream directly but first passes it through a filter that eliminates any substitutions whose certainty factor is less than 0.2. This prunes results that lack sufficient confidence.

```
(defun solve (goal substitutions)
   (filter-stream
      (if (conjunctive-goal-p goal)
         (filter-through-conj-goals
            (cdr (body goal))
            (solve (car (body goal))
                 substitutions))
           (solve-simple-goal goal
              substitutions))
            # '(lambda (x)
            (< 0.2 (subst-cf x)))))
```

This definition of `solve` has changed only slightly from the definition of `solve` in `logic-shell`. It is still a conditional statement that distinguishes between conjunctive goals and simple goals. One difference is the use of the general filter `filter-stream` to prune any solution whose certainty factor falls below a certain value. This test is passed as a lambda expression that checks whether or not the certainty factor of a substitution set/cf pair is less than 0.2. The other difference is to use `solve-simple-goal` in place of infer. Handling simple goals is complicated by the ability to ask for user information. We define `solve-simple-goal` as:

```
(defun solve-simple-goal (goal substitutions)
     (declare (special *assertions*))
     (declare (special *case-specific-data*))
     (or (told goal substitutions
              *case-specific-data*)
      (infer goal substitutions *assertions*)
      (ask-for goal substitutions)))
```

`solve-simple-goal` uses an or form to try three different solution strategies in order. First it calls told to check whether the goal has already been solved by the user in response to a previous query.

User responses are bound to the global variable `*case-specific-data*`; `told` searches this list to try to find a match for the goal. This keeps `lisp-shell` from asking for the same piece of information twice. If this fails, the information was not asked for earlier, and `solve-simple-goal` attempts to infer the goal using the rules in `*assertions*`. Finally, if these fail, it calls `ask-for` to query the user for the information. These functions are defined below.

The top-level read-solve-print loop has changed little, except for the inclusion of a statement initializing `*case-specific-data*` to `nil` before solving a new goal. Note that when `solve` is called initially, it is not just passed the empty substitution set, but a pair consisting of the empty substitution set and a certainty factor of 0. This certainty value has no real meaning: it is included for syntactic reasons until a meaningful substitution set and certainty factor pair is generated by user input or by a fact in the knowledge base.

```
(defun lisp-shell ()
      (declare (special *case-specific-data*))
      (setq *case-specific-data* ( ))
      (prin1 'lisp-shell> )
      (let ((goal (read)))
      (terpri)
      (cond ((equal goal 'quit) 'bye)
            (t (print-solutions goal
                (solve goal
                   (subst-record nil 0)))
              (terpri)
              (lisp-shell)))))
```

`filter-through-conj-goals` is not changed, but `filter-through-goal` must compute the certainty factor for a conjunctive expression as the minimum of the certainties of the conjuncts. To do so, it binds the first element of `substitution-stream` to the symbol `subs` in a `let` block. It then calls `solve` on the goal and this substitution set; passing the result through the general mapping function, `map-stream`, which takes the stream of substitution pairs returned by `solve` and recomputes their certainty factors as the minimum of the certainty factor of the result and the certainty factor of the initial substitution set. These functions are defined:

```
(defun filter-through-conj-goals (goals
          substitution-stream)
      (if (null goals)
      substitution-stream
      (filter-through-conj-goals (cdr goals)
          (filter-through-goal (car goals)
                  substitution-stream))))
```

```
(defun filter-through-goal (goal
                            substitution-stream)
   (if (empty-stream-p substitution-stream)
      (make-empty-stream)
      (let ((subs (head-stream
                   substitution-stream)))
         (combine-streams
         (map-stream (solve goal subs)
              # '(lambda (x)
                (subst-record (subst-list x)
                         (min (subst-cf x)
                        (subst-cf subs)))))
         (filter-through-goal goal
                  (tail-stream
                  substitution-stream))))))
```

The definition of `infer` has been changed to take certainty factors into account. Although its overall structure reflects the version of `infer` written for the logic programming interpreter in Section 16.2, we must now compute the certainty factor for solutions to the goal from the certainty factors of the rule and the certainties of solutions to the rule premise. `solve-rule` calls `solve` to find all solutions to the premise and uses `map-stream` to compute the resulting certainties for the rule conclusion.

```
(defun infer (goal substitutions kb)
   (if (null kb)
      (make-empty-stream)
      (let* ((assertion
         (rename-variables (car kb)))
           (match (if (rulep assertion)
         (unify goal (conclusion assertion)
               subst-list substitutions))
         (unify goal assertion
              (subst-list substitutions)))))
            (if (equal match 'failed)
               (infer goal substitutions
                        (cdr kb))
               (if (rulep assertion)
                  (combine-streams
                     (solve-rule assertion
                       (subst-record match
                       (subst-cf  substitutions)))
                     (infer goal substitutions
                        (cdr kb)))
```

```
                    (cons-stream
                    (subst-record match
                    (fact-cf assertion))
                          (infer goal substitutions
                       (cdr kb))))))))
((defun solve-rule (rule substitutions)
     (map-stream
      (solve (premise rule) substitutions)
        # '(lambda (x) (subst-record
              (subst-list x)
              (* (subst-cf x)
               (rule-cf rule))))))
```

Finally, we modify **print-solutions** to use certainty factors:

```
(defun print-solutions (goal substitution-stream)
   (cond ((empty-stream-p substitution-stream)nil)
         (t (print (apply-substitutions goal
            (subst-list (head-stream
                 substitution-stream))))
           (write-string "cf =")
           (prin1 (subst-cf (head-stream
                    substitution-stream)))
           (terpri)
           (print-solutions goal
                  (tail-stream
              substitution-stream)))))
```

The remaining functions, such as **apply-substitutions** and functions for accessing rules and goals, are unchanged from Section 16.2.

The remainder of **lisp-shell** consists of the functions **ask-for** and **told**, which handle user interactions. These are straightforward, although the reader should note that we have made some simplifying assumptions. In particular, the only response allowed to queries is either "y" or "n". This causes the binding set passed to **ask-for** to be returned with a cf of either 1 or –1, respectively; the user may not give an uncertain response directly to a query. **ask-rec** prints a query and reads the answer, repeating until the answer is either y or n. The reader may expand **ask-rec** to take on any value within the –1 to 1 range. (–1 and 1, of course, offers an arbitrary range; particular applications may use other ranges.)

askable verifies whether the user may be asked for a particular goal. Any asked goal must exist as a pattern in the global list ***askables***; the architect of an expert system may in this way determine which goals may be asked for and which may only be inferred from the knowledge base. **told** searches through the entries in the global ***case-specific-data*** to find whether the user has already answered a query. It is similar to **infer** except it assumes that everything in ***case-specific-data*** is stored as a fact. We define these functions:

```
(defun ask-for (goal substitutions)
  (declare (special *askables*))
  (declare (special *case-specific-data*))
  (if (askable goal *askables*)
      (let* ((query (apply-substitutions goal
                (subst-list substitutions)))
             (result (ask-rec query)))
  ((setq *case-specific-data*
           (cons (subst-record query result)
                 *case-specific-data*))
        (cons-stream
           (subst-record (subst-list substitutions)
                result)
           (make-empty-stream)))))
(defun ask-rec (query)
     (prin1 query)
     (write-string ">")
          (let ((answer (read)))
          (cond ((equal answer 'y) 1)
                ((equal answer 'n) – 1)
                (t (print
                  "answer must be y or n")
                   (terpri)
                    (ask-rec query)))))
(defun askable (goal askables)
     (cond ((null askables) nil)
        ((not (equal (unify goal car askables) ( ))
                 'failed)) t)
        (t (askable goal (cdr askables)))))
(defun told (goal substitutions case-specific-data)
     (cond ((null case-specific-data)
                 (make-empty-stream))
        (t (combine-streams
           (use-fact goal (car case-specific-data)
                 substitutions)
           (told goal substitutions
              (cdr case-specific-data))))))
```

This completes the implementation of our Lisp-based expert system shell. In the next section we use `lisp-shell` to build a simple classification expert system.

Classification Using lisp-shell

We now present a small expert system for classifying trees and bushes. Although it is far from botanically complete, it illustrates the use and behavior of the `lisp-shell` software. The knowledge base resides in

two global variables: ***assertions***, which contains the rules and facts of the knowledge base, and ***askables***, which lists the goals that may be asked of the user. The knowledge base used in this example is constructed by two calls to **setq**:

```
(setq *assertions* '(
   (rule
         if (and (size (var x) tall)
                 (woody (var x)))
         then (tree (var x)) .9)
   (rule
         if (and (size (var x) small)
                 (woody (var x)))
         then (bush (var x)) .9)
   (rule
         if (and (tree (var x)) (evergreen (var x))
             (color (var x) blue))
         then (kind (var x) spruce) .8)
   (rule
         if (and (tree (var x)) (evergreen (var x))
             (color (var x) green))
         then (kind (var x) pine) .9)
   (rule
         if (and (tree (var x)) (deciduous (var x))
             (bears (var x) fruit))
         then (fruit-tree (var x)) 1)
   (rule
         if (and (fruit-tree (var x))
                 (color fruit red)
                 (taste fruit sweet))
         then (kind (var x) apple-tree) .9)
   (rule
         if (and (fruit-tree (var x))
                 (color fruit yellow)
                 (taste fruit sour))
         then (kind (var x) lemon-tree) .8)
   (rule
         if (and (bush (var x))
                 (flowering (var x))
                 (thorny (var x)))
         then (rose (var x)) 1)
   (rule
         if (and (rose (var x)) (color (var x) red))
         then (kind (var x) american-beauty) 1)))
```

```
(setq *askables* '(
    (size (var x) (var y))
    (woody (var x))
    (soft (var x))
    (color (var x) (var y))
    (evergreen (var x))
    (thorny (var x))
    (deciduous (var x))
    (bears (var x) (var y))
    (taste (var x) (var y))
    (flowering (var x))))
```

A sample run of the trees knowledge base appears below. The reader is encouraged to trace through the rule base to observe the order in which rules are tried, the propagation of certainty factors, and the way in which possibilities are pruned when found to be false:

```
> (lisp-shell)
lisp-shell>(kind tree-1 (var x))
(size tree-1 tall) >y
(woody tree-1) >y
(evergreen tree-1) >y
(color tree-1 blue) >n
(color tree-1 green) >y
(kind tree-1 pine) cf 0.81
(deciduous tree-1) >n
(size tree-1 small) >n
lisp-shell>(kind bush-2 (var x))
(size bush-2 tall) >n
(size bush-2 small) >y
(woody bush-2) >y
(flowering bush-2) >y
(thorny bush-2) >y
(color bush-2 red) >y
(kind bush-2 american-beauty) cf 0.9
lisp-shell>(kind tree-3 (var x))
(size tree-3 tall) >y
(woody tree-3) >y
(evergreen tree-3) >n
(deciduous tree-3) >y
(bears tree-3 fruit) >y
(color fruit red) >n
(color fruit yellow) >y
(taste fruit sour) >y
```

```
(kind tree-3 lemon-tree) cf 0.72
(size tree-3 small) >n
lisp-shell>quit
bye
?
```

In this example, several anomalies may be noted. For example, `lisp-shell` occasionally asks whether a tree is small even though it was told the tree is tall, or it asks whether the tree is deciduous even though the tree is an evergreen. This is typical of the behavior of expert systems. The knowledge base does not know anything about the relationship between tall and small or evergreen and deciduous: they are just patterns to be matched. Because the search is exhaustive, all rules are tried. If a system is to exhibit deeper knowledge than this, these relationships must be coded in the knowledge base. For example, a rule may be written that states that small implies not tall. In this example, `lisp-shell` is not capable of representing these relationships because we have yet to implement the `not` operator. This extension is left as an exercise.

Exercises

1. Rewrite the solution to finding the first `n` odd Fibonacci numbers problem of Section 17.1 so that it uses the general stream filter, `filter-stream`, instead of `filter-odds`. Modify this to return the first `n` even Fibonacci numbers and then modify it again to return the squares of the first `n` Fibonacci numbers.

2. Select a problem such as automotive diagnosis or classifying different species of animals and solve it using `lisp-shell`.

3. Expand the expert system shell of Section 17.2 to allow the user responses other than `y` or `n`. For example, we may want the user to be able to provide bindings for a goal. Hint: This may be done by changing the `ask-for` and related functions to let the user also enter a pattern, which is matched against the goal. If the match succeeds, ask for a certainty factor.

4. Extend `lisp-shell` to include `not`. For an example of how to treat negation using uncertain reasoning, refer to the Prolog-based expert system shell in Chapter 6.

5. In Section 16.3, we presented a general mapping function and a general filter for lists. These functions, `map-simple` and `filter`, can be modified to function on streams. Create the `filter-stream` and `map-stream` functions used in 17.2.

6. Extend `lisp-shell` to produce an answer even when all rules fail to match. In other words, remove the `nil` as a possible result for `lisp-shell`.

18 Semantic Networks, Inheritance, and CLOS

Chapter Objectives

We build *Semantic Networks* in Lisp:
    Supported by *property lists*
First implementation (early 1970s) of *object systems*
Object systems in Lisp include:
    Encapsulation
    Inheritance
        Hierarchical
    Polymorphism
The Common Lisp Object System (CLOS)
Encapsulation
    Inheritance
        Inheritance search programmer designed
Example CLOS implementation
    Further implementations in exercises

Chapter Contents

18.1 Semantic Networks and Inheritance in Lisp

This chapter introduces the implementation of semantic networks and inheritance, and a full object-oriented programming system in Lisp. As a family of representations, semantic networks provide a basis for a large variety of inferences, and are widely used in natural language processing and cognitive modeling. We do not discuss all of these, but focus on a basic approach to constructing network representations using *property lists*. After these are discussed and used to define a simple semantic network, we define a function for class inheritance. Finally, since semantic networks and inheritance are important precursors of object-oriented design, we present CLOS, the Common Lisp Object System, Section 18.2, and an example implementation in 18.3.

A Simple Semantic Network

Lisp is a convenient language for representing any graph structure, including semantic nets. Lists provide the ability to create objects of arbitrary complexity and these objects may be bound to names, allowing for easy reference and the definition of relationships between them. Indeed, all Lisp data structures are based on an internal implementation as chains of pointers, a natural isomorph to graph structures.

For example, labeled graphs may be represented using association lists: each node is an entry in an association list with all the arcs out of that node stored in the datum of the node as a second association list. Arcs are described by an

association list entry that has the arc name as its key and that has the arc destination as its datum. Using this representation, the built-in association list functions are used to find the destination of a particular arc from a given node. For example, the labeled, directed graph of Figure 18.1 is represented by the association list:

```
((a (1 . b))
    (b (2 . c))
    (c (2 . b) (3 . a)))
```

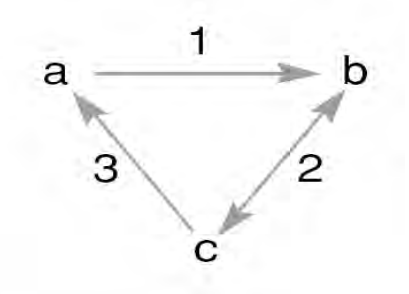

Figure 18.1 A simple labeled directed graph

This approach is the basis of many network implementations. Another way to implement semantic networks is through the use of *property lists*.

Essentially, property lists are a built-in feature of Lisp that allows named relationships to be attached to symbols. Rather than using `setq` to bind an association list to a symbol, with property lists we can program the direct attachment of named attributes to objects in the global environment. These are bound to the symbol not as a value but as an additional component called the property list.

Functions for managing property lists are `get`, `setf`, `remprop`, and `symbol-plist`. `get`, which has the syntax:

```
(get <symbol> <property-name>)
```

may be used to retrieve a property from <`symbol`> by its <`property-name`>. For example, if the symbol `rose` has a `color` property of `red` and a `smell` property of `sweet`, then `get` would have the behavior:

```
(get 'rose 'color)
red
(get 'rose 'smell)
sweet
(get 'rose 'party-affiliation)
nil
```

As the last of these calls to `get` illustrates, if an attempt is made to retrieve a nonexistent property, one that is not on the property list, `get` returns a value of `nil`.

Properties are attached to objects using the `setf` function, which has the syntax:

```
(setf <form> <value>)
```

`setf` is a generalization of `setq`. The first argument to `setf` is taken from a large but specific list of forms. `setf` does not use the `value` of the form but the location where the `value` is stored. The list of forms includes `car` and `cdr`. `setf` places the value of its second argument in that location. For example, we may use `setf` along with the list functions to modify lists in the global environment, as the following transcript shows:

```
? (setq x '(a b c d e))
(a b c d e)
? (setf (nth 2 x) 3)
3
? x
(a b 3 d e)
```

We use `setf`, along with `get`, to change the value of properties. For instance, we may define the properties of a `rose` by:

```
> (setf (get 'rose 'color) 'red)
red
> (setf (get 'rose 'smell) 'sweet)
sweet
```

`remprop` takes as arguments a symbol and a property name and causes a named property to be deleted. For example:

```
> (get 'rose 'color)
red
> (remprop 'rose 'color)
color
> (get 'rose 'color)
nil
```

`symbol-plist` takes as argument a symbol and returns its property list. For example:

```
> (setf (get 'rose 'color) 'red)
red
> (setf (get 'rose 'smell) 'sweet)
sweet
> (symbol-plist 'rose)
(smell sweet color red)
```

Using property lists, it is straightforward to implement a semantic network. For example, the following calls to `setf` implement the semantic network description of species of birds from Figure 2.1. The `isa` relations define inheritance links.

```
(setf (get 'animal 'covering) 'skin)
(setf (get 'bird 'covering) 'feathers)
(setf (get 'bird 'travel) 'flies)
(setf (get 'bird 'isa) animal)
```

```
(setf (get 'fish 'isa) animal)
(setf (get 'fish 'travel) 'swim)
(setf (get 'ostrich 'isa) 'bird)
(setf (get 'ostrich 'travel) 'walk)
(setf (get 'penguin 'isa) 'bird)
(setf (get 'penguin 'travel) 'walk)
(setf (get 'penguin 'color) 'brown)
(setf (get 'opus 'isa) 'penguin)
(setf (get 'canary 'isa) 'bird)
(setf (get 'canary 'color) 'yellow)
(setf (get 'canary 'sound) 'sing)
(setf (get 'tweety 'isa) 'canary)
(setf (get 'tweety 'color) 'white)
(setf (get 'robin 'isa) 'bird)
(setf (get 'robin 'sound) 'sings)
(setf (get 'robin 'color) 'red)
```

Using this representation of semantic nets, we now define control functions for hierarchical inheritance. This is simply a search along `isa` links until a parent is found with the desired property. The parents are searched in a depth-first fashion, and search stops when an instance of the property is found. This is typical of the inheritance algorithms provided by many commercial systems. Variations on this approach include the use of breadth-first search as an inheritance search strategy.

`inherit-get` is a variation of `get` that first tries to retrieve a property from a symbol; if this fails, `inherit-get` calls `get-from-parents` to implement the search. `get-from-parents` takes as its first argument either a single parent or a list of parents; the second argument is a property name. If the parameter `parents` is `nil`, the search halts with failure. If `parents` is an atom, it calls `inherit-get` on the parent to either retrieve the property from the parent itself or continue the search. If `parents` is a list, `get-from-parents` calls itself recursively on the `car` and `cdr` of the list of `parents`. The tree walk based function `inherit-get` is defined by:

```
(defun inherit-get (object property)
    (or (get object property)
       (get-from-parents (get object 'isa)
                         property)))
(defun get-from-parents (parents property)
    (cond ((null parents) nil)
          ((atom parents)
                (inherit-get parents property))
          (t (or (get-from-parents (car parents)
                                   property)
                 (get-from-parents (cdr parents)
                                   property)))))
```

In the next section we generalize our representations for things, classes, and inheritance using the CLOS object-oriented programming library.

18.2 Object-Oriented Programming Using CLOS

Object-Orientation Defined

In spite of the many advantages of functional programming, some problems are best conceptualized in terms of objects that have a state that changes over time. Simulation programs are typical of this. Imagine trying to build a program that will predict the ability of a steam heating system to heat a large building: we can simplify the problem by thinking of it as a system of objects (rooms, thermostats, boilers, steam pipes, etc.) that interact to change the temperature and behavior of each other over time. Object-oriented languages support an approach to problem solving that lets us decompose a problem into interacting objects. These objects have a state that can change over time, and a set of functions or methods that define the object's behaviors. Essentially, object-oriented programming lets us solve problems by constructing a model of the problem domain as we understand it. This model-based approach to problem solving is a natural fit for artificial intelligence, an effective programming methodology in its own right, and a powerful tool for thinking about complex problem domains.

There are a number of languages that support object-oriented programming. Some of the most important are Smalltalk, C++, Java and the Common Lisp Object System (CLOS). At first glance, Lisp, with its roots in functional programming, and object orientation, with its emphasis on creating objects that retain their state over time, may seem worlds apart. However, many features of the language, such as dynamic type checking and the ability to create and destroy objects dynamically, make it an ideal foundation for constructing an object-oriented language. Indeed, Lisp was the basis for many of the early object-oriented languages, such as Smalltalk, Flavors, KEE, and ART. As the Common Lisp standard was developed, the Lisp community has accepted CLOS as the preferred way to do object-oriented programming in Lisp.

In order to fully support the needs of object-oriented programming, a programming language must provide three capabilities: 1) *encapsulation*, 2) *polymorphism*, and 3) *inheritance*. The remainder of this introduction describes these capabilities and an introduction to the way in which CLOS supports them.

> **Encapsulation**. All modern programming languages allow us to create complex data structures that combine atomic data items into a single entity. Object-oriented encapsulation is unique in that it combines both data items and the procedures used for their manipulation into a single structure, called a *class*. For example, the abstract data types seen previously (e.g., Section 16.2) may quite properly be seen as classes. In some object-oriented languages, such as Smalltalk, the encapsulation of procedures (or methods as they are called in the object-oriented community) in the object definition is explicit. CLOS takes a different approach, using Lisp's type-checking to provide this same ability. CLOS implements methods as

generic functions. These functions check the type of their parameters to guarantee that they can only be applied to instances of a certain class. This gives us a logical binding of methods to their objects.

Polymorphism. The word polymorphic comes from the roots "poly", meaning *many*, and "morphos", meaning *form*. A function is polymorphic if it has many different behaviors, depending on the types of its arguments. Perhaps the most intuitive example of polymorphic functions and their importance is a simple drawing program. Assume that we define objects for each of the shapes (square, circle, line) that we would like to draw. A natural way to implement this is to define a method named draw for each object class. Although each individual method has a different definition, depending on the shape it is to draw, all of them have the same name. Every shape in our system has a draw behavior. This is much simpler and more natural than to define a differently named function (draw-square, draw-circle, etc.) for every shape. CLOS supports polymorphism through generic functions. A generic function is one whose behavior is determined by the types of its arguments. In our drawing example, CLOS enables us to define a generic function, draw, that includes code for drawing each of the shapes defined in the program. On evaluation, it checks the type of its argument and automatically executes the appropriate code.

Inheritance. Inheritance is a mechanism for supporting class abstraction in a programming language. It lets us define general classes that specify the structure and behavior of their specializations, just as the class "tree" defines the essential attributes of pine trees, poplars, oaks, and other different species. In Section 18.1, we built an inheritance algorithm for semantic networks; this demonstrated the ease of implementing inheritance using Lisp's built-in data structuring techniques. CLOS provides us with a more robust, expressive, built-in inheritance algorithm.

Defining Classes and Instances in CLOS

The basic data structure in CLOS is the `class`. A `class` is a specification for a set of object instances. We define classes using the `defclass` macro. `defclass` has the syntax:

```
(defclass <class-name> (<superclass-name>*)
      (<slot-specifier>*))
```

<`class-name`> is a symbol. Following the class name is a list of direct superclasses (called `superclass`); these are the class's immediate parents in the inheritance hierarchy. This list may be empty. Following the list of parent classes is a list of zero or more `slot-specifiers`. A `slot-specifier` is either the name of a `slot` or a list consisting of a `slot-name` and zero or more `slot-options`:

```
slot-specifier ::= slotname |
(slot-name [slot-option])
```

For instance, we may define a new class, `rectangle`, which has slots values for `length` and `width`:

```
> (defclass rectangle()
      (length width))
#<standard-class rectangle>
```

`make-instance` allows us to create instances of a class, taking as its argument a class name and returning an instance of that class. It is the instances of a class that actually store data values. We may bind a symbol, `rect`, to an instance of `rectangle` using `make-instance` and `setq`:

```
> (setq rect (make-instance 'rectangle))
#<rectangle #x286AC1>
```

The slot options in a `defclass` define optional properties of slots. Slot options have the syntax (where "|" indicates alternative options):

```
slot-option ::= :reader <reader-function-name>|
          :writer <writer-function-name>|
          :accessor <reader-function-name>|
          :allocation <allocation-type>|
          :initarg <initarg-name>|
          :initform <form>
```

We declare slot options using keyword arguments. Keyword arguments are a form of optional parameter in a Lisp function. The keyword, which always begins with a "`:`", precedes the value for that argument. Available slot options include those that provide `accessors` to a slot. The `:reader` option defines a function called `reader-function-name` that returns the value of a slot for an instance. The `:writer` option defines a function named `writer-function-name` that will write to the slot. `:accessor` defines a function that may read a slot value or may be used with `setf` to change its value.

In the following transcript, we define `rectangle` to have slots for `length` and `width`, with slot `accessors` `get-length` and `get-width`, respectively. After binding `rect` to an instance of `rectangle` using `make-instance`, we use the `accessor`, `get-length`, with `setf` to bind the `length` slot to a value of `10`. Finally, we use the `accessor` to read this value.

```
> (defclass rectangle ()
   ((length :accessor get-length)
   (width :accessor get-width)))
#<standard-class rectangle>
> (setq rect (make-instance 'rectangle))
#<rectangle #x289159>
> (setf (get-length rect) 10)
10
> (get-length rect)
10
```

In addition to defining **accessors**, we can access a slot using the primitive function **slot-value**. **slot-value** is defined for all slots; it takes as arguments an instance and a slot name and returns the value of that slot. We can use it with **setf** to change the slot value. For example, we could use **slot-value** to access the **width** slot of **rect**:

```
> (setf (slot-value rect 'width) 5)
5
> (slot-value rect 'width)
5
```

:allocation lets us specify the memory allocation for a slot. **allocation-type** may be either **:instance** or **:class**. If allocation type is **:instance**, then CLOS allocates a local slot for each instance of the type. If allocation type is **:class**, then all instances share a single location for this slot. In **:class** allocation, all instances will share the same value of the slot; changes made to the slot by any instance will affect all other instances. If we omit the **:allocation** specifier, allocation defaults to **:instance**.

:initarg allows us to specify an argument that we can use with **make-instance** to specify an initial value for a slot. For example, we can modify our definition of **rectangle** to allow us to initialize the **length** and **width** slots of instances:

```
> (defclass rectangle ()
    ((length :accessor get-length
                    :initarg init-length)
    (width :accessor get-width :initarg init-width)))
#<standard-class rectangle>
>(setq rect (make-instance 'rectangle
       'init-length 100 'init-width 50))
#<rectangle #x28D081>
> (get-length rect)
100
> (get-width rect)
50
```

:initform lets us specify a form that CLOS evaluates on each call to **make-instance** to compute an initial value of the slot. For example, if we would like our program to ask the user for the values of each new instance of rectangle, we may define a function to do so and include it in an **initform**:

```
> (defun read-value (query) (print query)(read))
read-value
> (defclass rectangle ()
       ((length :accessor get-length
              :initform (read-value "enter length"))
```

```
        (width :accessor get-width
               :initform (read-value "enter width"))))
#<standard-class rectangle>
> (setq rect (make-instance 'rectangle))
"enter length" 100
"enter width" 50
#<rectangle #x290461>
> (get-length rect)
100
> (get-width rect)
50
```

Defining Generic Functions and Methods

A generic function is a function whose behavior depends upon the type of its arguments. In CLOS, generic functions contain a set of *methods*, indexed by the type of their arguments. We call generic functions with a syntax similar to that of regular functions; the generic function retrieves and executes the method associated with the type of its parameters.

CLOS uses the structure of the class hierarchy in selecting a method in a generic function; if there is no method defined directly for an argument of a given class, it uses the method associated with the "closest" ancestor in the hierarchy. Generic functions provide most of the advantages of "purer" approaches of methods and message passing, including inheritance and overloading. However, they are much closer in spirit to the functional programming paradigm that forms the basis of Lisp. For instance, we can use generic functions with `mapcar`, `funcall`, and other higher-order constructs in the Lisp language.

We define generic functions using either `defgeneric` or `defmethod`. `defgeneric` lets us define a generic function and several methods using one form. `defmethod` enables us to define each method separately, although CLOS combines all of them into a single generic function. `defgeneric` has the (simplified) syntax:

```
(defgeneric f-name lambda-list <method-description>*)
<method-description> ::= (:method specialized-lambda-
     list form)
```

`defgeneric` takes a name of the function, a `lambda` list of its arguments, and a series of zero or more method descriptions. In a method description, `specialized-lambda-list` is just like an ordinary `lambda` list in a function definition, except that a formal parameter may be replaced with a (symbol parameter-specializer) pair: symbol is the name of the parameter, and parameter-specializer is the class of the argument. If an argument in a method has no parameter specializer, its type defaults to `t`, which is the most general class in a CLOS hierarchy. Parameters of type `t` can bind to any object. The specialized `lambda` list of each method specifier must have the same number of arguments as the `lambda` list in the `defgeneric`. A `defgeneric` creates a generic function with the specified methods, replacing any existing generic functions.

As an example of a generic function, we may define classes for `rectangle` and `circle` and implement the appropriate methods for finding `area`s:

```
(defclass rectangle ()
    ((length :accessor get-length
             :initarg init-length)
    (width :accessor get-width :initarg init-width)))
(defclass circle ()
    ((radius :accessor get-radius
             :initarg init-radius)))
(defgeneric area (shape)
    (:method ((shape rectangle))
       (* (get-length shape)
          (get-width shape)))
    (:method ((shape circle))
    (* (get-radius shape) (get-radius shape) pi)))
(setq rect (make-instance 'rectangle 'init-length 10
       'init-width 5))
(setq circ (make-instance 'circle 'init-radius 7))
```

We can use the `area` function to compute the `area` of either shape:

```
> (area rect)
50
> (area circ)
153.93804002589985
```

We can also define methods using `defmethod`. Syntactically, `defmethod` is similar to `defun`, except it uses a specialized `lambda` list to declare the class to which its arguments belong. When we define a method using `defmethod`, if there is no generic function with that name, `defmethod` creates one; if a generic function of that name already exists, `defmethod` adds a new method to it. For example, suppose we wish to add the class `square` to the above definitions, we can do this with:

```
(defclass square ()
      ((side :accessor get-side :initarg init-side)))
(defmethod area ((shape square))
      (* (get-side shape)
         (get-side shape)))
(setq sqr (make-instance 'square 'init-side 6))
```

`defmethod` does not change the previous definitions of the `area` function; it simply adds a new method to the generic function:

```
> (area sqr)
36
```

```
> (area rect)
50
> (area circ)
153.93804002589985
```

Inheritance in CLOS

CLOS is a multiple-inheritance language. Along with offering the program designer a very flexible representational scheme, multiple inheritance introduces the potential for creating anomalies when inheriting slots and methods. If two or more ancestors have defined the same method, it is crucial to know which method any instance of those ancestors will inherit. CLOS resolves potential ambiguities by defining a *class precedence* list, which is a total ordering of all classes within a class hierarchy.

Each `defclass` lists the direct parents of a class in left-to-right order. Using the order of direct parents for each class, CLOS computes a partial ordering of all the ancestors in the inheritance hierarchy. From this partial ordering, it derives the total ordering of the class precedence list through a topological sort. The precedence list follows two rules:

1. Any direct parent class precedes any more distant ancestor.
2. In the list of immediate parents of `defclass`, each class precedes those to its right.

CLOS computes the class precedence list for an object by topologically sorting its ancestor classes according to the following algorithm. Let `C` be the class for which we are defining the precedence list:

1. Let S_C be the set of `C` and all its superclasses.
2. For each class, `c`, in S_C, define the set of ordered pairs:

   $R_c = \{(c, c_1), (c_1, c_2), \ldots (c_{n-1}, c_n)\}$

   where c_1 through c_n are the direct parents of `c` in the order they are listed in `defclass`. Note that each R_c defines a *total order*.
3. Let `R` be the union of the R_cs for all elements of S_c. `R` may or may not define a partial ordering. If it does not define a partial ordering, then the hierarchy is inconsistent and the algorithm will detect this.
4. Topologically sort the elements of `R` by:
   a. Begin with an empty precedence list, `P`.
   b. Find a class in `R` having no predecessors. Add it to the end of `P` and remove the class from S_c and all pairs containing it from `R`. If there are several classes in S_c with no predecessor, select the one that has a direct subclass nearest the end in the current version of `P`.
   c. Repeat the two previous steps until no element can be found that has no predecessor in R.

d. If S_c is not empty, then the hierarchy is inconsistent; it may contain ambiguities that cannot be resolved using this technique.

Because the resulting precedence list is a total ordering, it resolves any ambiguous orderings that may have existed in the class hierarchy. CLOS uses the class precedence list in the inheritance of slots and the selection of methods.

In selecting a method to apply to a given call of a generic function, CLOS first selects all applicable methods. A method is applicable to a generic function call if each parameter specializer in the method is consistent with the corresponding argument in the generic function call. A parameter specializer is consistent with an argument if the specializer either matches the class of the argument or the class of one of its ancestors.

CLOS then sorts all applicable methods using the precedence lists of the arguments. CLOS determines which of two methods should come first in this ordering by comparing their parameter specializers in a left-to-right fashion. If the first pair of corresponding parameter specializers are equal, CLOS compares the second, continuing in this fashion until it finds corresponding parameter specializers that are different. Of these two, it designates as more specific the method whose parameter specializer appears leftmost in the precedence list of the corresponding argument. After ordering all applicable methods, the default method selection applies the most specific method to the arguments. For more details, see Steele (1990).

18.3 CLOS Example: A Thermostat Simulation

The properties of object-oriented programming that make it a natural way to organize large and complex software implementations are equally applicable in the design of knowledge bases. In addition to the benefits of class inheritance for representing taxonomic knowledge, the message-passing aspect of object-oriented systems simplifies the representation of interacting components.

As a simple example, consider the task of modeling the behavior of a steam heater for a small office building. We may naturally view this problem in terms of interacting components. For example:

- Each office has a thermostat that turns the heat in that office on and off; this functions independently of the thermostats in other offices.
- The boiler for the heating plant turns itself on and off in response to the heat demands made by the offices.
- When the demand on the boiler increases, there may be a time lag while more steam is generated.
- Different offices place different demands on the system; for example, corner offices with large windows lose heat faster than inner offices. Inner offices may even gain heat from their neighbors.

- The amount of steam that the system may route to a single office is affected by the total demand on the system.

These points are only a few of those that must be taken into account in modeling the behavior of such a system; the possible interactions are extremely complex. An object-oriented representation allows the programmer to focus on describing one class of objects at a time. We would represent thermostats, for example, by the temperature at which they call for heat, along with the speed with which they respond to changes in temperature.

The steam plant could be characterized in terms of the maximum amount of heat it can produce, the amount of fuel used as a function of heat produced, the amount of time it takes to respond to increased heat demand, and the rate at which it consumes water.

A room could be described in terms of its volume, the heat loss through its walls and windows, the heat gain from neighboring rooms, and the rate at which the radiator adds heat to the room.

The knowledge base is built up of classes such as `room` and `thermostat`, which define the properties of the class, and instances such as `room-322` and `thermostat-211`, which model individual situations.

The interactions between components are described by messages between instances. For example, a change in `room` temperature would cause a message to be sent to an instance of the class `thermostat`. If this new `temperature` is low enough, the `thermostat` would switch after an appropriate delay. This would cause a message to be sent to the `heater` requesting more heat. This would cause the `heater` to consume more oil, or, if already operating at maximum capacity, to route some heat away from other rooms to respond to the new demand. This would cause other `thermostat`s to turn on, and so forth.

Using this simulation, we can test the ability of the system to respond to external changes in temperature, measure the effect of heat loss, or determine whether the projected heating is adequate. We could use this simulation in a diagnostic program to verify that a hypothesized fault could indeed produce a particular set of symptoms. For example, if we have reason to believe that a heating problem is caused by a blocked steam pipe, we could introduce such a fault into the simulation and see whether it produces the observed symptoms.

The significant thing about this example is the way in which an object-oriented approach allows knowledge engineers to deal with the complexity of the simulation. It enables them to build the model a piece at a time, focusing only on the behaviors of simple classes of objects. The full complexity of the system behavior emerges when we execute the model.

The basis of our CLOS implementation of this model is a set of object definitions. `Thermostat`s have a single slot called `setting`. The `setting` of each instance is initialized to `65` using `initform`. `heater-thermostat` is a subclass of `thermostat` for controlling `heaters` (as opposed to air conditioners); they have a single slot that will be bound to an instance of the `heater` class. Note that the `heater` slot has a class allocation; this captures the constraint that the `thermostat`s in different rooms of a building control the single building's `heater-obj`.

```
(defclass thermostat ()
      ((setting :initform 65
      :accessor therm-setting)))

(defclass heater-thermostat (thermostat)
      ((heater :allocation :class
      :initarg heater-obj)))
```

A `heater` has a state (`on` or `off`) that is initialized to `off`, and a `location`. It also has a slot, `rooms-heated`, that will be bound to a list of objects of type `room`. Note that instances, like any other structure in Lisp, may be elements of a list.

```
(defclass heater ()
      ((state :initform 'off
      :accessor heater-state)
      (location :initarg loc)
      (rooms-heated)))
```

`room` has slots for `temperature`, initialized to `65` degrees; `thermostat`, which will be bound to an instance of `thermostat`; and `name`, the name of `room`.

```
(defclass room ()
      ((temperature :initform 65
         :accessor room-temp)
      (thermostat :initarg therm
         :accessor room-thermostat)
      (name :initarg name
         :accessor room-name)))
```

These class definitions define the hierarchy of Figure 18.2.

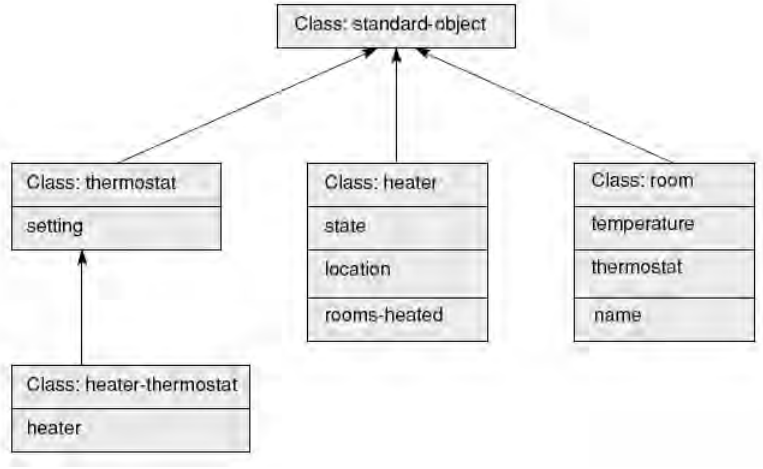

Figure 18.2. A class hierarchy for the room/heater/thermostat simulation.

We represent our particular simulation as a set of instances of these classes. We will implement a simple system of one `room`, one `heater`, and one `thermostat`:

```
(setf office-heater (make-instance 'heater 'loc
'office))

(setf room-325 (make-instance 'room
   'therm (make-instance 'heater-thermostat
      'heater-obj office-heater)
   'name 'room-325))

(setf (slot-value office-heater 'rooms-heated) (list
room-325))
```

Figure 18.3 shows the definition of instances, the allocation of slots, and the bindings of slots to values.

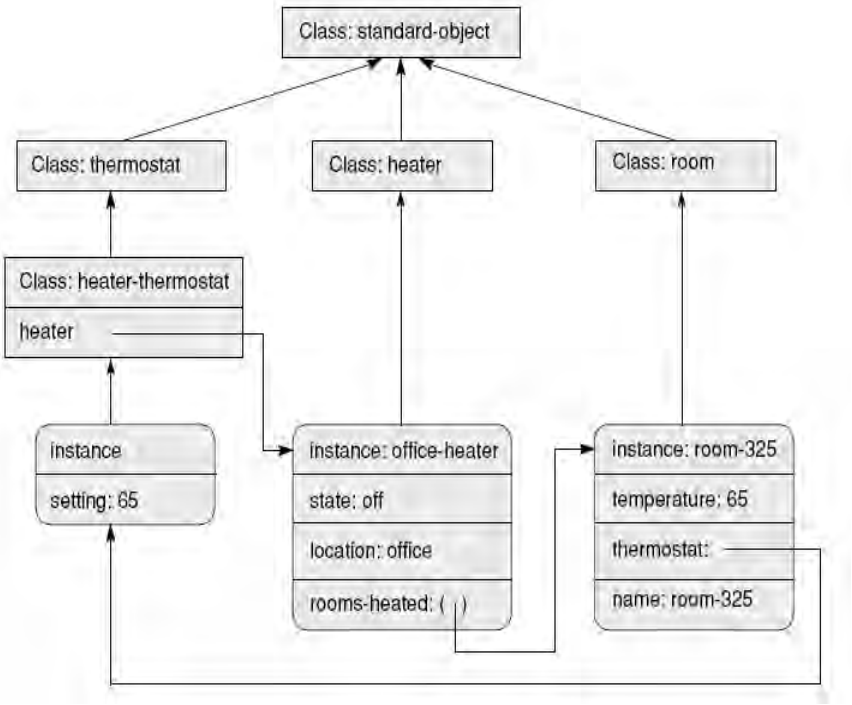

Figure 18.3. The creation of instances and binding of slots in the simulation.

We define the behavior of `room`s through the methods `change-temp`, `check-temp`, and `change-setting`. `change-temp` sets the `temperature` of a `room` to a new value, prints a message to the user, and calls `check-temp` to determine whether the `heater` should come on. Similarly, `change-setting` changes the thermostat setting, `therm-setting`, and calls `check-temp`, which simulates the `thermostat`. If the `temperature` of the `room` is less than the thermostat setting, it sends the `heater` a message to turn `on`; otherwise it sends an `off` message.

```
(defmethod change-temp ((place room) temp-change)
   (let ((new-temp (+ (room-temp place)
                      temp-change)))
         (setf (room-temp place) new-temp)
         (terpri)
         (prin1 "the temperature in")
         (prin1 (room-name place))
         (prin1 " is now ")
         (prin1 new-temp)
         (terpri)
         (check-temp place)))
(defmethod change-setting ((room room) new-setting)
      (let ((therm (room-thermostat room)))
         (setf (therm-setting therm) new-setting)
         (prin1 "changing setting of thermostat in")
         (prin1 (room-name room))
         (prin1 " to ")
         (prin1 new-setting)
         (terpri)
         (check-temp room)))
(defmethod check-temp ((room room))
      (let* ((therm (room-thermostat room))
            (heater (slot-value therm 'heater)))
            (cond ((> (therm-setting therm)
                      (room-temp room))
                   (send-heater heater 'on))
                  (t (send-heater heater 'off)))))
```

The heater methods control the state of the heater and change the temperature of the rooms. `send-heater` takes as arguments an instance of `heater` and a message, `new-state`. If `new-state` is `on` it calls the `turn-on` method to start the `heater`; if `new-state` is `off` it shuts the `heater` down. After turning the heater on, `send-heater` calls `heat-rooms` to increase the temperature of each room by one degree.

```
(defmethod send-heater ((heater heater) new-state)
      (case new-state
         (on (if (equal (heater-state heater) 'off)
             (turn-on heater))
             (heat-rooms (slot-value heater
                                     'rooms-heated) 1))
```

```
          (off (if (equal (heater-state heater) 'on)
                   (turn-off heater)))))
(defmethod turn-on ((heater heater))
      (setf (heater-state heater) 'on)
      (prin1 "turning on heater in")
      (prin1 (slot-value heater 'location))
      (terpri))
(defmethod turn-off ((heater heater))
      (setf (heater-state heater) 'off)
      (prin1 "turning off heater in")
      (prin1 (slot-value heater 'location))
      (terpri))
(defun heat-rooms (rooms amount)
      (cond ((null rooms) nil)
            (t (change-temp (car rooms) amount)
               (heat-rooms (cdr rooms) amount))))
```

The following transcript illustrates the behavior of the simulation.

```
> (change-temp room-325 5)
"the temperature in "room-325" is now "60
"turning on heater in "office
"the temperature in "room-325" is now "61
"the temperature in "room-325" is now "62
"the temperature in "room-325" is now "63
"the temperature in "room-325" is now "64
"the temperature in "room-325" is now "65
"turning off heater in "office
nil
> (change-setting room-325 70)
"changing setting of thermostat in "room-325" to "70
"turning on heater in "office
"the temperature in "room-325" is now "66
"the temperature in "room-325" is now "67
"the temperature in "room-325" is now "68
"the temperature in "room-325" is now "69
"the temperature in "room-325" is now "70
"turning off heater in "office
nil
```

Exercises

1. Create two semantic network representations (Section 17.1) for an application of your choice. Build the representation first using association lists and then build it using property lists. Comment on the differences in these two approaches for representing semantic information.

2. Add to the CLOS simulation of Section 18.3 a cooling system so that if any room's temperature gets above a certain temperature it starts to cool. Also add a "thermal" factor to each room so that it heats and cools as a function of its volume and insulation value.

3. Create a CLOS simulation in another domain, e.g., a building that has both heating and cooling. You can add specifics to each room such as an insulation value that mitigates heat/cooling loss. See the discussion at the beginning of Section 18.3 for parameters you might build in to your augmented system.

4. Create a CLOS simulation for an ecological situation. For example, you might have classes for grass, wolves, cattle, and weather. Then make a set of rules that balances their ecological survival across time.

19 Machine Learning in Lisp

Chapter Objectives

ID3 algorithm and inducing decision trees from lists of examples.
A basic Lisp implementation of ID3
Demonstration on a simple credit assessment example.

Chapter Contents

19.1 Learning: The ID3 Algorithm

In this section, we implement the ID3 induction algorithm described in Luger (2009, Section 10.3). ID3 infers decision trees from a set of training examples, which enables classification of an object on the basis of its properties. Each internal node of the decision tree tests one of the properties of a candidate object, and uses the resulting value to select a branch of the tree. It continues through the nodes of the tree, testing various properties, until it reaches a leaf, where each leaf node denotes a classification. ID3 uses an information theoretic test selection function to order tests so as to construct a (nearly) optimal decision tree. See Table 19.1 for a sample data set and Figure 19.1 for an ID3 induced decision tree. The details for the tree induction algorithms may be found in Luger (2009, Section 10.3) and in Quinlan (1986).

The ID3 algorithm requires that we manage a number of complex data structures, including objects, properties, sets, and decision trees. The heart of our implementation is a set of structure definitions, aggregate data types similar to records in the Pascal language or structures in C. Using `defstruct`, Common Lisp allows us to define types as collections of named slots; `defstruct` constructs functions needed to create and manipulate objects of that type.

Along with the use of structures to define data types, we exploit higher order functions such as `mapcar`. As the stream-based approach to our expert system shell demonstrated, the use of maps and filters to apply functions to lists of objects can often capture the intuition behind an algorithm with greater clarity than less expressive programming styles. The ability to treat functions as data, to bind function closures to symbols and process them using other functions, is a cornerstone of Lisp programming style.

A Credit History Example

This chapter will demonstrate the ID3 implementation using a simple credit assessment example. Suppose we want to determine a person's credit risk (high, moderate, low) based on data recorded from past loans. We can represent this as a decision tree, where each node examines one aspect of a person's credit profile. For example, if one of the factors we care about is

A Credit History Example

This chapter will demonstrate the ID3 implementation using a simple credit assessment example. Suppose we want to determine a person's credit risk (high, moderate, low) based on data recorded from past loans. We can represent this as a decision tree, where each node examines one aspect of a person's credit profile. For example, if one of the factors we care about is collateral, then the collateral node will have two branches: no collateral and adequate collateral.

The challenge a machine learning algorithm faces is to construct the "best" decision tree given a set of training examples. Exhaustive training sets are rare in machine learning, either because the data is not available, or because such sets would be too large to manage effectively. ID3 builds decision trees under the assumption that the simplest tree that correctly classifies all training instances is most likely to be correct on new instances, since it makes the fewest assumptions from the training data. ID3 infers a simple tree from training data using a greedy algorithm: select the test property that gives the most information about the training set, partition the problem on this property and recur. The implementation that we present illustrates this algorithm.

We will test our algorithm on the data of table 19.1.

No.	Risk	Credit History	Debt	Collateral	Income
1.	high	bad	high	none	$0 to $15k
2.	high	unknown	high	none	$15k to $35k
3.	moderate	unknown	low	none	$15k to $35k
4.	high	unknown	low	none	$0 to $15k
5.	low	unknown	low	none	over $35k
6.	low	unknown	low	adequate	over $35k
7.	high	bad	low	none	$0 to $15k
8.	moderate	bad	low	adequate	over $35k
9.	low	good	low	none	over $35k
10.	low	good	high	adequate	over $35k
11.	high	good	high	none	$0 to $15k
12.	moderate	good	high	none	$15k to $35k
13.	low	good	high	none	over $35k
14.	high	bad	high	none	$15k to $35k

Table 19.1 Training data for the credit example

Figure 19.1 shows a decision tree that correctly classifies this data.

`defstruct` allows us to create structure data items in Lisp. For example, using `defstruct`, we can define a new *data type*, `employee`, by evaluating a form. `employee` is the name of the defined type; `name`,

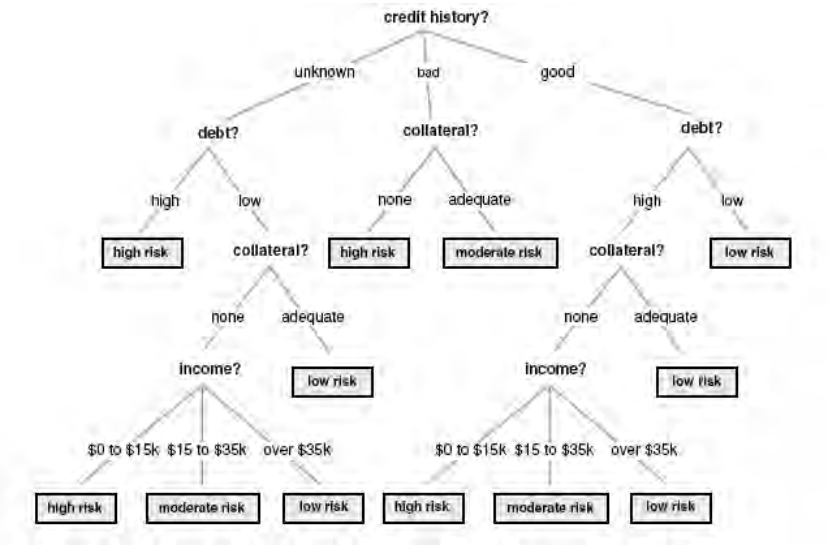

Figure 19.1 A decision tree that covers the data of Table 19.1

Defining Structures Using Defstruct

```
(defstruct employee
    name
    address
    serial-number
    department
    salary)
```

Here, `defstruct` takes as its arguments a symbol, which will become the name of a new type, and a number of slot specifiers. Here, we have defined five slots by name; slot specifiers also allow us to define different properties of slots, including type and initialization information, see Steele (1990).

Evaluating the `defstruct` form has a number of effects, for example:

```
(defstruct <type name>
    <slot name 1>
    <slot name 2>
       …
    <slot name n>)
```

`defstruct` defines a function, named according to the scheme: make-`<type name>`, that lets us create instances of this type. For example, after defining the structure, `employee`, we may bind `new-employee` to an object of this type by evaluating:

```
(setq new-employee (make-employee))
```

We can also use slot names as keyword arguments to the `make` function, giving the instance initial values. For example:

```
(setq new-employee
    (make-employee
        :name '(Doe Jane)
        :address "1234 Main, Randolph, Vt"
        :serial-number 98765
        :department 'Sales
        :salary 4500.00))
```

`defstruct` makes `<type name>` the name of a data type. We may use this name with `typep` to test if an object is of that type, for example:

```
> (typep new-employee 'employee)
t
```

Furthermore, `defstruct` defines a function, `<type-name>-p`, which we may also use to test if an object is of the defined type. For instance:

```
> (employee-p new-employee)
t
> (employee-p '(Doe Jane))
nil
```

Finally, `defstruct` defines an accessor for each slot of the structure. These accessors are named according to the scheme:

```
<type name>-<slot name>
```

In our example, we may access the values of various slots of `new-employee` using these accessors:

```
> (employee-name new-employee)
(Doe Jane)
> (employee-address new-employee)
"1234 Main, Randolph, Vt"
> (employee-department new-employee)
Sales
```

We may also use these accessors in conjunction with `setf` to change the slot values of an instance. For example:

```
> (employee-salary new-employee)
4500.0
> (setf (employee-salary new-employee) 5000.00)
5000.0
> (employee-salary new-employee)
5000.0
```

So we see that using structures, we can define predicates and accessors of a data type in a single Lisp form. These definitions are central to our implementation of the ID3 algorithm.

When given a set of examples of known classifications, we use the sample information offered in Table 19.1, ID3 induces a tree that will correctly classify all the training instances, and has a high probability of correctly

classifying new people applying for credit, see Figure 19.1. In the discussion of ID3 in Luger (2009, Section 10.3), training instances are offered in a tabular form, explicitly listing the properties and their values for each instance. Thus, Table 19.1 lists a set of instances for learning to predict an individual's credit risk. Throughout this section, we will continue to refer to this data set.

Tables are only one way of representing examples; it is more general to think of them as objects that may be tested for various properties. Our implementation makes few assumptions about the representation of objects. For each property, it requires a function of one argument that may be applied to an object to return a value of that property. For example, if `credit-profile-1` is bound to the first example in Table 19.1, and `history` is a function that returns the value of an object's credit history, then:

```
> (history credit-profile-1)
bad
```

Similarly, we require functions for the other properties of a credit profile:

```
> (debt credit-profile-1)
high
> (collateral credit-profile-1)
none
> (income credit-profile-1)
0-to-15k
> (risk credit-profile-1)
high
```

Next we select a representation for the credit assignment example, making objects as association lists in which the keys are property names and their data are property values. Thus, the first example of Table 19.1 is represented by the association list:

```
((risk . high) (history . bad) (debt . high)
(collateral . none)(income . 0-15k))
```

We now use `defstruct` to define instances as structures. We represent the full set of training instances as a list of association lists and bind this list to examples:

```
(setq examples
    '(((risk . high) (history . bad) (debt . high)
         (collateral . none) (income . 0-15k))
     ((risk . high) (history . unknown)
         (debt . high)(collateral . none)
         (income . 15k-35k))
     ((risk . moderate) (history . unknown)
         (debt . low) (collateral . none)
         (income . 15k-35k))
    ((risk . high) (history . unknown) (debt . low)
         (collateral . none) (income . 0-15k))
```

```
((risk . low) (history . unknown) (debt . low)
     (collateral . none) (income . over-35k))
((risk . low) (history . unknown) (debt . low)
    (collateral . adequate)
    (income . over-35k))
((risk . high) (history . bad) (debt . low)
     (collateral . none) (income . 0-15k))
((risk . moderate) (history . bad) (debt . low)
     (collateral . adequate)
    (income . over-35k))
((risk . low) (history . good) (debt . low)
     (collateral . none) (income . over-35k))
((risk . low) (history . good) (debt . high)
    (collateral . adequate) (income . over-35k))
((risk . high) (history . good) (debt . high)
    (collateral . none) (income . 0-15k))
((risk . moderate) (history . good)
    (debt . high) (collateral . none)
    (income . 15k-35k))
((risk . low) (history . good) (debt . high)
    (collateral . none) (income . over-35k))
((risk . high) (history . bad) (debt . high)
    (collateral . none) (income . 15k-35k))))
```

Since the purpose of a decision tree is the determination of `risk` for a new individual, `test-instance` will include all properties except `risk`:

```
(setq test-instance
      '((history . good) (debt . low)
        (collateral . none) (income . 15k-35k)))
```

Given this representation of `objects`, we next define `property`:

```
(defun history (object)
       (cdr (assoc 'history object :test #'equal)))
(defun debt (object)
       (cdr (assoc 'debt object :test #'equal)))
(defun collateral (object)
       (cdr (assoc 'collateral object :test
             #'equal)))
(defun income (object)
       (cdr (assoc 'income object :test #'equal)))
(defun risk (object)
       (cdr (assoc 'risk object :test #'equal)))
```

A `property` is a function on `objects`; we represent these functions as a slot in a structure that includes other useful information:

```
(defstruct property
    name
    test
    values)
```

The `test` slot of an instance of `property` is bound to a function that returns a `property` value. `name` is the name of the property, and is included solely to help the user inspect definitions. `values` is a list of all the values that may be returned by `test`. Requiring that the values of each property be known in advance simplifies the implementation greatly, and is not unreasonable.

We now define `decision-tree` using the following structures:

```
(defstruct decision-tree
     test-name
     test
     branches)
(defstruct leaf
     value)
```

Thus `decision-tree` is either an instance of `decision-tree` or an instance of `leaf`. `leaf` has one slot, a `value` corresponding to a classification. Instances of type `decision-tree` represent internal nodes of the tree, and consist of a `test`, a `test-name` and a set of `branches`. `test` is a function of one argument that takes an object and returns the value of a property. In classifying an object, we apply `test` to it using `funcall` and use the returned value to select a branch of the tree. `test-name` is the name of the property. We include it to make it easier for the user to inspect decision trees; it plays no real role in the program's execution. `branches` is an association list of possible subtrees: the keys are the different values returned by `test`; the data are subtrees.

For example, the tree of Figure 19.1 would correspond to the following set of nested structures. The `#S` is a convention of Common Lisp I/O; it indicates that an s-expression represents a structure.

```
#S(decision-tree
    :test-name income
    :test #<Compiled-function income #x3525CE>
    :branches
        ((0-15k . #S(leaf :value high))
        (15k-35k . #S(decision-tree
           :test-name history
           :test
           #<Compiled-function history #x3514D6>
           :branches
```

```
((good . #S(leaf :value moderate))
(bad . #S(leaf :value high))
(unknown . #S(decision-tree
    :test-name debt
    :test
    #<Compiled-function debt #x351A7E>
    :branches
     ((high . #S(leaf :value high))
     (low . #S(leaf
         :value moderate)))))))
(over-35k . #S(decision-tree
    :test-name history
    :test
         #<Co…d-fun.. history #x3514D6>
    :branches
         ((good . #S(leaf :value low))
         (bad . #S(leaf :value
                         moderate))
         (unknown . #S(leaf :value
                         low)))))))
```

Although a set of training examples is, conceptually, just a collection of objects, we will make it part of a structure that includes slots for other information used by the algorithm. We define `example-frame` as:

```
(defstruct example-frame
       instances
       properties
       classifier
       size
       information)
```

`instances` is a list of objects of known classification; this is the training set used to construct a decision tree. `properties` is a list of objects of type `property`; these are the `properties` that may be used in the nodes of that tree. `classifier` is also an instance of `property`; it represents the classification that ID3 is attempting to learn. Since the examples are of known classification, we include it as another `property`. `size` is the number of examples in the `instances` slot; `information` is the information content of that set of examples. We compute `size` and `information` content from the examples. Since these values take time to compute and will be used several times, we save them in these slots.

ID3 constructs trees recursively. Given a set of examples, each an instance of `example-frame`, it selects a property and uses it to partition the set of training instances into non-intersecting subsets. Each subset contains all the instances that have the same value for that property. The property

selected becomes the test at the current node of the tree. For each subset in the partition, ID3 recursively constructs a subtree using the remaining properties. The algorithm halts when a set of examples all belong to the same class, at which point it creates a leaf.

Our final structure definition is `partition`, a division of an example set into subproblems using a particular property. We define the type `partition`:

```
(defstruct partition
       test-name
       test
       components
       info-gain)
```

In an instance of `partition`, the `test` slot is bound to the property used to create the `partition`. `test-name` is the name of the `test`, included for readability. `components` will be bound to the subproblems of the `partition`. In our implementation, `components` is an association list: the keys are the different values of the selected `test`; each datum is an instance of `example-frame`. `info-gain` is the information gain that results from using `test` as the node of the tree. As with `size` and `information` in the `example-frame` structure, this slot caches a value that is costly to compute and is used several times in the algorithm. By organizing our program around these data types, we make our implementation more clearly reflect the structure of the algorithm.

19.2 Implementing ID3

The heart of our implementation is the function `build-tree`, which takes an instance of `example-frame`, and recursively constructs a decision tree.

```
(defun build-tree (training-frame)
    (cond
                          ;Case 1: empty example set.
    ((null (example-frame-instances training-frame))
         (make-leaf :value
              "unable to classify: no examples"))
                    ;Case 2: all tests have been used.
    ((null (example-frame-properties
                       training-frame))
         (make-leaf :value (list-classes
                       training-frame)))
                  ;Case 3: all examples in same class.
    ((zerop (example-frame-information
                       training-frame))
         (make-leaf :value (funcall (property-test
```

```
                (example-frame-classifier
                        training-frame))
                    (car (example-frame-instances
                              training-frame)))))
                       ;Case 4: select test and recur.
        (t (let ((part (choose-partition
             (gen-partitions training-frame))))
                (make-decision-tree
                   :test-name
                         (partition-test-name part)
                   :test (partition-test part)
                   :branches (mapcar #'(lambda (x)
                        (cons (car x)
                          (build-tree (cdr x))))
                          (partition-components
                                          part)))))))
```

Using `cond`, `build-tree` analyzes four possible cases. In case 1, the example frame does not contain any training instances. This might occur if ID3 is given an incomplete set of training examples, with no instances for a given value of a property. In this case it creates a leaf consisting of the message: "`unable to classify: no examples`".

The second case occurs if the `properties` slot of `training-frame` is empty. In recursively building the decision tree, once the algorithm selects a property, it deletes it from the `properties` slot in the example frames for all subproblems. If the example `set` is inconsistent, the algorithm may exhaust all properties before arriving at an unambiguous classification of training instances. In this case, it creates a leaf whose value is a list of all classes remaining in the set of training instances.

The third case represents a successful termination of a branch of the tree. If `training-frame` has an information content of zero, then all of the examples belong to the same class; this follows from Shannon's definition of information, see Luger (2009, Section 13.3). The algorithm halts, returning a leaf node in which the value is equal to this remaining class.

The first three cases terminate tree construction; the fourth case recursively calls `build-tree` to construct the subtrees of the current node. `gen-partitions` produces a list of all possible partitions of the example set, using each test in the properties slot of `training-frame`. `choose-partition` selects the test that gives the greatest information gain. After binding the resulting partition to the variable part in a `let` block, `build-tree` constructs a node of a decision tree in which the test is that used in the chosen partition, and the `branches` slot is bound to an association list of subtrees. Each key in `branches` is a value of the test and each datum is a decision tree constructed by a recursive call to `build-tree`. Since the `components` slot of `part` is already an association list in which the keys are property values and the data are instances of `example-frame`, we implement the construction of subtrees using `mapcar` to apply `build-tree` to each datum in this association list.

`gen-partitions` takes one argument, `training-frame`, an object of type `example-frame-properties`, and generates all partitions of its instances. Each `partition` is created using a different property from the `properties` slot. `gen-partitions` employs a function, `partition`, that takes an instance of an example frame and an instance of a property; it partitions the examples using that property. Note the use of `mapcar` to generate a partition for each element of the `example-frame-properties` slot of `training-frame`.

```
(defun gen-partitions (training-frame)
       (mapcar #'(lambda (x)
             (partition training-frame x))
         (example-frame-properties training-frame)))
```

`choose-partition` searches a list of candidate partitions and chooses the one with the highest information gain:

```
(defun choose-partition (candidates)
    (cond ((null candidates) nil)
         ((= (list-length candidates) 1)
                (car candidates))
          (t (let ((best (choose-partition
                       (cdr candidates))))
      (if (> (partition-info-gain (car candidates))
          (partition-info-gain best))
              (car candidates) best)))))
```

`partition` is the most complex function in the implementation. It takes as arguments an example frame and a `property`, and returns an instance of a `partition` structure:

```
(defun partition (root-frame property)
      (let ((parts (mapcar #'(lambda (x)
             (cons x (make-example-frame)))
                  (property-values property))))
          (dolist (instance
             (example-frame-instances root-frame))
            (push instance (example-frame-instances
                (cdr (assoc (funcall
                      (property-test property)
                                  instance)
                     parts)))))
          (mapcar #'(lambda (x)
             (let ((frame (cdr x)))
             (setf (example-frame-properties frame)
                   (remove property
                      (example-frame-properties
                                 root-frame)))
```

```
                (setf (example-frame-classifier frame)
                          (example-frame-classifier
                                    root-frame))
                (setf (example-frame-size frame)
                 (list-length
                     (example-frame-instances frame)))
                (setf
                     (example-frame-information frame)
                     (compute-information
                     (example-frame-instances frame)
                     (example-frame-classifier
                                    root-frame)))))
          parts)
    (make-partition
           :test-name (property-name property)
           :test (property-test property)
           :components parts
           :info-gain
    (compute-info-gain root-frame parts))))
```

`partition` begins by defining a local variable, `parts`, using a `let` block. It initializes `parts` to an association list whose keys are the possible values of the test in `property`, and whose data will be the subproblems of the `partition`. `partition` implements this using the `do`list macro. `do`list binds local variables to each element of a list and evaluates its body for each binding At this point, they are empty instances of `example-frame`: the instance slots of each subproblem are bound to `nil`. Using a `do`list form, `partition` pushes each element of the instances slot of `root-frame` onto the instances slot of the appropriate subproblem in `parts`. `push` is a Lisp macro that modifies a list by adding a new first element; unlike `cons`, `push` permanently adds a new element to the list.

This section of the code accomplishes the actual partitioning of `root-frame`. After the `do`list terminates, `parts` is bound to an association list in which each key is a value of `property` and each datum is an example frame whose instances share that value. Using `mapcar`, the algorithm then completes the information required of each example frame in `parts`, assigning appropriate values to the `properties`, `classifier`, `size` and `information` slots. It then constructs an instance of `partition`, binding the `components` slot to `parts`.

`list-classes` is used in case 2 of `build-tree` to create a leaf node for an ambiguous classification. It employs a `do` loop to enumerate the `classes` in a list of examples. The `do` loop initializes `classes` to all the values of the classifier in `training-frame`. For each element of `classes`, it adds it to `classes-present` if it can find an element of the instances slot of `training-frame` that belongs to that class.

```
(defun list-classes (training-frame)
      (do
        ((classes (property-values
            (example-frame-classifier
                training-frame)) (cdr classes))
         (classifier (property-test
            (example-frame-classifier
                training-frame))) classes-present)
          ((null classes) classes-present)
          (if (member (car classes)
            (example-frame-instances
                training-frame)
                :test #'(lambda (x y)
                     (equal x (funcall
                         classifier y))))
           (push (car classes) classes-present))))
```

The remaining functions compute the information content of `examples`. `compute-information` determines the information content of a list of `examples`. It counts the number of instances in each class, and computes the proportion of the total training set belonging to each class. Assuming this proportion equals the probability that an object belongs to a class, it computes the information content of examples using Shannon's definition:

```
(defun compute-information (examples classifier)
    (let ((class-count
     (mapcar #'(lambda (x) (cons x 0))
           (property-values classifier))) (size 0))
    ;count number of instances in each class
     (dolist (instance examples)
           (incf size) (incf (cdr (assoc
             (funcall (property-test classifier)
                      instance) class-count))))
    ;compute information content of examples
    (sum #'(lambda (x) (if (= (cdr x) 0) 0
             (* -1
             (/ (cdr x) size)
             (log (/ (cdr x) size) 2))))
                      class-count)))
```

`compute-info-gain` gets the information gain of a partition by subtracting the weighted average of the information in its components from that of its parent examples.

```
(defun compute-info-gain (root parts)
       (- (example-frame-information root)
            (sum #'(lambda (x)
```

```
                (* (example-frame-information (cdr x))
                   (/ (example-frame-size (cdr x))
                        (example-frame-size root))))
          parts)))
```

`sum` computes the values returned by applying `f` to all elements of `list-of-numbers`:

```
(defun sum (f list-of-numbers)
       (apply '+ (mapcar f list-of-numbers)))
```

This completes the implementation of `build-tree`. The remaining component of the algorithm is a function, `classify`, that takes as arguments a decision tree as constructed by `build-tree`, and an object to be classified; it determines the classification of the object by recursively walking the tree. The definition of `classify` is straightforward: `classify` halts when it encounters a leaf, otherwise it applies the test from the current node to `instance`, and uses the result as the key to select a branch in a call to `assoc`.

```
(defun classify (instance tree)
    (if (leaf-p tree)
         (leaf-value tree)
         (classify instance
              (cdr (assoc
                   (funcall (decision-tree-test tree)
                            instance)
                   (decision-tree-branches tree))))))
```

Using the object definitions just defined, we now call `build-tree` on the credit example of Table 19.1. We bind tests to a list of property definitions for `history`, `debt`, `collateral` and `income`. `classifier` tests the risk of an instance. Using these definitions we bind the credit examples to an instance of `example-frame`.

```
(setq tests
       (list (make-property
            :name 'history
            :test #'history
            :values '(good bad unknown))
       (make-property
            :name 'debt
            :test #'debt
            :values '(high low))
       (make-property
            :name 'collateral
            :test #'collateral
            :values '(none adequate))
```

```
        (make-property
             :name 'income
             :test #'income
             :values
              '(0-to-15k 15k-to-35k over-35k))))
(setq classifier
        (make-property
              :name 'risk
              :test #'risk
              :values '(high moderate low)))
(setq credit-examples
        (make-example-frame
              :instances examples
              :properties tests
              :classifier classifier
              :size (list-length examples)
              :information (compute-information
examples classifier)))
```

Using these definitions, we may now induce decision trees, and use them to classify instances according to their credit risk:

```
> (setq credit-tree (build-tree credit-examples))
#S(decision-tree
    :test-name income
    :test #<Compiled-function income #x3525CE>
    :branches
          ((0-to-15k . #S(leaf :value high))
          (15k-to-35k . #S(decision-tree
              :test-name history
              :test
              #<Compiled-function history #x3514D6>
              :branches
               ((good . #S(leaf :value moderate))
               (bad . #S(leaf :value high))
               (unknown . #S(decision-tree
              :test-name debt
              :test
                   #<Compiled-function debt #x351A7E>
              :branches
                   ((high . #S(leaf :value high))
                    (low .
                   #S(leaf :value moderate)))))))))
```

```
            (over-35k . #S(decision-tree
               :test-name history
               :test #<Compiled-function history #x…6>
               :branches
                    ((good . #S(leaf :value low))
                    (bad . #S(leaf :value moderate))
                     (unknown .
                        #S(leaf :value low)))))))
>(classify '((history . good) (debt . low)
(collateral . none) (income . 15k-to-35k)) credit-
tree)
moderate
```

Exercises

1. Run the ID3 algorithm in another problem domain and set of examples of your choice. This will require a set of examples similar to those of Table 19.1.

2. Take the credit example in the text and randomly select two-thirds of the situations. Use these cases to create the decision tree of this chapter. Test the resulting tree using the other one-third of the test cases. Do this again, randomly selecting another two-thirds. Test again on the other one-third. Can you conclude anything from your results?

3. Consider the issues of "bagging" and "boosting" presented in Luger (2009, Section 10.3.4). Apply these techniques to the example of this chapter.

4. There are a number of other decision-tree-type learning algorithms. Get on the www and examine algorithms such QA4. Test your results from the algorithms of this chapter against the results of QA4.

5. There are a number of test bed data collections available on the www for comparing results of decision tree induction algorithms. Check out Chapter 29 and compare results for various test domains.

20 Lisp: Final Thoughts

Both Lisp and Prolog are based on formal mathematical models of computation: Prolog on logic and theorem proving, Lisp on the theory of recursive functions. This sets these languages apart from more traditional languages whose architecture is just an abstraction across the architecture of the underlying computing (von Neumann) hardware. By deriving their syntax and semantics from mathematical notations, Lisp and Prolog inherit both expressive power and clarity.

Although Prolog, the newer of the two languages, has remained close to its theoretical roots, Lisp has been extended until it is no longer a purely functional programming language. The primary culprit for this diaspora was the Lisp community itself. The pure lisp core of the language is primarily an assembly language for building more complex data structures and search algorithms. Thus it was natural that each group of researchers or developers would "assemble" the Lisp environment that best suited their needs. After several decades of this the various dialects of Lisp were basically incompatible. The 1980s saw the desire to replace these multiple dialects with a core Common Lisp, which also included an object system, CLOS. Common Lisp is the Lisp language used in Part III.

But the primary power of Lisp is the fact, as pointed out many times in Part III, that the data and commands of this language have a uniform structure. This supports the building of what we call *meta-interpreters*, or similarly, the use of *meta-linguistic abstraction*. This, simply put, is the ability of the program designer to build interpreters within Lisp (or Prolog) to interpret other suitably designed structures in the language. We saw this many time in Part III, including building a Prolog interpreter in Lisp, the design of the expert system interpreter `lisp-shell`, and the ID3 machine learning interpreter used for data mining. But Lisp is, above all, a practical programming language that has grown to support the full range of modern techniques. These techniques include functional and applicative programming, data abstraction, stream processing, delayed evaluation, and object-oriented programming.

The strength of Lisp is that it has built up a range of modern programming techniques as extensions of its core model of functional programming. This set of techniques, combined with the power of lists to create a variety of symbolic data structures, forms the basis of modern Lisp programming. Part III is intended to illustrate that style.

Partly as a result of the Lisp diaspora that produced Common Lisp, was the creation of a number of other functional programming languages. With the desire to get back to the semantic foundations on which McCarthy created Lisp (*Recursive functions of symbolic expressions and their computation by machine*, 1960),

several important functional language developments began. Among these we mention Scheme, SML, and OCaml. Scheme, a small, sometimes called *academic* Lisp, was developed by Guy Steele and Gerald Sussman in the 1970s. Scheme chose static, sometimes called *lexical*, scope over the dynamic scope of Common Lisp. For references on Scheme see Gerald Sussman and Guy Steele. *SCHEME: An Interpreter for Extended Lambda Calculus,* AI Memo 349, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, December 1975 and *The Scheme Programming Language* by R. Kent Dybvig (1996).

Standard ML (SML) is a general-purpose functional language with compile-time type checking. Type inference procedures use compile-time checking to limit run-time errors. For further information see Robin Milner, Mads, Tofte, Robert Harper, and David MacQueen. (1997). *The Definition of Standard ML (Revised)*. Objective Caml or Ocaml is an object-oriented extension to the functional language Caml. It has an interactive interpreter, a byte-code compiler, and an optimized native-code compiler. It integrates object-orientation with functional programming with SML-like type inference. The language is maintained by INRIA; for further details see *Introduction to Objective Caml* by Jason Hickey (2008) and *Practical OCaml* by Joshua Smith (2006).

In designing the algorithms of Part III, we have been influenced by Abelson and Sussman's book *The Structure and Interpretation of Computer Programs* (1985). Steele (1990) offers an essential guide to using Common Lisp. Valuable tutorials and textbooks on Lisp programming include *Lisp* (Winston and Horn 1984), *Common LispCraft* (Wilensky 1986), *Artificial Intelligence Programming*, Charniak et al. (1987), *Common Lisp Programming for Artificial Intelligence* (Hasemer and Domingue 1989), *Common Lisp: A Gentle Introduction to Symbolic Computation* (Touretzky 1990), *On Lisp: Advanced Techniques for Common Lisp* (Graham 1993), and *ANSI Common Lisp* (Graham 1995).

A number of books explore the use of Lisp in the design of AI problem solvers. *Building Problem Solvers* (Forbus and deKleer 1993) is an encyclopedic treatment of AI algorithms in Lisp and an invaluable reference for AI practitioners. Also, see any of a number of general AI texts that take a more Lisp-centered approach to the basic material, including *The Elements of Artificial Intelligence Using Common Lisp* by Steven Tanimoto (1990). Finally, we mention *Practical Common Lisp* an introductory book on Common Lisp by Peter Seibel (2004).

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.

21 Java, Representation, and Object-Oriented Programming

Chapter Objectives

The primary representational constructs for Java are introduced including:
- Objects and classes
- Polymorphism
- Encapsulation
- Inheritance constructs presented
  - Single inheritance
  - Interfaces
  - Scoping and access
- Java Standard Libraries
- The Java idiom

Chapter Contents

21.1 Introduction to O-O Representation and Design

Java has its roots in several languages and their underlying ways of thinking about computer programming. Its syntax resembles C++, its semantics reflects Objective-C, and its philosophy of development owes a fundamental debt to Smalltalk. However, over the years, Java has matured into its own language. This chapter traces the roots of Java to the ideas of object-oriented programming, a way of thinking about programming, program structure, and the process of development that it shares with these earlier languages.

The origins of object-oriented programming are within the artificial intelligence research community. The first implementation of this programming paradigm was built at Xerox's Palo Alto Research Center with the creation of Smalltalk. The first release was Smalltalk-72 (in 1972). Object-orientation was a critical component of the early AI research community's search for representational techniques that supported intelligent activity both in humans and machines. The 1960s and 1970s saw the AI community develop semantic nets, frames, flavors, as well as other techniques, all of which had a role in the eventual development of object systems (see Luger 2009, Section 7.1).

This chapter will quickly cover the fundamentals of Java, but its purpose is not to teach the basics of Java programming, nor to provide a comprehensive guide of Java features. This chapter builds a conceptual framework with which to understand and reason about Java-style problem solving, as well as to set the stage for subsequent chapters.

21.2 Object-Orientation

In previous sections of this book, we discussed functional and declarative (or logic-based) programming as implemented in Lisp and Prolog respectively. Each of these programming models has its strengths and clear advantages for certain tasks. The underlying focus of OOP and Java is the desire to model a problem as a composition of pieces that interact with one another in meaningful ways. Among the goals of this approach is helping us to think about programs in terms of the semantics of the problem domain, rather than of the underlying computer; supporting incremental prototyping by letting us build and test programs one object at a time; and the ability to create software by assembling prefabricated parts, much like manufacturers create goods from raw inputs. Indeed, OOP grew out of a need for a different software representation and development process altogether.

The key to OOP is its representational flexibility supporting the decomposition of a problem into components for which one can define behaviors and characteristics: A typical car has an engine that transfers power to the wheels, both of which are attached to a frame and enclosed in a body. Each of these components can be further broken down: an engine consists of valves, pistons, and so forth. We can proceed down to some atomic level, at which components are no longer composed of some fundamentally simpler part. This introduces the idea of abstraction, whereby we regard some composition of pieces as a single thing; for example, a car. In Java, these pieces are called objects. One of the central ideas of object-orientation is that these objects are complete definitions of their "real-world" correlates. They define both the data and behaviors needed to model those objects.

A major consequence of OOP is the goal of creating general, reusable objects that can be used to build different programs, much like how a 9-volt battery from any vendor can be plugged into thousands of entirely different devices. This requires separating the definition of what an object does from the implementation of how it does it, just as the battery's specification of its voltage frees its users from worrying about the number of cells or the chemical reactions inside it. This idea has been very important to the growth of Java.

Originally, the Java language started out small, with only a few constructs more than languages like C. However, as it has found use, Java has grown through the efforts of developers to create packages of reusable objects to manage everything from user interface development, the creation of data structures, to network communication. In fact, if we look closely at Java, it retains this flavor of a small kernal that is enhanced by numerous reusable packages the programmer can draw upon in their work. To support this

growth, Java has provided rich, standardized mechanisms for creating and packaging reusable software components that not only support hiding implementation details behind object interfaces, but also give us a rich language of types, variable scoping, and inheritance to manage these interface definitions.

21.3 Classes and Encapsulation

The first step in building reusable composite parts is the ability to describe their composition and behaviors. Java uses the notion of classes to describe objects. A class is simply a blueprint for an object. It describes an object's composition by defining state variables, and the object's behaviors by defining methods. Collectively, state variables and methods are called the *fields* of an object.

In general, a program can create multiple objects of a given class; these individual objects are also called instances of a class. Typically, the state variables of each instance have distinct values. Although all members of a class have the same structure and types of behaviors, the values stored in state variables are unique to the instance. Methods, on the other hand, are simply lists of operations to perform. All instances of a class share the same methods.

Let's say we are designing a class to model microwave ovens. Such a class would be composed of a magnetron and a timer (among other things), and would provide methods for getting the object to do something, such as cooking the food. We next show how this class may look in Java. `Magnetron` and `Timer` are classes defined elsewhere in the program, and `mag` and `t` are state variables of those types. `setTimer` and `cookMyFood` are the methods. State variables establish what is called an assembly, or "has a," relationship between classes. In this example, we would say that a `Microwave` "has a" `Magnetron` and "has a" `Timer`. This approach is analogous to the early AI notion of inheritance reflected in semantic networks (Luger 2009, Section 7.1).

```
public class Microwave {
  private Magnetron mag;
  private Timer t;
  public void setTimer (Time howLongToCook) {...}
  public Food cookMyFood (Food coldFood) {...}
}
```

Objects encapsulate their internal workings. A well-designed object does not allow other objects to access its internal structure directly. This further reinforces the separation of what an object does from how it does it. This pays benefits in the maintenance of software: a programmer can change the internals of an object, perhaps to improve performance or fix some bug, but as long as they do not change the syntax or semantics of the object interface, they should not affect its ability to fit into the larger program. Think again of the `Microwave` class (or the oven itself). You don't touch the magnetron. You don't even have to know what a magnetron is. You control the behavior that matters to you by adjusting relevant settings via

the interface, and leave the rest to the microwave alone. Java provides the access modifiers to control what it exposes to the rest of the program. These modifiers can be *private*, *public*, and *protected*, and are described in further detail in Section 21.6.

21.4 Polymorphism

Polymorphism has its roots in the Greek words "polos" meaning many, and "morphos" meaning form, and refers to a particular behavior that can be defined for different classes of objects. Whether you drive a car or a truck, you start it by inserting a key and turning it. Even if you only know how to start a car, you can easily start a truck because the interface to both is the same. The mechanical and electrical events that occur, when we start a truck's diesel engine say, are different from those of starting a car, but at a level of abstraction, the actions are the same. If we were defining cars and trucks in an object-oriented language, we would say that the start method is polymorphic across both types of vehicles.

Java supports polymorphism by allowing different objects to respond to different methods with the same name. In other words, two different classes can provide method implementations of the same name. Let's say we have two classes, one modeling microwave ovens and the other modeling traditional stoves. Both can implement their own `startCooking` method, even if they do so in fundamentally different ways, using completely different method codes to do it.

Polymorphism is one benefit of OOP's separation of an object's interface from its implementation, and it provides several benefits. It simplifies the management of different classes of related types, and lets us write code that will work with arbitrary classes at some point in the future. For example, we can write a program for controlling a car object, leaving it up to some other developer to actually provide the car object itself. Let's say your code makes calls to the car methods `turnOn` and `shiftGear`. With such a framework, a developer might decide to plug in a truck object instead of a car. If the appropriate measures are taken, it can actually work – even though we never expected our program to work with trucks.

The benefits of polymorphism are in simplifying the structure of the software by exploiting similarities between classes, and in simplifying the addition of new objects to existing code. As a more practical example, suppose we want to add a new type of text field to a user interface package. Assume this new field only allows users to enter numbers, and ignores letter keys. If we give this new text field object the same methods as the standard text field (polymorphism), we can substitute it into programs without changing the surrounding code.

This leads to a concept at the heart of both AI and Java: semantics. What is it that makes the car object and the truck object semantically similar? How do you write code that lets somebody swap a car for a truck? These questions introduce the notion of *inheritance*.

21.5 Inheritance

Cars and trucks are both travel vehicles. Traditional stoves and microwave ovens are both cooking appliances. To model these relationships in Java, we would first create *superclasses* for the more general, abstract things (travel vehicles and cooking appliances). We would then create classes for the more specific, concrete things (cars, trucks, stoves and microwave ovens) by making them *subclasses* of the appropriate superclasses. Superclasses and subclasses are also referred to as *parent* classes and *child* classes, respectively.

What does this buy us? Two things: consolidation and reuse. To write a class to model a car, and then one to model a truck, it should be obvious that there is considerable overlap between the two definitions. Rather than duplicating the definition of such common functions like starting, stopping, accelerating, and so on, in each object, we can consolidate the descriptions in the methods of the more abstract notion of a vehicle. Furthermore, we can then use the vehicle superclass to quickly create a motorcycle class: motorcycles will inherit the general functionality of a vehicle (starting, stopping, etc), but will add those properties that are unique to it, such as having two wheels.

In Java terminology, the classes for each of cars, trucks, and motorcycles are said to *extend* or *inherit* from the class for vehicle. The code skeletons for implementing this inheritance are shown next. In particular, note that there is nothing in the `Vehicle` class that specifies that it is to be used as a superclass. This is important, as another developer might at some point want to extend our class, even if the desire to do so never entered our mind when we originally designed the software. In Java, most classes can later be extended to create new subclasses.

```
public class Vehicle
{
    public void start()     /* code defines start*/
    public void stop()      /* code defines stop */
    public void drive()    /* code defines drive */
    public boolean equals (Vehicle a)
         /* code defines if two vehicles are same */
    public int getWheels() { return 4;}
}
public class Motorcycle extends Vehicle
{
    public int getWheels() { return 2;}
                      /* Overrides Vehicle method */
}
public class Car extends Vehicle
{
    /* define methods that are unique to Car here */
}
```

```
public class Truck extends Vehicle
{
            /*define methods unique to Truck here*/
}
```

To develop this idea further, imagine that we wanted a class to model dump trucks. Again, we have overlap with existing components. Only this time, there may be more overlap with the class for trucks than the class for travel vehicle. It might make more sense to have dump trucks inherit from trucks. Thus, a dump truck is a truck, which is a travel vehicle. Figure 21.1 illustrates this entire class hierarchy using an inheritance diagram, an indispensable design aid.

We next discuss *inheritance-based polymorphism*. When we extend a superclass, our new subclass inherits all public fields of its parent. Any code that uses an object of our subclass can call the public methods or access the public state variables it inherited from its parent class. In other words, any program written for the superclass can treat our subclass exactly as if it were an instance of the superclass. This capability enables polymorphism (Section 21.3), and is crucial for code reuse; it allows us to write programs that will work with classes that have not even been written yet.

The ability to extend most classes with subclasses establishes a semantic relationship: we know that dump trucks are trucks, and that trucks are travel vehicles, therefore trucks are travel vehicles. Furthermore, we know how to start travel vehicles, so we clearly know how to start dump trucks. In Java, if the class `Vehicle` has a method `start`, then so does the class `DumpTruck`. We next present fragments of code that illustrate how polymorphism is enabled by inheritance.

Since DumpTruck and Car are descendants of Vehicle, variables of type Vehicle can be assigned objects of either type:

```
Vehicle trashTruck = new DumpTruck();
Vehicle dreamCar = new Car();
```

Methods that take arguments of type Vehicle can take instances of any descendants:

```
if (dreamCar.equals(trashTruck))
then
    . . .
```

Finally, methods that return type `travelvehicle` can return instances of any subclass. The following example illustrates this, and also shows an example of the Factory design pattern. A Factory pattern defines a class whose function is to return instances of another class, often testing to determine variations on those instances.

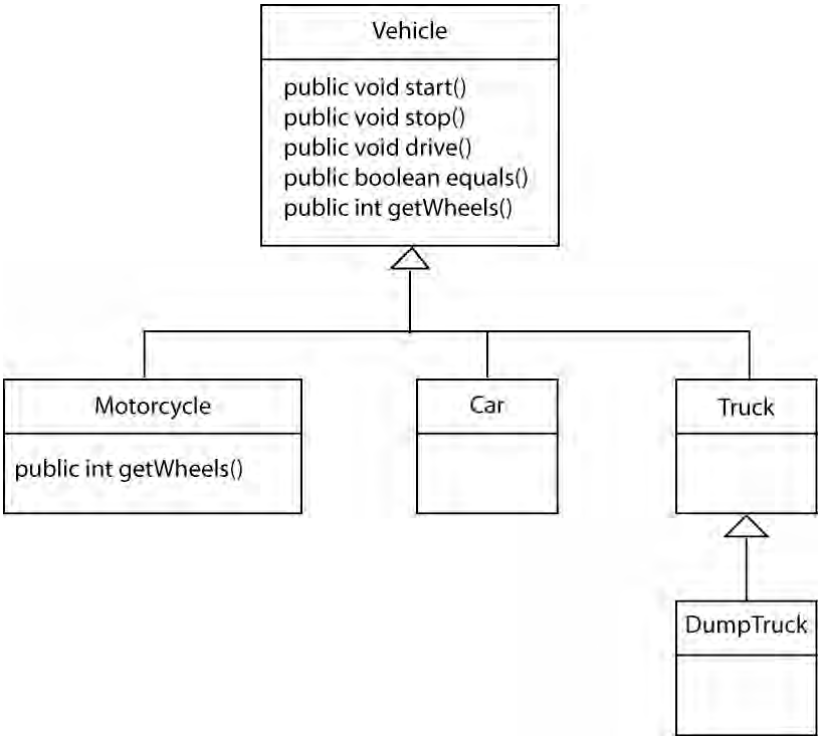

Figure 21.1: An inheritance diagram of our travelvehicle class hierarchy. Arrows denote "extends," also known as the "is a" relationship. For example, a dump truck is a truck, which is a travelvehicle.

```
class Vehicle Factory()
{
    public Vehicle getCar(Person customer)
    {
        if(customer.job = "Construction")
        then return new Truck();
        if (customer.job = "Salesman"_
        then return new Car();

        // etc.
}
```

We have seen semantic relationships and type hierarchies before. In Section 2.4 we used them to model symbolic knowledge in the form of semantic networks and frames. Semantic relationships serve the same purpose in Java. We are using classes to represent things in a hierarchically organized way. Java can credit classical AI research as the source of modeling knowledge using semantic hierarchies on a computer (see Luger 2009, Section 7.1).

Another important aspect of inheritance and polymorphism is the ability to give different classes the same methods, but to give them different definitions for each class. A common application of this is the ability to change a method definition in a descendant. For example, suppose we

want to change the start method for a motorcycle from using a starter to a simple kick-start. We would still make motorcycle a descendant of Automobile, but would redefine the start method:

```
class Motorcycle extends Vehicle
{
    public void start()
    {
        // Motorcycles use this definition of start
        // instead of Vehicle's
    }
}
```

We make one final point on inheritance. In our class hierarchy, `DumpTruck` has exactly one parent: `Truck`. That parent has only one parent as well: `Vehicle`. Every class in Java has exactly one parent. What is the parent of `Vehicle`? When a class does not explicitly state which class it extends, it automatically inherits from the generic class called `Object`.

`Object` is the root of all class hierarchies in Java. This automatic inheritance means that every single object in a Java program is, by inheritance, an instance of the `Object` class. This superclass provides certain facilities to every single object in a Java program, many of which are rarely used directly by the programmer, but which can, in turn, be used by other classes in the Java Standard Library.

In some languages (e.g., C++ and CLOS), classes can inherit from multiple parents. This capability is known as *multiple inheritance*. Java does not support multiple inheritance, opting instead for single inheritance. There is an ongoing debate between proponents of each approach, primarily related to program integrity and inadvertent creation of side effects, but we will not go into theses issues here.

21.6 Interfaces

Up to this point, the word *interface* has been used to refer to the public fields of a class. Here we introduce a special Java construct that also happens to be known as an "interface". Although these concepts are related, it is important to distinguish between the two.

A Java interface is a mechanism for standardizing the method names of classes for some purpose. It is effectively a contract that a class can agree to. The term of this agreement is that a class will implement at least those methods named in the interface definition.

A Java interface definition consists of the interface name, and a list of method names. A class that agrees to implement the methods in an interface is said to implement that interface. A class specifies the interfaces it implements in the class declaration using an implements clause. If a class agrees to implement an interface, but lacks even one of the prescribed methods, it won't even compile. For example, consider the interface:

```
interface Transportation
{
    public int  getPassengerCapacity();
    public double getMaximumSpeed();
    public double getCost();
}
```

The definition of the `Transportation` interface does not define the implementation of the methods `getPassengerCapacity()`, `getMaximumSpeed()`, or `getCost()`. It only states that any instance of `Transportation` must define these methods. If we had changed the definition of the class `Automobile` to read as follows, then we would be required to define the methods of `Transportation` in the class `Vehicle`.

```
class Vehicle implements Transportation
{
    . . .
}
```

This feels a bit like inheritance. After all, when we extend a parent class, we are promising that all public fields from that parent class are available, allowing programs to treat our new class like the parent class. Even if the inherited fields are overridden, they are sure to exist. However, there are conceptual and practical differences between inheritance and Java interfaces.

What is most important is that interfaces are only a specification of the methods in a class. They carry no implementation information. This allows us to define "contracts" between classes for how classes should behave without concern for their implementation. Suppose, for example, we wanted to define a class to manage all our vehicles. This class, which we call `Fleet`, could be defined to manage instances of the interface `Transportation`. Any class that implemented that interface could be in `Fleet`:

```
class Fleet
{
    public void add(Transportation v)
    {
        //Define add here
    }
    public double getTotalCost()
    {
        //sum of getCost on all Transportation
    }
}
```

This technique will be extremely important in using Java for building AI applications, since it allows us to create general tools for searching problem

spaces, pattern matching, and so on without knowing what types of spaces we will be searching or objects we will be matching. We simply create the interface such objects must define, and write our search engines to handle those interfaces.

At the operational level, classes can implement any number of interfaces. The methods defined in the interfaces can even overlap. Because the implementing class does not actually inherit method implementations from the interface, overlap is irrelevant. This gives us many of the benefits of multiple inheritance, and yet retains some security from unwanted side effects. Finally, interfaces provide another mechanism to enable polymorphism. As long as we know that a specific object can perform certain operations, we can ask it to do so.

Interfaces are especially useful when writing frameworks that require some specific behavior from objects of some class, but that do not care at all about the object's family. Thus, the framework does not constrain the semantic relationships of the class, which may come from a completely different developer, and might not have even been written yet. Java frameworks that impose such constraints often over-step their boundary, particularly when an interface would have been sufficient.

21.7 Scoping and Access

Java allows programmers to specify the context in which declarations and variables have an effect. This context is known as the *scope*. For example, if we declare a variable inside of a method, that variable is only accessible within that method.

Fields declared *private* are only accessible by methods in the same class; not even children of the class can access these fields. Although we often would like child classes to be able to access these variables, it is usually good style to define all member variables as `private`, and to provide public accessor functions to them. For example:

```
public class Foo
{
    private int value;
    public int getValue()
    {
        return value;
    }
    public void setValue(int newValue)
    {
        value = newValue;
    }
}
```

The reason for enforcing the private scope of member variables so strongly is to maintain the separation of interface and implementation. In the previous example, the class `Foo` does not expose the representation of the

variable `value`. It only makes a commitment to provide such a value through accessor functions `getValue()` and `setValue(int newValue)`. This allows programmers to change the internal representation of value in subtle ways without affecting child classes. For example, we may, at some time, want to have the class `Foo` notify other classes when `value` changes using Java's event model. If `value` were public, we could not be sure some external method did not simply assign directly to it without such a notification. Using this private variable/public accessor pattern makes this easy to guarantee.

In contrast, protected fields are accessible by inherited methods, and public fields are accessible by anybody. As a rule, protected scope should be used carefully, as it violates the separation of interface and implementation in the parent class.

Fields declared *static* are unique to the class, not to the instance. This means that there is only a single value of a static variable, and all instances of a given class will access that value. Because this creates the potential for interesting side-effects, it should be used judiciously. Static methods do not require an instance in order to be called. As such, static methods cannot access non-static state variables, and have no this variable.

Another useful modifier is *final*. You cannot create subclasses of a class that is declared `final`, or override a final method in a subclass. This allows the programmer to limit modifications to methods of a parent class. It also allows the creation of constants in classes, since variables declared `final` cannot be changed. Generally, `final` variables are declared `static` as well:

```
public class Foo
{
    public static final double pi = 3.1416;
    . . .
}
```

21.8 The Java Standard Library

In pursuing the goal of widespread reuse of software components, Java comes prepackaged with a substantial collection of classes and interfaces known as the *Java Standard Library*. This library provides a variety of facilities, including priority queues, file I/O, regular expressions, interprocess communication, multithreading, and graphical user interface components, to name just a few.

The Java standard library has proven to one of Java's most strategic assets. At a minimum, developers can focus on high-level objects by reusing common components, instead of building every application from a few basic commands. Using the techniques described above, standardization saves developers from having to learn multiple libraries that provide identical functionality, which is a common problem with other languages for which multiple software vendors distribute their own libraries. The Java library continues to grow driven by emerging challenges and technologies.

21.9 Conclusions: Designing in Java

As we strive to acquire the Java idiom, and not simply "learn the language", we must begin to think in Java. As stated earlier, Java and OOP emerged out of a need to improve the entire process of creating software. More than other languages, Java is linked to a particular set of assumptions about how it will be used. For example, *reuse* is an important aspect of this approach. When possible, we should not build Java programs from scratch using the basic language, but should look for reusable classes in the standard libraries, and may even want to consider commercial vendors or open source providers for complex functionality. In addition, Java programmers should always try to generalize their own code to create classes for possible later reuse.

One intention of object-oriented programming is to simplify later program maintenance by organizing code around structures that reflect the structure of the problem domain. This not only makes the code easier to understand, especially for other programmers, but also tends to reduce the impact of future changes. In maintenance, it is important for the change process to be reasonable to the customer: change requests that seem small to the customer should not take a long time. Because the final user's estimate of the complexity of a requested change reflects her or his own conceptual model of the problem, capturing this model in the computer code helps to insure a reasonable change process.

Finally, the ability to create classes that hide their internal structure makes Java an ideal language for prototyping (Luger 2009, Section 8.1). We can build a program up one class at a time, debugging and testing classes as we go. This sort of iterative approach is ideal for the kinds of complex projects common to modern AI, and programmers should design classes to support such an iterative approach. Guidelines for this approach to programming include:

> Try to make the breakdown of the program into classes reflect the logical structure of the problem domain. This not only involves defining classes to reflect objects in the domain, but also anticipating the source of pressures to change the program, and making an effort to isolate anticipated changes.
>
> Always make variables `private`, and expose `public` methods only as needed.
>
> Keep methods small and specialized. Java methods almost never take more than a full screen of code, with the vast majority consisting of less than a dozen lines.
>
> Use inheritance and interface definitions to "program by contract". This idea extends such useful ideas as strong variable typing to allow the programmer to tightly control the information passing between objects, further reducing the possibility of bugs in the code.

Finally, we should remember that this is not simply a book on programming in Java, but one on AI programming in Java. A primary goal

of Artificial Intelligence has always been to find general patterns of complex problem solving, and to express them in a formal language. Although Java does not offer the clear AI structures found in Lisp and Prolog, it does offer the programmer a powerful tool for creating them in the form of reusable objects/classes.

In writing this chapter, we faced a very difficult challenge: how can we write a meaningful introduction to so complex a language as Java, without falling into the trap of trying to write a full introductory text in the language. Our criteria for organizing this chapter was to emphasize those features of Java that best support the kinds of abstraction and structuring that support the complexity of AI programming. As with Lisp and Prolog, we will not approach Java simply as a programming language, but as a language for capturing the patterns of AI programming in general, reusable ways.

We should not simply think of Java as a programming language, but as a language for creating specialized languages: powerful classes and tools we can use to solve our programs. In a very real sense, master Java programmers do not program in Java, as much as they use it to define specialized languages and packages tailored to solve the various problems they face. This technique is called *meta-linguistic abstraction*, and it is at the heart of OOP. Essentially meta-linguistic abstraction means using a base language to define a specialized language that is better suited to a particular problem.

Meta-linguistic abstraction is a powerful programming tool that underlies many of the examples that follow in later chapters, especially search engines, the design of expert systems shells, and control structures for genetic algorithms and machine learning. We can think of these generalized search strategies, representations, or expert-system shells as specialized languages built on the Java foundation. All the features of Java described in this chapter, including inheritance, encapsulation, polymorphism, scoping, and interfaces serve this goal.

Exercises

1. Create an inheritance hierarchy for a heating and air conditioning system for a building using centralized heating/cooling sources. Attempt to make the overall control parallel and asynchronous. Capture your design with an inheritance diagram (see Figure 25.3).

2. Expand on problem 1, where each floor of the building has a number of rooms that need to be heated/cooled. Create a room hierarchy where the generic room has a fixed set of attributes and then each individual room (office, common room store room) inherits different properties. Each room should have its individual controller to communicate with the general-purpose heating/cooling system.

3. Problems 1 and 2 can be even further expanded, where each room has its own volume, insulation factor (the area of windows, say), and heating/cooling requirements. Suppose you want to design the ultimate "green" or conservation-oriented structure. How might you design such a building?

4. Create an inheritance hierarchy for an elevator control system. Take, for example a fifteen-storey building and three elevators. Capture your design with an inheritance diagram (see Figure 25.3).

5. Create Java structure code, similar to that in Sections 21.4.2, 21.5 and 21.6 for implementing the examples you created in questions you considered from exercises 1 - 4.

6. Consult the Java Library. What software components therein might support your program designs for question 3?

22 Problem Spaces and Search

Chapter Objectives

Uninformed state space search strategies are revisited:
- Depth-first search
- Breadth-first search

Heuristic or best-first search is presented

Java idioms are created for implementing state-space search
- Establishing a framework
  - Interface class
    - `Solver`
    - `AbstractSolver`
  - Implements search algorithms

Chapter Contents

22.1 Abstraction and Generality in Java

This book, like the history of programming languages in general, is a story of increasingly powerful abstraction mechanisms. Because of the complexity of the problems it addresses and the rich body of theory that defines those problems in computational terms, AI has proven an ideal vehicle for developing languages and the abstraction mechanisms that give them much of their value. Lisp advanced ideas of procedural abstraction by following a mathematical model – recursive functions – rather than the explicitly machine influenced structures of earlier languages. The result was a sophisticated approach to variable binding, data structures, function definition and other ideas that made Lisp a powerful test bed for ideas about symbolic computing and programming language design. In particular, Lisp contributed to the development of object-oriented programming through a variety of Lisp-based object languages culminating in the Common Lisp Object System (CLOS). Prolog took a more radical approach, combining unification based pattern matching, automated theorem proving, and built-in search to render procedural mechanisms almost completely implicit and allow programmers to approach programs in a declarative, constraint-based manner.

This chapter continues exploring abstraction methods, taking as its focus the object-oriented idiom, and using the Java programming language as a vehicle. It builds on Java's object-orientated semantics, employing such mechanisms as inheritance and encapsulation. It also explores the interaction between AI theory and program architecture: the problems of

rendering theories of search and representation into code in ways that honor the needs of both. In doing so, it provides a foundation for the search engines used in expert systems, cognitive models, automated reasoning tools, and similar approaches that we present later. As we develop these ideas, we urge the reader to consider the ways in which AI theories of search and representation shaped programming techniques, and the abstractions that project that theory into working computer programs.

One of the themes we have developed throughout this book concerns generality and reuse. Through the development of object-oriented programming, many of these ideas have evolved into the notion of a framework: a collection of code that furnishes general, reusable components (typically data structures, algorithms, software tools, and abstractions) for building a specific type of software. Just as a set of modular building components simplifies house construction, frameworks simplify the computational implementation of applications.

Creating useful frameworks in Java builds on several important abstraction mechanisms and design patterns. Class inheritance is the most basic of these, allowing us to specify frameworks as general classes that are specified to an application. However, class inheritance is only the starting point for the subtler problems of implementing search algorithms. Additional forms of abstractions we use include interfaces and generic collections.

Our approach follows the development of search algorithms leading to the Lisp and Prolog search shells in Chapters 4 and 14 respectively, but translates it into the unique Java idiom. We encourage the reader to reflect on the differences between these approaches and their implications for programming. Before implementing a framework for basic search, we present a brief review of the theory of search.

22.2 Search Algorithms

Search is ubiquitous in computer science, particularly in Artificial Intelligence where it is the foundation of both theoretical models of problem solving, practical applications, and general tools and languages (including Prolog). This chapter uses the theory of search and Java idioms to develop a framework for implementing search strategies. We have explored the theory of Artificial Intelligence search elsewhere (Luger 2009, Chapters 3, 4, and 6), but will review the key theoretical ideas briefly.

Both the analysis of problem structure and the implementation of problem solving algorithms depend upon modeling the structure of a problem graphically: as a state-space. The elements defining a state-space are:

> **A formal representation of possible states of a problem solution**. We can think of these as all possible steps in a solution process, including both complete solutions and partial steps toward them. An example of a state might be a configuration of a puzzle like the Rubik's cube, an arrangement of tiles in the 16-puzzle, or a complex set of logical assertions in an expert system.

Operators for generating new states from a given state. In our puzzle example, these operators are legal moves of the puzzle. In more sophisticated applications, they can be logical inferences, steps in data interpretation, or heuristic rules for expert reasoning. These operators not only generate the next states in a problem solving process, but also define the arcs or links in a state-space graph.

Some way of recognizing a goal state.

A starting state of the problem, represented as the root of the graph.

Figure 22.1 shows a portion of the state-space for the 8-puzzle, an example we will develop later in this chapter.

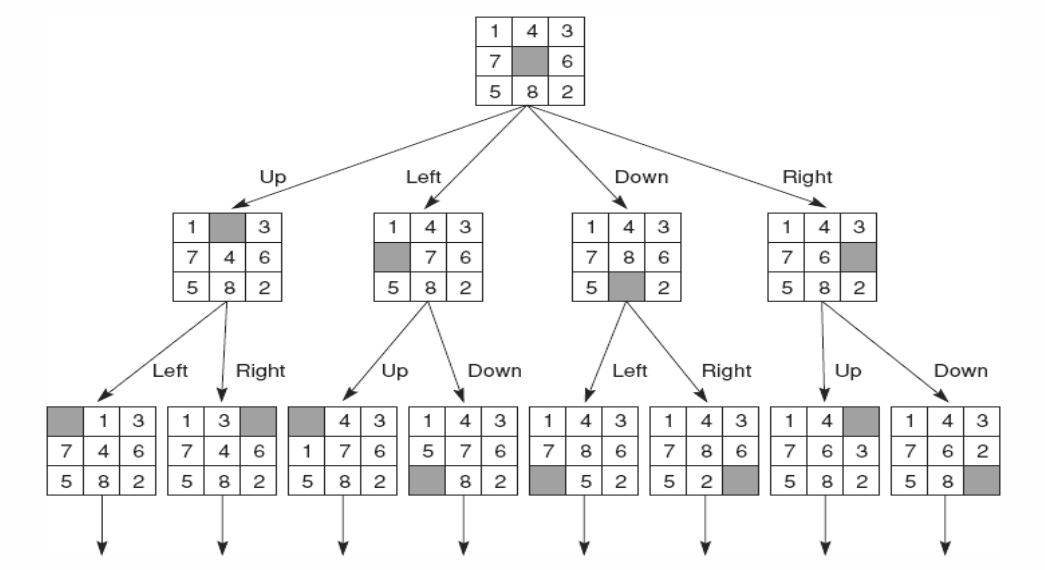

Figure 22.1. A sample state space to be searched. The goal is to have the numbers in clockwise order from the upper left hand corner. States are generated by "moving" the blank.

Our discussion begins with two basic "uninformed" algorithms: depth-first search (DFS) and breadth-first search (BFS). We call these "uninformed" because they do not use knowledge of the problem-space to guide search, but proceed in a fixed order. DFS picks a branch of the search space and follows it until it finds a goal state; if the branch ends without finding a goal, DFS "backs up" and tries another branch. In contrast, breadth-first search goes through the state space one layer at a time.

Figure 22.2 illustrates the difference between these approaches for the 8-puzzle problem, where depth-first search is given a five level depth bound. Note that although both searches find solutions, breadth-first search finds the solution that is closest to the root or start state. This is always true because BFS does not consider a solution path of distance `d` from the start until it has considered all solution paths of distance `d-1` from the start.

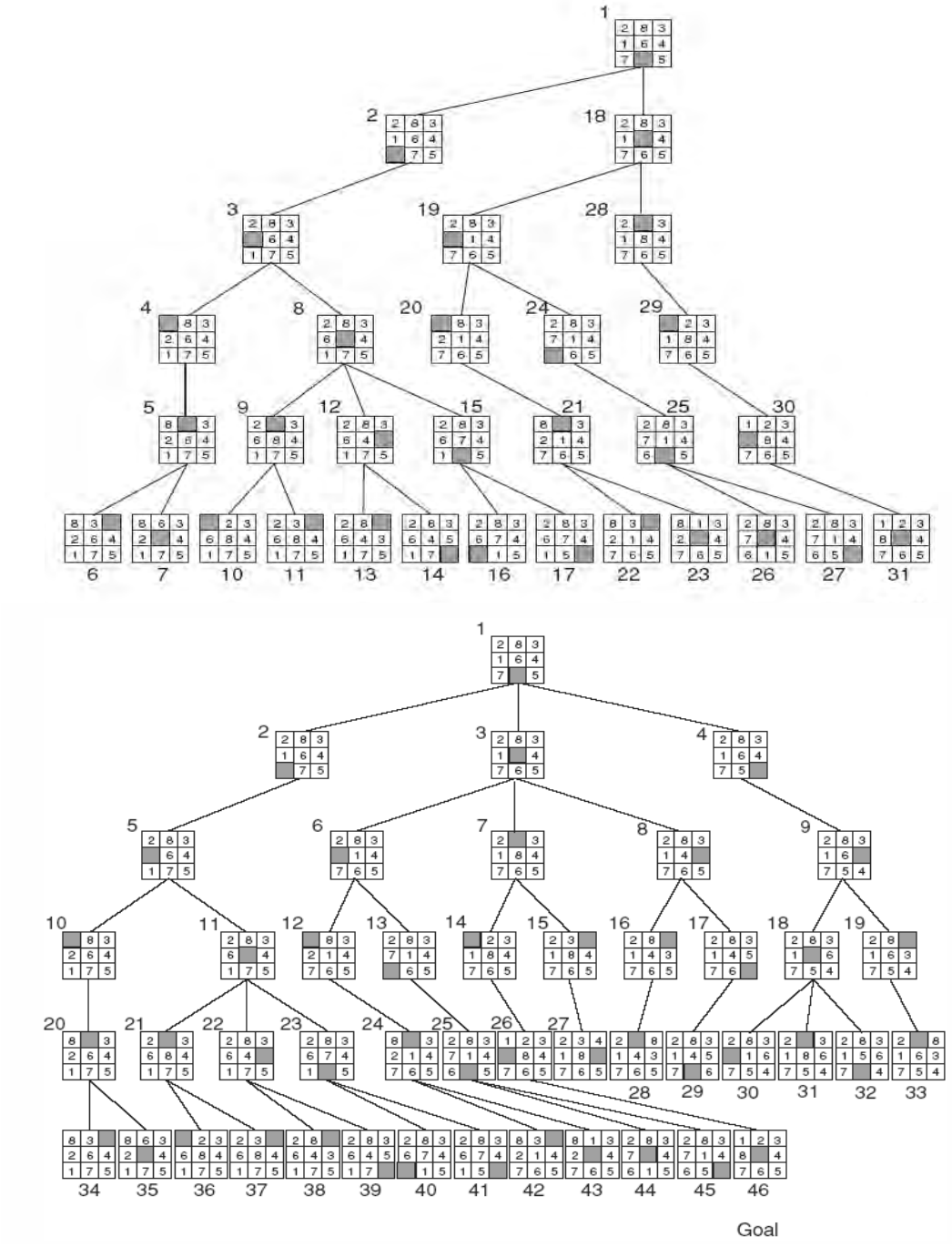

Figure 22.2. Depth-first search (a), using a depth bound of five levels, and breadth-first search (b) of the 8-puzzle. States are generated by "moving" the blank space.

These algorithms search a state-space in very different orders, but share a common general structure: both go through an iterative process of selecting a state in the graph, beginning with the start-state, quitting with success if the state is a goal, and generating its child states if it is not. What distinguishes them is how they handle the collection of child states. The following pseudo code gives a basic search implementation:

```
Search(start-state)
{
    place start-state on state-list;
    while (state-list is not empty)
    {
         state = next state on state-list;
         if(state is goal)
              finish with success;
         else
              generate all children of state and
              place those not previously visited
              on state-list;
    }
}
```

This algorithm is the general structure for all the implementations of search in this chapter: what distinguishes different search algorithms is how they manage unvisited states. DFS manages the `state-list` as a *stack*, while BFS uses a *queue*. The theoretical intuition here is that the last-in/first-out nature of a stack gives DFS its aggressive qualities of pursuing the current path. The first-in/first-out state management of a queue assures the algorithm will not move to a deeper level until all prior states have been visited. (It is worth noting that the Prolog interpreter maintains execution states on a stack, giving the language its depth-first character).

However, BFS and DFS proceed blindly through the space. The order of the states they visit depends only on the structure of the state space, and as a result DFS and/or BFS may not find a goal state efficiently (or at all). Most interesting problems involve multiple transitions from each problem state. Iterating on these multiple child states through the entire problem space often leads to an exponential growth in the number of states involved in the `state-list`. For small enough spaces, we can find a solution through uninformed search methods. Larger spaces, however, must be searched intelligently, through the use of heuristics. *Best-first search* is our third algorithm; it uses heuristics to evaluate the set of states on `state-list` in order to choose which of several possible branches in a state space to pursue next. Although there are general heuristics, such as means-ends analysis and inheritance (Luger 2009), the most powerful techniques exploit knowledge unique to the particular problem domain.

A general implementation of best-first or heuristic search computes a numerical estimate of each state's "distance" from a solution. Best-first search always tries the state that has the closest estimate of its distance to a goal. We can implement best-first search using the same general algorithm as the other state space searches presented above, however we must assign a heuristic rank to each state on the `state-list`, and maintain it as a sorted list, guaranteeing that the best ranked states will be evaluated first. `state-list` is thus handled as a priority queue.

This summary of search is not intended as a tutorial, but rather as a brief refresher of the background this chapter assumes. We refer readers who need further information on this theory to an AI text, such as Luger (2009, see Chapters 3 and 4).

21.3 Abstracting Problem States

We begin with the search pseudo-code defined above, focusing on the problem of representing states of the search space, as is typical of AI programming. This also reflects architectures we have used throughout the book: the separation of representation and control. Because of the relative simplicity of the search algorithms we implement, we approach this as separate implementations of states and search engines.

Our goal is to define an abstract representation of problem states that supports the general search algorithm and can be easily specialized through the mechanism of class inheritance. Our basic approach will be to define a class, called `State`, that specifies the methods common to all problem states and to define subclasses that add problem-specific information to it, as we see in Figure 22.3.

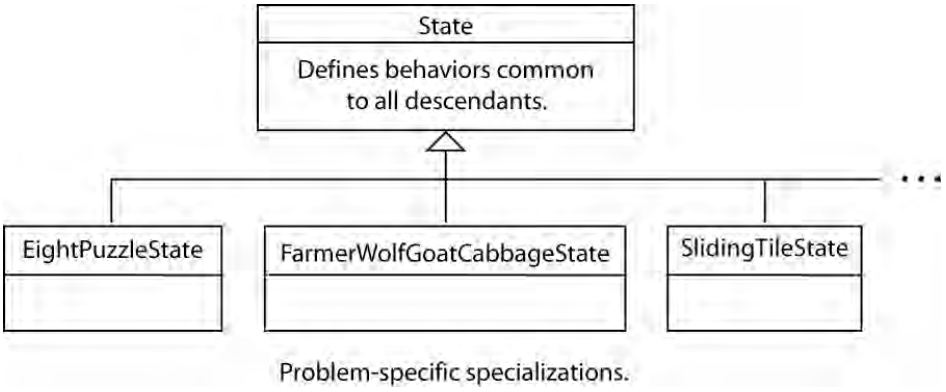

Figure 22.3. State representation for search containing problem-specific specifications.

In many cases, general class definitions like `State` will implement methods to be used (unless overridden) by descendant classes. However, this is only one aspect of inheritance; we may also define method names and the types of their arguments (called the *method signature*), without actually implementing them. This mechanism, implemented through abstract classes and methods, allows us to specify a sort of contract with future extensions to the class: we can define other methods in the framework to call these methods in instances of `State`'s descendants without knowing how they will implement them. For example, we know that all states must be able to produce next moves, determine if they are a solution, or calculate their heuristic value, even though the implementation of these is particular to specific problem states.

We can specify this as an abstract class:

```
public abstract class State
{
    public abstract Set<State> getPossibleMoves();
    public abstract boolean isSolution();
    public abstract double getHeuristic();
}
```

Note that both the methods and the class itself must be labeled `abstract`. The compiler will treat efforts to instantiate this class as errors. Abstract methods are an important concept in object-oriented programming, since the signature provides a full specification of the method's interface, allowing us to define code that uses these methods prior to their actual implementation.

Requiring the state class to inherit from the `State` base class raises a number of design issues. Java only allows inheritance from a single parent class, and the programmer may want to use the functionality of another parent class in creating the definition of the state. In addition, it seems wasteful to use a class definition if all we are defining are method signatures; generally, object-oriented programming regards classes as containing at least some state variables or implemented methods.

Java provides a mechanism, the *interface*, which addresses this problem. If all we need is a specification – essentially, a contract – for how to interact with instances of a class, we leave the implementation to future developers, we can define it as an `interface`. In addition to having a single parent class, a class can be declared to implement multiple `interface`s.

In this case, we will define `State` as an interface. An interface is like a class definition in which all methods are abstract: it carries no implementation, but is purely a contract for how the class must be implemented. In addition to the above methods, our `interface` will also define methods for getting the parent of the state, and its distance from the goal:

```
public interface State
{
    public Iterable<State> getPossibleMoves();
    public boolean isSolution();
    public double getHeuristic();
    public double getDistance();
    public State getParent();
}
```

Note in this situation the expression `Iterable<State>` returned by `getPossibleMoves()`. The expression, `Iterable<State>` is part of Java's *generics* capability, which was introduced to the language in Java 1.5. Generics use the notation `Collection-Type<Element-Type>` to specify a collection of elements of a specific type, in this case, an `Iterable` collection of objects of type `State`. In earlier versions of

Java, collections could contain any descendants of `Object`, the root of Java's class hierarchy. This prevented adequate type checking on collection elements, creating the potential for run-time type errors. Generics prevent this, allowing us to specify the type of collection elements.

Like `State`, `Iterable<State>` is an interface, rather than a class definition. `Iterable` defines an interface for a variety of classes that allow us to iterate over their members. This includes the `Set`, `List`, `Stack`, `PriorityQueue` and other collection classes. We could define this collection of child states using a specific data structure, such as a list, stack, etc. However, it is generally a bad idea to constrain later specialization of framework classes unnecessarily. Suppose the developer has good reason to collect child states in a different data structure? Once again, by using an interface to define a data type, we create a contract that will allow our search framework to implement functions that use classes, while leaving their instantiation to future programmers.

This interface is adequate to implement the search algorithms of Section 22.1, but before implementing the rest of our framework, note that two of the methods specified in the `State` interface are general enough to be defined for most, if not all, problems: `getDistance()` computes the distance from the start state, and `getParent()` returns the `parent` state in the search space. To simplify the work for future programmers, we implement a class, `AbstractState`, that provides a default implementation of these methods.

```
public abstract class AbstractState implements
         State
{
    private State parent = null;
    private double distance = 0;

    public AbstractState() {}
    public AbstractState(State parent)
    {
         this.parent = parent;
         this.distance = parent.getDistance() + 1;
    }
    public State getParent()
    {
         return parent;
    }
    public double getDistance()
    {
         return distance;
    }
}
```

Note that `AbstractState` implements the `State` interface, so classes that extend it can be used in our framework, freeing the programmer for the need to implement certain methods. It may seem that we have gone in a circle in this discussion, first defining `State` as an abstract class, then arguing that it should be an `Interface`, and now reintroducing an abstract class again. However, there are good reasons for this approach. Figure 22.4 illustrates a common Java design pattern, the use of an interface to specify an implementation "contract", with an abstract class providing default implementations of general methods.

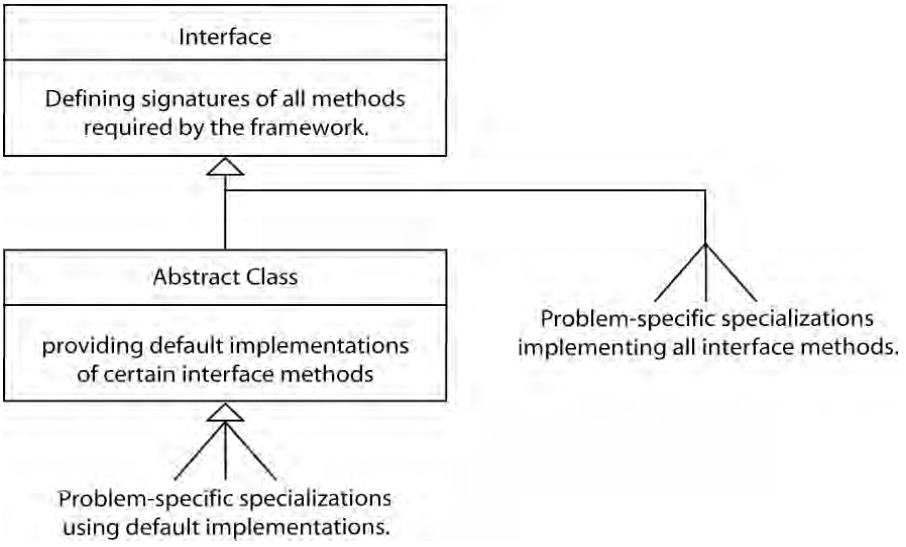

Figure 22.4. A Java design pattern: using an interface to specify a contract.

The pattern of specifying method signatures in an interface and providing default implementations of certain methods in an abstract class, is common in Java. By defining all methods required of the framework in an interface and using the interface to specify all types, we do not constrain future programmers in any way. They can bypass the abstract class entirely to address efficiency concerns, the needs of a problem that may not fit the default implementations, or simply to improve on the defaults. In many situations, however, programmers will use the abstract class implementation to simplify coding.

The next section repeats this pattern in implementing the control portion of our framework: the depth-first, breadth-first, and best-first search algorithms described earlier.

22.4 Traversing the Problem Space

Although simple, the `State` interface fully specifies the "contract" between the search framework and developers of problem-specific state representations. It gives the method signatures for testing if a state is a goal, and for generating child states. It leaves the specific representation to descendant classes. The next task is to address the implementation of search itself: defining the list of states and the mechanisms for moving

through them in search. As with `State`, we will begin with an `interface` definition:

```
public interface Solver
{
    public List<State> solve(State initialState);
}
```

Although simple, this captures a number of constraints on solvers. In addition to requiring an initial state as input, the solve method returns the list of visited states as a result. Once again, it defines the returned collection using a generic interface. A `List<E>` is a collection of ordered elements of type `E`. As with `Set<E>`, the list interface is supported by a variety of implementations.

Using the pattern of Figure 22.4, we will provide an abstract implementation of `Solver`. The code fragment below implements a general search algorithm that does not specify the management of open states:

```
private Set<State> closed = new HashSet<State>();
public List<State> solve(State initialState)
{                              //Reset closed and open lists
    closed.clear();
    clearOpen();
    addState(initialState);
    while (hasElements())
    {
         State s = nextState();
         if (s.isSolution())
              return findPath(s);
         closed.add(s);
         Iterable<State> moves =
              s.getPossibleMoves();
         for (State move : moves)
              if (!closed.contains(move))
                   addState(move);
    }
    return null;
}
```

In this method implementation, we maintain a closed list of visited states to detect loops in the search. We maintain `closed` as a `Set<State>` and implement it as a `HashSet<State>` for efficiency. We use the `Set<State>` `interface` since the closed list will contain no duplicates.

The `solve` method begins by clearing any states from the `closed` list, and adding the initial state to the `open` list using the `addState` method. We specify `addState` as an abstract method, along with the methods

`hasElements()` and `nextState()`. These methods allow us to add and remove states from the `open` list, and test if the list is empty. We specify them as abstract methods to hide the implementation of the `open` list, allowing the particular implementation to be defined by descendents of `AbstractSolver`.

The body of the method is a loop that:

Tests for remaining elements in the `open` list, using the abstract method `hasElements()`;

Acquires the next state from the list using the abstract method `nextState()`;

Tests to see if it is a solution and returns the list of visited states using the method `findPath` (to be defined);

Adds the state to the `closed` list; and

Generates child states, placing them on the `open` list using the abstract `addState()` method.

Perhaps the most significant departure from the Lisp and Prolog versions of the algorithm is the use of an iterative loop, rather than recursion to implement search. This is mainly a question of style. Like all modern languages, Java supports recursion, and it is safe to assume that the compiler will optimize tail recursion so it is as efficient as a loop. However, Java programs tend to favor iteration and we follow this style.

We have now implemented the general search algorithm. We complete the definition of the `AbstractSolver` class by defining the `findPath` method and specifying the abstract methods that its descendents must implement:

```
public abstract class AbstractSolver implements
          Solver
{
    private Set<State> closed =
          new HashSet<State>();

    public List<State> solve(State initialState)
    {                                  // As defined above
    }

    public int getVisitedStateCount()
    {
         return closed.size();
    }

    private List<State> findPath(State solution)
    {
         LinkedList<State> path =
                new LinkedList<State>();
         while (solution != null) {
              path.addFirst(solution);
              solution = solution.getParent();
```

```
            }
            return path;
        }
        protected abstract boolean hasElements();
        protected abstract State nextState();
        protected abstract void addState(State s);
        protected abstract void clearOpen();
    }
```

Note once again how the abstract methods hide the implementation of the maintenance of `open` states. We can then implement the different search algorithms by creating subclasses of `AbstractSolver` and implementing these methods. Depth-first search implements them using a `stack` structure:

```
public class DepthFirstSolver extends AbstractSolver
{
    private Stack<State> stack = new Stack<State>();
    protected void addState(State s)
    {
        if (!stack.contains(s))
            stack.push(s);
    }
    protected boolean hasElements()
    {
        return !stack.empty();
    }
    protected State nextState()
    {
        return stack.pop();
    }
    protected void clearOpen()
    {
        stack.clear();
    }
}
```

Similarly, we can implement breadth-first search as a subclass of `AbstractSolver` that uses a `LinkedList` implementation:

```
public class BreadthFirstSolver extends
          AbstractSolver
{
    private Queue<State> queue =
          new LinkedList<State>();
    protected void addState(State s)
    {
```

```
            if (!queue.contains(s))
                queue.offer(s);
        }
        protected boolean hasElements()
        {
            return !queue.isEmpty();
        }
        protected State nextState()
        {
            return queue.remove();
        }
        protected void clearOpen()
        {
            queue.clear();
        }
    }
```

Finally, we can implement the best-first search algorithm by extending `AbstractSolver` and using a priority queue, PriorityQueue, to implement the open list:

```
    public class BestFirstSolver extends AbstractSolver
    {
        private PriorityQueue<State> queue = null;
        public BestFirstSolver()
        {
            queue = new PriorityQueue<State>(1,
                      new Comparator<State>()
                    {
                    public int compare(State s1, State s2)
                    {
                    //f(x) = distance + heuristic
                    return Double.compare(
                    s1.getDistance() + s1.getHeuristic(),
                    s2.getDistance() + s2.getHeuristic());
                    }
                    });
        }
        protected void addState(State s)
        {
            if (!queue.contains(s))
                queue.offer((State)s);
        }
```

```
        protected boolean hasElements()
        {
             return !queue.isEmpty();
        }
        protected State nextState()
        {
            return queue.remove();
        }
        protected void clearOpen()
        {
            queue.clear();
        }
    }
```

In defining the `open` list as a `PriorityQueue`, this algorithm passes in a comparator for states that uses the heuristic evaluators defined in the `State` interface.

Note that both `BreadthFirstSolver` and `BestFirstSolver` define the open list using the interface `Queue<State>`, but instantiate them as `LinkedList<State>` and `PriorityQueue<State>` respectively. This suggests we could gain further code reuse by combining these definitions into a common superclass. This is left as an exercise.

22.5 Putting the Framework to Use

All that remains in order to apply these search algorithms is to define an appropriate state representation for a problem. As an example, we define a state representation for the *framer, wolf, goat and cabbage*, FWGC, problem as a subclass of `AbstractState`. (We have presented representations and generalized search strategies for the FWGC problem in both Prolog, Chapter 4 and Lisp, Chapter 13. These different language-specific approaches to the same problem can offer insight into the design patterns and idioms of each language.)

The first step in this implementation is representing problem states. A simple, and for this problem effective, way to do so is to define the location of each element of the problem. Following a common Java idiom, we will create a user-defined type for locations using the Java `enum` capability. This simplifies readability of the code. We will also create two constructors, one that creates a default starting state with everyone on the east bank, and a private constructor that can create arbitrary states to be used in generating moves:

```
public class FarmerWolfGoatState extends
          AbstractState
{
    enum Side
    {
        EAST { public Side getOpposite()
```

```
                { return WEST; } },
        WEST { public Side getOpposite()
                { return EAST; } };
        abstract public Side getOpposite();
    }
    private Side farmer = Side.EAST;
    private Side wolf = Side.EAST;
    private Side goat = Side.EAST;
    private Side cabbage = Side.EAST;
    /**
     * Constructs a new default state with everyone on the east side.
     */
    public FarmerWolfGoatState()
    {}
    /**
     * Constructs a move state from a parent state.
     */
    public FarmerWolfGoatState(
             FarmerWolfGoatState parent,
             Side farmer, Side wolf,
             Side goat, Side cabbage)
     {
        super(parent);
        this.farmer = farmer;
        this.wolf = wolf;
        this.goat = goat;
        this.cabbage = cabbage;
    }
}
```

Having settled on a representation, the next step is to define the abstract methods specified in `AbstractState`. We define `isSolution()` as a straightforward check for the goal state, i.e., if everyone is on the west bank:

```
public boolean isSolution()
{
        return farmer==Side.WEST &&
               wolf==Side.WEST &&
               goat==Side.WEST &&
               cabbage==Side.WEST;
}
```

The most complex method is `getPossibleMoves()`. To simplify this definition, we will use the `getOpposite()` method defined above, and `addIfSafe(. . .)` will add the state to the set of moves if it is legal:

```
private final void addIfSafe(Set<State> moves)
{
        boolean unsafe =
              (farmer != wolf && farmer != goat) ||
              (farmer != goat && farmer != cabbage);
        if (!unsafe)
            moves.add(this);
}
```

Although simple, these methods have some interest, particularly their use of the `final` specification. This indicates that the method will not be redefined in a subclass. They also indicate to the compiler that it can substitute the actual method code for a function call as a compiler optimization. We implement `getPossibleMoves`:

```
public Iterable<State> getPossibleMoves()
{
        Set<State> moves = new HashSet<State>();
        if (farmer==wolf)                    //Move wolf
            new FarmerWolfGoatState(
                 this,farmer.getOpposite(),
                wolf.getOpposite(),
                 goat,
                 cabbage).addIfSafe(moves);
        if (farmer==goat)                    //Move goat
            new FarmerWolfGoatState(
                 this,farmer.getOpposite(),
                wolf,
                 goat.getOpposite(),
                 cabbage).addIfSafe(moves);
        if (farmer==cabbage)                 //Move cabbage
            new FarmerWolfGoatState(
                 this,farmer.getOpposite(),
                wolf,
                goat,
                 cabbage.getOpposite()).
                          addIfSafe(moves);
        new FarmerWolfGoatState(             //Move just farmer
                 this,farmer.getOpposite(),
                wolf,
                goat,
                 cabbage).addIfSafe(moves);
        return moves;
}
```

Although we will leave implementation of `getHeuristic()` as an exercise, there are a few more details we must address. Among the methods defined in Object (the root of the Java hierarchy), are `equals` and `hashCode`. We must override the default definitions of these because two states should be considered equal if the four participants are at the same location, ignoring the move count and parent states that are also recorded in states. Simple definitions of these methods are:

```
public boolean equals(Object o)
{
        if (o==null ||
             !(o instanceof FarmerWolfGoatState))
        return false;
        FarmerWolfGoatState fwgs =
                   (FarmerWolfGoatState)o;
        return farmer  == fwgs.farmer &&
               wolf    == fwgs.wolf &&
               cabbage == fwgs.cabbage &&
               goat    == fwgs.goat;
 }
public int hashCode()
{
        return (farmer  == Side.EAST ? 1 : 0)+
               (wolf    == Side.EAST ? 2 : 0)+
               (cabbage == Side.EAST ? 4 : 0)+
               (goat    == Side.EAST ? 8 : 0);
}
```

This chapter examined a number of Java techniques and idioms. Perhaps the most important, however, is the use of interfaces and abstract classes to specify a contract for viable extensions to the basic search methods. This was essential to our approach to building reusable search methods, and will continue to be an important abstraction method throughout Part IV.

Exercises

1. Building on the code and design patterns suggested in Chapter 22, finish coding and run the complete solution of the Farmer, Wolf, Coat, and Cabbage problem. Implement depth-first, breadth-first, and best-first solutions.

2. At the end of section 22.4, we noted that the `LinkedList<State>` and `PriorityQueue<State>` used to manage the open list in `BreadthFirstSolver` and `BestFirstSolver` respectively both used the interface `Queue<State>`. This suggests the possibility of creating a superclass of both solvers to provide shared definitions of the open list functions. Rewrite the solver framework to include such a class. What are the advantages of doing so for the maintainability, understandability, and usefulness of the framework? The disadvantages?

3. The current solver stops when it finds the first solution. Extend it to include a `nextSolution()` method that continues the search until it finds a next solution, and a reset() method that resets the search to the beginning.

4. Use the Java framework of Section 22.5 to create depth-first, breadth-first, and best-first solutions for the Missionary and Cannibal problem.

> Three missionaries and three cannibals come to the bank of a river they wish to cross. There is a boat that will hold only two, and any of the group is able to row. If there are ever more missionaries than cannibals on any side of the river the cannibals will get converted. Devise a series of moves to get all the people across the river with no conversions.

5. Use and extend your code of problem 4 to check alternative forms of the missionary and cannibal problem—for example, when there are four missionaries and four cannibals and the boat holds only two. What if the boat can hold three? Try to generalize solutions for the whole class of missionary and cannibal problems.

6. Use the Java framework of Section 22.5 to create depth-first, breadth-first, and best-first solutions for the Water Jugs problem:

> There are two jugs, one holding 3 and the other 5 gallons of water. A number of things can be done with the jugs: they can be filled, emptied, and dumped one into the other either until the poured-into jug is full or until the poured-out-of jug is empty. Devise a sequence of actions that will produce 4 gallons of water in the larger jug. (Hint: use only integers.)

23 A Java Representation for Predicate Calculus and Unification

Chapter Objectives

A brief introduction to the predicate calculus
- Predicates with:
  - Atoms
  - Variables
  - Functions
- A unification algorithm
  - A `car/crd` recursive tree walk
  - Ignoring occurs check

Techniques proposed for mapping predicate calculus to an object system
- A meta-linguistic interpreter
- Class hierarchy for `PCExpression`

Built `Unifiable` interface

Important discussion on design issues
- Building a representation for predicate calculus based reasoning
- Discussion in Section 23.3 and 23.5

First of three chapters covering predicate calculus
- Representation and unification (Ch 23)
- Reasoning with the predicate calculus (Ch 24)
- A rule-based expert system shell in Java (Ch 25)

Chapter Contents

23.1 Introduction to the Task

Although Java supports classes, inheritance, relations, and other structures used in knowledge representation, we should not think of it as a representation language in itself, but as a general purpose programming language. In AI applications, Java is more commonly used to implement interpreters for higher-level representations such as logic, frames, semantic networks, or state-space search. Generally speaking, representing the elements of a problem domain directly in Java classes and methods is only feasible for well-defined, relatively simple problems. The complex, ill-formed problems that artificial intelligence typically confronts require higher-level representation and inference languages for their solution.

The difference between AI representation languages and Java is a matter of semantics. As a general programming language, Java grounds object-oriented principles in the practical virtual machine architecture – the abstract architecture at the root of Java's platform independence – rather

than in mathematical systems or knowledge representation theories. Although Java draws on ideas from knowledge representation, such as class inheritance, its underlying semantics is procedural, defining loops, branches, variable scoping, memory addresses, and other machine constructs. This contrasts with higher-level knowledge representation languages, which draw their semantics from mathematical (formal logic or the lambda calculus), psychological (frames, semantic networks), or neural (genetic and connectionist network) theories of symbols, reference, and reasoning. The power of higher-level representation languages is in addressing the specific problems of reasoning about complex domains. This also simplifies the verification and validation of code, since a theory-based implementation can support code integrity better than the combination of machine semantics and often ill-defined user requirements.

Meta-linguistic abstraction is the technique of using one language to implement an interpreter for another language whose structure better fits a given class of problems. The term, *meta-linguistic*, refers to the use of a language's constructs to represent the elements of the target language, rather than elements of the final problem domain, as seen in Figure 23.1. We can think of meta-linguistic abstraction as a series of mappings, the arrows in Figure 23.1. The elements of a representation language, in this case, predicate calculus, are mapped into Java classes, and the entities in our problem domain are mapped into the representation language. If done carefully, this simplifies both mappings and their implementation – indeed, one of the benefits of this approach is that the theoretical basis of the representation language serves as a well-defined, mathematically grounded basis for the implementation. In turn, this simplifies both the implementation of the representation language, and the development of problem solvers.

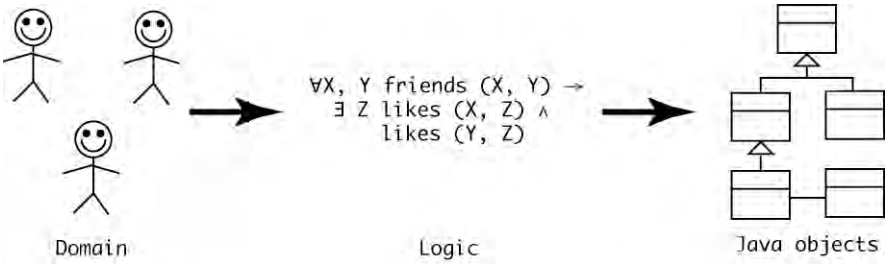

Figure 23.1. Creating an interpreter in Java to represent predicate calculus expressions, which, capture the semantics of a problem domain.

The search engines of Chapter 22 hinted at this technique through their use of solver and state classes to describe general search elements rather than defining classes for a particular problem, e.g, farmers, wolves, goats, and cabbages. Chapters 23, 24 and 25 provide a more sophisticated example of meta-linguistic abstraction, using Java to build an inference engine for first-order predicate calculus.

Implementing a logic-based reasoner in an object-oriented language like Java offers an interesting challenge, largely because the predicate calculus's "flat" declarative semantics is so different from that of Java. In the predicate calculus every predicate is a statement that is either true or false

for the domain of discourse; there is no hierarchy within predicate relationships, nor is there inheritance of predicates, variables, or truth-values across predicate expressions. In addition, the scope of variables is limited to a single predicate. What predicate calculus gives us in turn is representational generality and theoretically supported algorithms for logical inference and variable binding through unification.

In building a predicate calculus problem solver, we begin with simple predicate calculus expressions, and then implement a unification algorithm that determines the variable substitutions that make two expressions equivalent (Luger 2009, Section 2.3.3).

Chapter 24 addresses the representation of more complex logical expressions (`and`, `or`, `not` and `implies`), and then uses the unification algorithm as the basis of a logic problem-solver that searches an and/or graph of logical inferences. This problem solver then implements a depth-first search with backtracking, and constructs a proof tree for each solution found. This can be seen as building a Prolog interpreter in Java.

23.2 A Review of the Predicate Calculus and Unification

The predicate calculus is, first of all, a formal language: it is made up of tokens and a grammar for creating predicate names, variables, and constants. Chapter 2 of Luger (2009) describes predicate calculus in detail, but we offer a brief summary in this section.

The *atomic* unit of meaning in the predicate calculus is the predicate *sentence* or *expression*. A simple predicate expression, or simple sentence, consists of a predicate name, such as likes or friends in the following examples, followed by zero or more arguments. The arguments of predicates can be atoms (represented, by convention, as symbols beginning with a lower case letter), variables (symbols beginning with an upper case letter), or functions (in the same syntactic form as predicates). A function may itself have zero or more arguments, expressions separated by commas and enclosed by parentheses. A function is interpreted in the traditional manner, that is, by replacing it and its arguments, which are taken from its domain of definition, by the unique constant that is the function's evaluation. For example, `father_of(david)` is evaluated to `george`, when `george` is the computed father of `david`. Although our interpreter allows functions in expressions and will match them as patterns, we do not support their interpretation. Examples of simple sentences include:

```
likes(george,  kate).
likes(kate, wine).
likes(david, kate).
likes(kate, father_of(david)).
```

Predicate calculus also allows the construction of complex sentences using the operators, ∧ (and), ∨ (or), ¬ (not), and ← (implies). For example,

```
friends(X, Y) ← likes(X, Z)  ∧ likes(Y, Z)
```

can be interpreted as stating that `X` and `Y` are `friends` if there is some individual `Z` such that `X` and `Y` both have the `likes` relationship with `Z`. When using variables in logical expressions, there are various possibilities

for their interpretation. Predicate calculus uses variable quantification to specify the scope of meaning of the variables in a sentence. The above sentence is more properly written using the existential (∃) and universal (∀) quantifiers:

```
∀ X, Y (friends(X, Y) ←
  ∃ Z (likes(X, Z) ∧ likes(Y, Z)))
```

This can be read: "for all X and Y, X and Y are friends if there exists a Z such that X likes Z and Y likes Z."

Horn Clauses and Unification

In developing an inference engine for predicate calculus, we will follow Prolog conventions and restrict ourselves to a subset of logical expressions called *Horn Clause Logic*. Although their theoretical definition is more complex, for our purposes, we can think of a Horn clause as an implication, or rule, with only a single predicate on the left hand side of the implication, ←. The right hand side, or "body" of the clause can be empty, a simple predicate, or any syntactically well-formed expression made up of simple predicates. Horn clauses may consist of the body only; these are called goal clauses. Examples in propositional form include:

```
p ← q ∧ r ∧ s
p ←
q ∧ r ∧ s
```

Following Prolog conventions, we extend the definition to allow ∨ (or), and ¬ (not) in the body of the Horn clause. In order to accommodate Java syntax, we vary these syntactic conventions in ways that will be evident over the next few chapters.

The power of Horn clauses is in their simplification of logical reasoning. As we will see in the next chapter, restricting the head of implications to a single predicate simplifies the development of a backward chaining search engine. To answer the query of whether two people `X` and `Y` can be found who are `friends`, a search engine must determine if there are any variable substitutions for `X`, `Y`, and `Z` that satisfy the `likes(X, Z)` and `likes(Y, Z)` relationships. In this case, the substitutions `george/X`, `david/Y`, and `kate/Z` lead to the conclusion: `friends(george, david)`. *Unification* is the algorithm for matching predicate calculus expressions and managing the variable substitutions generally required for such matches. Unification, combined with the use of search to try all possible matches, form the heart of a logic problem solver.

Logic-based reasoning requires determining the equivalence of two expressions. For example, consider the reasoning schema, *modus ponens*:

```
Given: q(X) ← p(X) and p(george)
Infer: q(george)
```

We can read this as "if `p` of `X` implies `q` of `X`, and `p(george)` are both true, then we infer that `q(george)` is true as well." This inference is a result of the equivalence of the fact "`p(george)`" and the premise "`p(X)`" in the implication. What makes this difficult for the predicate calculus is the more complex structure of sentences, and, more importantly, the handling of variables. In the example above, the sentence

```
likes(X, Z)  ∧ likes(Y, Z)
```

matched the sentences `likes(george, kate)` and `likes(david, kate)` under the variable bindings `george/X`, `david/Y` and `kate/Z`. Note that, although the variable `Z` appears in two places in the sentence, it can only be *bound* once: in this case, to `kate`. In another example, the expression `likes(X, X)` could not match the sentence since `X` can only be bound to one constant. The algorithm for determining the bindings under which two predicate expressions match is called *unification*.

The unification algorithm determines a minimal set of variable bindings under which two sentences are equivalent. This minimal set of bindings is called the *unifier*. It maintains these bindings in a *substitution set*, a list of variables paired with the expressions (constants, functions or other variables) to which they are bound. It also insures consistency of these bindings throughout their scope. It is also important to recognize that in the above example, where `X` is bound to `george`, it is possible for the variable `X` to appear in different predicates with different scopes and bindings. Unification must manage these different contexts as well.

Luger (2009, Chapter 2) defines the unification algorithm as returning a substitution set if two expressions unify. The algorithm treats the expressions as lists of component expressions, a technique we will use in our own implementation:

```
function unify(E1, E2)
  begin
     case
        both E1 and E2 are constants or empty list:
           if E1 = E2 then
              return the empty substitution set
           else return FAIL;
        E1 is a variable:
           if E1 is bound then
              return unify(binding of E1, E2);
           if E1 occurs in E2 then return FAIL;
           return the substitution set {E1/E2};
        E2 is a variable:
           if E2 is bound then
              return unify(binding of E2, E1);
           if E2 occurs in E1 then return FAIL
           else return the substitution set {E2/E1};
        either E1 or E2 are different lengths:
           return FAIL;
        otherwise:
           HE1 = first element of E1;
           HE2 = first element of E2;
           S1 = unify(E1, E2);
           if S1 = FAIL then return Fail;
```

```
TE1 = apply S1 to the tail of E1;
TE2 = apply S1 to the tail of E2;
S2 = unify (TE1, TE2);
if S2 = FAIL then return FAIL
else return the composition of S1 and S2;
```

Although the algorithm is straightforward, a few aspects of it are worth noting. This is a recursive algorithm, and follows the pattern of head/tail recursion already discussed for Lisp and Prolog. We will retain a recursive approach in the Java implementation, although we will adapt it to an object-oriented idiom. Variables may bind to other variables; in this case, if either of the variables becomes bound to a constant or function expression, then both will share this binding. Finally, note that before binding a variable to an expression, the algorithm first checks if the variable is in the expression. This is called the *occurs check*, and it is necessary because, if a variable binds to an expression that contains it, replacing all occurrences of the variable with the expression will result in an infinite structure. We will omit the occurs check in our implementation, both for efficiency (as with Prolog) and to simplify the discussion. We do, however, leave its implementation as an exercise.

23.3 Building a Predicate Calculus Problem Solver in Java

Representing Basic Predicates

As with any object-oriented implementation, we began with representation, defining the elements of the predicate calculus as Java classes. In the present chapter, we define the classes `Constant`, `Variable`, and `SimpleSentence`. In Chapter 24 we define the classes `And`, and `Rule` (or `Implies`). We will leave the definition of `Or` and `Not` as an exercise.

We organize `Constant`, `Variable`, and `SimpleSentence` into a hierarchy, with the interface `PCExpression` as its root. As we develop the algorithm, this interface will come to define signatures for methods shared across all predicate calculus expressions.

```
public interface PCExpression {}
```

We also create a child interface, called `Unifiable`, that defines the signature for the `unify(. . .)` method, which is the focus of this chapter. For now, we leave the arguments to `unify` unspecified.

```
public interface Unifiable extends PCExpression
  { public boolean unify(. . .); }
```

The reason for introducing the `Unifiable` interface will become evident as the discussion moves into Chapter 24.

Based on these interfaces, we define the classes shown in the hierarchy of Figure 23.2.

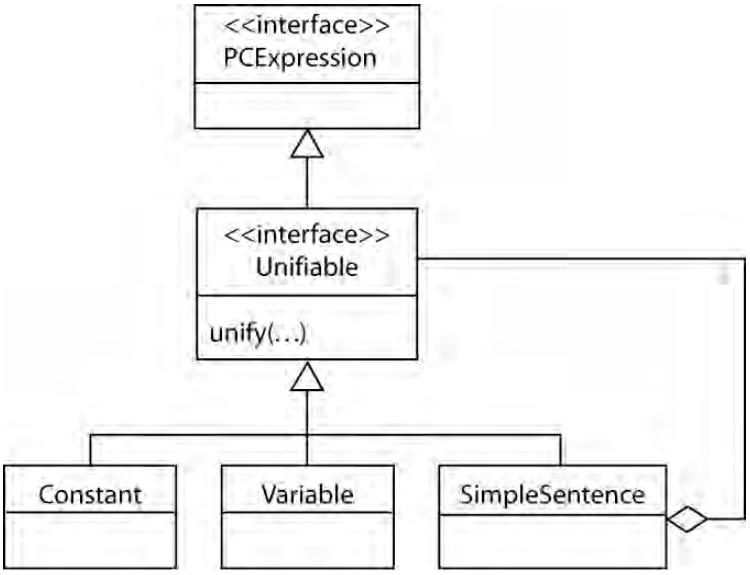

Figure 23.2. The class hierarchy for PCExpression.

`Constant`, `Variable` and `SimpleSentence` are all subclasses of `Unifiable`. Each instance of these classes will have a one-to-one relationship to its corresponding expression. This simplifies testing if constants or variables are the same: they are the same if and only if they are the same Java object. We can test this using the "`==`" operator. A `SimpleSentence` is an assembly of instances of `Unifiable`. This recursive structure allows sentences such as:

```
likes(kate, wine).
likes(david, X).
likes(kate, father_of(david))
```

Note that we do not distinguish between predicate expressions and functions in this implementation. This works for our implementation since we are not evaluating functions, and the syntax of predicate expressions and functions is the same. Introducing evaluable functions into this model is left as an exercise. Under this approach, an initial implementation of `Constant` is straightforward:

```
public class Constant implements Unifiable
{
  private String printName = null;
  private static int nextId = 1;
  private int id;
  public Constant()
  {
     this.id = nextId++;
  }
  public Constant(String printName)
  {
     this();
```

```
        this. printName= printName;
    }
    public String toString()
    {
        if (printName!= null)
            return printName;
        return "constant_" + id;
    }
                    //unify and other functions to be defined shortly.
}
```

This definition includes a `printName` member variable, which we will use to display constants like `george` or `kate` in our earlier example. This practice of distinguishing the display name of an object from its internal representation is a common object-oriented practice that allows us to optimize internal representation while maintaining a familiar "face" for printing objects. `Variable` has a nearly identical definition:

```
public class Variable implements Unifiable
{
    private String printName = null;
    private static int nextId = 1;
    private int id;
    public Variable()
    {
        this.id = nextId++;
    }
    public Variable(String printName)
    {
        this();
        this.printName = printName;
    }
    public String toString()
    {
        if (printName != null)
            return printName + "_" + id;
        return "V" + id;
    }
                    //unify and other functions to be defined shortly.
}
```

The `id` member variable in these classes serves several purposes. It can serve as an identifier for unnamed constants or variables. Although, generally speaking, unnamed constants and variables can be confusing, we include them for completeness and to be consistent with similar features in Prolog. A more important use of the `id` variable is in the `Variable` class.

As mentioned earlier, the same variable name may be used in different sentences, where it is treated as different variables. By appending the `id` to the `printName` in the `toString` method, we enable the programmer to more easily distinguish variables in this case, simplifying tracing of program execution.

Also note the introduction of the constructor `Variable(Variable v)`. This pattern is called a *copy constructor*, and serves the same function as the `clone` method. We prefer this approach because the semantics of `clone` are problematic, with programmers frequently redefining it to reflect their own needs. Using a copy constructor emphasizes that the semantics of the copy are specific to the class. In the case of `Variable`, we define copying to use the same `printName` but a different `id`; this will be important for dealing with occurrences of the same variable in different contexts, as presented in Chapter 24.

As we mentioned, for the portion of the definition displayed above, `Constant` and `Variable` are essentially the same, and this part of their definition could be placed in a common parent class. We have chosen not to do so, feeling that the functionality is so simple that a common parent definition would buy us too little in the way of maintainability to justify the added complexity of doing so. However, like many design decisions, this is a matter of taste. We encourage the reader to explore the trade-offs of this decision on her own.

Finally, we define `SimpleSentence` as an array of type `Unifiable`:

```
public class SimpleSentence implements Unifiable
{
  private Unifiable[] terms;

  public SimpleSentence(Constant predicateName,
        Unifiable... args)

  {
     this.terms = new Unifiable[args.length + 1];
     terms[0] = predicateName;
     System.arraycopy(args, 0, terms, 1,
                 args.length);
  }

  private SimpleSentence(Unifiable... args)

  {
     terms = args
  }

  public String toString()

  {
     String s = null;
     for (Unifiable p : terms)
        if (s == null)
           s = p.toString();
```

```
            else
               s += " " + p;
            if (s == null)
            return "null";
         return "(" + s + ")";
      }
      public int length()
      {
         return terms.length;
      }
      public Unifiable getTerm(int index)
      {
         return terms[index];
      }
                        //unify and other functions to be defined shortly.
   }
```

Representing a `simpleSentence` as an array of `Unifiable` simplifies access to its elements using the `length` and `getTerm` methods. These will be important to our implementation of the `unify` method. Although first-order predicate calculus requires that the first term of a simple sentence be a constant, to gain the benefits of using an array of type `Unifiable` to represent simple sentences, we did not make this distinction internally. Instead, we enforce it in the constructor. This approach maintains the integrity of the Predicate Calculus implementation, while giving is the internal simplicity of representing a simple sentence as an array of items of type `Unifiable`.

Defining unify(...) and Substitution Sets

To complete this part of the implementation, we need to define the `unify` method. `unify` has the signature:

```
public SubstitutionSet unify(Unifiable p,
  SubstitutionSet s)
```

A call to unify takes a `Unifiable` expression and a `SubstitutionSet` containing any bindings from the algorithm so far. For example, if `exp1` and `exp2` are both of type `Unifiable`, and `s` is an initial `SubstitutionSet`, we unify the expressions by:

```
exp1.unify(exp2, s)
```

or by:

```
exp2.unify(exp1, s)
```

Both calls are equivalent. If the unification succeeds, unify will return a new substitution set, adding additional variable bindings to those passed in the parameters. If it fails, `unify` will return `null.` In either case, the original `SubstitutionSet` will be unchanged. The class maintains variable bindings as a list of `Variable/Unifiable` pairs, using the `HashMap` collection type:

```
public class SubstitutionSet
{
  private HashMap<Variable, Unifiable> bindings =
     new HashMap<Variable, Unifiable>();
  public SubstitutionSet(){}
  public SubstitutionSet(SubstitutionSet s)
  {                  this.bindings =
        new HashMap<Variable,
           Unifiable>(s.bindings);
  }
  public void clear()
  {
     bindings.clear();
  }
  public void add(Variable v, Unifiable exp)
  {
     bindings.put(v, exp);
  }
  public Unifiable getBinding(Variable v)
  {
     return (Unifiable)bindings.get(v);
  }
  public boolean isBound(Variable v)
  {
     return bindings.get(v) != null;
  }
  public String toString()
  {
     return "Bindings:[" + bindings + "]";
  }
}
```

This is a straightforward implementation that does little more than "wrap" the `HashMap` in the `SubstitutionSet` object (see the design discussion in section 23.3.1 for the reasons behind this approach). Finally, we will define `unify` for each class. Implementing it for `Constant` is straightforward and addresses two cases: if the expression to be matched is equal to the constant, unify returns a new substitution set; if the expression is a variable, it calls `unify` on that variable.

```
public class Constant implements Unifiable
{
  //constructors and other methods as defined above.
```

```
    public SubstitutionSet unify(Unifiable exp,
                                 SubstitutionSet s)
    {
       if (this == exp)
          return new SubstitutionSet(s);
       if (exp instanceof Variable)
          return exp.unify(this, s);
       return null;
    }
}
```

Defining `unify` for a variable is a bit more complicated, since it must manage bindings:

```
public class Variable implements Unifiable
{
   // constructors and other methods as defined above
   public SubstitutionSet unify(Unifiable p,
               SubstitutionSet s)
   {
      if (this == p) return s;
      if(s.isBound(this))
         return s.getBinding(this).unify(p, s);
      SubstitutionSet sNew = new SubstitutionSet(s);
      sNew.add(this, p);
      return sNew;
   }
}
```

This definition checks three cases. The first is if the expressions are equal: anything matches itself. Second, it checks if the variable is bound in the substitution set `s`; if it is, it retrieves the binding, and calls `unify` on it. Finally, the variable is unbound and the algorithm adds the binding to a new substitution set and returns it.

We define `unify` for `SimpleSentence` as unifying two lists by moving through them, unifying corresponding elements. If this succeeds, it returns the accumulated substitutions.

```
public SubstitutionSet unify(Unifiable p,
        SubstitutionSet s)
{
   if (p instanceof SimpleSentence)
   {
      SimpleSentence s2 = (SimpleSentence) p;
      if (this.length() != s2.length())
         return null;
      SubstitutionSet sNew = new SubstitutionSet(s);
```

```
            for (int i = 0; i < this.length(); i++)
            {
               sNew = this.getTerm(i).unify(s2.getTerm(i),
                        sNew);
               if(sNew == null)
                  return null;
            }
            return sNew;
         }
         if(p instanceof Variable)
            return p.unify(this, s);
         return null;
      }
```

This method tests two cases. If the argument `p` is a simple sentence, it casts `p` to an instance of `SimpleSentence`: `s2`. As an efficiency measure, the method checks to make sure both simple sentences are the same length, since they cannot match otherwise. Then, the method iterates down the elements of each simple sentence, attempting to unify them recursively. If any pair of elements fails to unify the entire unification fails. The second case is if `p` is a `Variable`. If so, the method calls `unify` on `p`.

Testing the unify Algorithm

To simplify testing of the algorithm, we write one more method to replace any bound variable with its binding. We will define this method signature at the level of the interface `PCExpression`:

```
public interface PCExpression
{
   public PCExpression
         replaceVariables(SubstitutionSet s);
}
```

Defining the method for a constant is straightforward:

```
public class Constant implements Unifiable
{
                  //Use constructors and other methods as defined above.
   public PCExpression
      replaceVariables(SubstitutionSet s)
   {
      return this;
   }
}
```

In the case of variables, the method must search the substitution set to find the binding of the variable. Since a variable may be bound to other variables, the method must search until it finds a constant binding or a final, unbound variable:

```
public class Variable implements Unifiable
{
```

```
                    //Use constructors and other methods as defined above.
  public PCExpression replaceVariables(
        SubstitutionSet s)
  {
     if(s.isBound(this))
        return
           s.getBinding(this).replaceVariables(s);
     else
        return this;
  }
}
```

Finally, a `SimpleSentence` replaces variables with bindings in all its terms, and then creates a new sentence from the results:

```
public class SimpleSentence implements Unifiable
{
                    //Use constructors and other methods as defined above.
  public PCExpression
        replaceVariables(SubstitutionSet s)
  {
     Unifiable[] newTerms = new
        Unifiable[terms.length];
     for(int i = 0; i < length(); i++)
        newTerms[i] =
           (Unifiable)terms[i].replaceVariables(s);
     return new SimpleSentence(newTerms);
}
```

Using these definitions of our key objects, an example `UnifyTester` can create a list of expressions and try a series of goals against them:

```
public class UnifyTester
{
  public static void main(String[] args)
  {
     Constant friend = new Constant("friend"),
              bill = new Constant("bill"),
              george = new Constant("george"),
              kate = new Constant("kate"),
              merry = new Constant("merry");
     Variable X = new Variable("X"),
              Y = new Variable("Y");
     Vector<Unifiable> expressions =
           new Vector<Unifiable>();
     expressions.add(new SimpleSentence(friend,
           bill, george));
```

```
            expressions.add(new SimpleSentence(friend,
                  bill, kate));
            expressions.add(new SimpleSentence(friend,
                  bill, merry));
            expressions.add(new SimpleSentence(friend,
                  george, bill));
            expressions.add(new SimpleSentence(friend,
                  george, kate));
            expressions.add(new SimpleSentence(friend,
                  kate, merry));
            //Test 1
            Unifiable goal = new SimpleSentence(friend, X,
                  Y);
            Iterator iter = expressions.iterator();
            SubstitutionSet s;
            System.out.println("Goal = " + goal);
            while(iter.hasNext()){
               Unifiable next = (Unifiable)iter.next();
               s = next.unify(goal, new SubstitutionSet());
               if(s != null)
                  System.out.println(
                        goal.replaceVariables(s));
               else
                  System.out.println("False");
            }
            //Test 2
            goal = new SimpleSentence(friend, bill, Y);
            iter = expressions.iterator();
            System.out.println("Goal = " + goal);
            while(iter.hasNext()){
               Unifiable next = (Unifiable)iter.next();
               s = next.unify(goal, new SubstitutionSet());
               if(s != null)
                  System.out.println(
                     goal.replaceVariables(s));
               else
                  System.out.println("False");
            }
         }
      }
```

`UnifyTester` creates a list of simple sentences, and tests if several goals bind with them. It produces the following output:

```
Goal = (friend X_1 Y_2)
(friend bill george)
(friend bill kate)
(friend bill merry)
(friend george bill)
(friend george kate)
(friend kate merry)
Goal = (friend bill Y_2)
(friend bill george)
(friend bill kate)
(friend bill merry)
False
False
False
```

We leave the creation of additional tests to the reader.

23.4 Design Discussion

Although simple, the basic `unify` method raises a number of interesting design questions. This section addresses these in more detail.

Why define the substitutionSet class?

The `SubstitutionSet` class has a very simple definition that adds little to the `HashMap` class it uses. Why not use `HashMap` directly? This idea of creating specialized data structures around Java's general collection classes gives us the ability to address problem specific questions while building on more general functionality. A particular example of this is in detecting problem specific errors. Although the `SubstitutionSet` class defined in this chapter works when used properly, it could be used in ways that might lead to subtle bugs. Specifically, consider the `add()` method:

```
public void add(Variable v, Unifiable exp)
{
  bindings.put(v, exp);
}
```

As defined, this method would allow a subtle bug: a programmer could add a binding for a variable that is already bound. Although our implementation makes sure the variable is not bound before adding a binding, a more robust implementation would prevent any such errors, throwing an exception if the variable is bound:

```
public void add(Variable v, Unifiable exp)
{
  if(isBound(v)
     //Throw an appropriate exception.
  else
     bindings.put(v, exp);
}
```

Finishing this method is left as an exercise.

Why is unify a method of unifiable?

An alternative approach to our implementation would make `unify` a method of `SubstitutionSet`. If we assume that `exp1` and `exp2` are `Unifiable`, and `s` is a `SubstitutionSet`, unifications would look like this:

```
s.unify(exp1, exp2);
```

This approach makes sense for a number of reasons. `SubstitutionSet` provides an essential context for unification. Also, it avoids the somewhat odd syntax of making one expression an argument to a method of another (`exp1.unify(exp2, s)`), even though they are equal arguments to what is intuitively a binary operator. Although harmless, many programmers find this asymmetry annoying.

In preparing this chapter, we experimented with both approaches. Our reasons for choosing the approach we did is that adding the `unify` method to the `SubstitutionSet` made it more than a "unification-friendly" data structure, giving it a more complex definition. In particular, it had to test the types of its arguments, an action our approach avoids. A valuable design guideline is to attempt to make all objects have simple, well-defined behaviors as this can reduce the impact of future changes in the design. Exercise 4 asks the reader to implement and evaluate both approaches.

Why introduce the interface definition: unifiable?

Early in the chapter, we introduced the `Unifiable` interface to define the `unify` method signature, rather than making `unify` a method of `PCExpression`. This raises a design question, since, as defined in (Luger 2009), the unification algorithm can apply to all Predicate Calculus expressions (including expressions containing *implies*, *and*, *or* and *not*).

The short answer to this question is that we could have placed the unification functionality in `PCExpression`. However, as we move into expressions containing operators, we encounter the added problems of search, particularly for implications, which can be satisfied in different ways. We felt it better to separate simple unification (this chapter) from the problems of search we present in Chapter 24) for several reasons.

First, since unifying two expressions with operators such as *and* decomposes into unifying their component terms, we can isolate the handling of variable bindings to simple sentences.

Second, our intuitions suggest that adding search to our problem solver naturally creates a new context for our design. Separating search from unification simplifies potential problems of adding specialized search capabilities to our problem solver.

Finally, our goal in a logic problem solver is not only to find a set of variable bindings that satisfy a goal, but also to construct a trace of the solution steps. This trace, called a *proof tree*, is an important structure in expert systems reasoning, as presented in Chapter 25. For simplicity sake, we did not want to include atomic items, e.g., constants and variables, into the proof tree. As we will see in the next chapter, introducing the `Unifiable` interface helps with this.

23.5 Conclusion: Mapping Logic into Objects

In the introduction to this chapter, we argued for the benefits of meta-linguistic abstraction as an approach to developing large-scale problem solvers, knowledge systems, learning programs, and other systems common to Artificial Intelligence. There are a number of advantages of taking this approach including:

Reuse. The most obvious benefit of meta-linguistic abstraction is in reuse of the high-level language. The logic reasoner we are constructing, like the Prolog language it mirrors, greatly simplifies solving a wide range of problems that involve logical reasoning about objects and relations in a domain. The Prolog chapters of this book illustrate the extent of logic's applicability. By basing our designs on well-structured formalisms like logic, we gain a much more powerful foundation for code reuse than the ad-hoc approaches often used in software organizations, since logic has been designed to be a general representation language.

Expressiveness. The expressiveness of any language involves two questions. What can we say in the language? What can we say easily? Formal language theories have traditionally addressed the first question, as reflected in the Chomsky hierarchy of formal languages (Luger 2009, Section 15.2), which defines a hierarchy of increasingly powerful languages going from regular expressions to Turing complete languages like Java, Lisp, or Prolog. Predicate calculus, when coupled with the appropriate interpreter is a complete language, although there are benefits to less expressive languages. For example, regular expressions are the basis of many powerful string-processing languages. The second question, what can we say easily, is probably of more practical importance to Artificial Intelligence. Knowledge representation research has given us a large number of languages, each with their own strengths. Logic has unique power as a model of sound reasoning, and a well-defined semantics. Semantic networks and Frames give us a psychologically plausible model of memory organization. Semantic networks also support reasoning about relationships in complex linguistic or conceptual spaces. Genetic algorithms implement a powerful heuristic for searching the intractable spaces found in learning and similar problems by unleashing large populations of simple, hill-climbing searches throughout the space. Although knowledge representation research is not in its infancy, it is still a young field that will continue to provide formalisms for the design of meta-languages.

Support for Design. Although languages like logic are very different from an object-oriented language like Java, object-orientation is surprisingly well suited to building interpreters for meta-linguistic abstraction. The reason for this is that, as formal languages, representation schemes have clearly defined objects and relations that support the standard object-oriented design process of mapping domain objects, relations, and behaviors into Java classes and methods. Although, as discussed above, design still involves hard choices with no easy answer, objects provide a strong framework for design.

Semantics and Interpretation. In this chapter, we have focused on the representation of predicate calculus expressions. The other aspect of meta-linguistic abstraction is semantic: how are these expressions interpreted in problem solving. Because many knowledge representation languages, when properly designed, have their foundations in formal mathematics, they offer a clear basis for implementing program behavior. In the example of predicate calculus, this foundation is in logic reasoning using inference rules like modus ponens and resolution. These provide a clear blueprint for implementing program behavior. As was illustrated by our approach to unification in this chapter, and logic-based reasoning in the next, meta-linguistic abstraction provides a much sounder basis for building quality software in complex domains.

Exercises

1. In the friends example of Section 23.1, check whether there are any other situations where friends(X, Y) is true. How many solutions to this query are there? How might you prevent someone from being friends of themselves?

2. Review the recursive list-based unification algorithm in Luger (2009, Section 2.3.3). Run that algorithm on the predicate pairs of friends(george, X, Y), friends(X, fred, Z), and friends(Y, bill, tuesday). Which pairs unify, which fail and why? The unification algorithm in this chapter is based on the Luger (2009, Section 2.3.3) algorithm without the backtrack component and occurs check.

3. Section 23.3.1 suggested augmenting the `SubstitutionSet` data structure with problem-specific error detection. Do so for the class definition, beginning with the example started in that section. Should we define our own exception classes, or can we use built-in exceptions? What other error conditions could we present?

4. Rewrite the problem solver to make `unify` a method of `SubstitutionSet`, as discussed in 23.3.2. Compare this approach with the chapter's approach for ease of understanding, ease of testing, maintainability, and robustness.

5. As defined, the unify method creates a new instance of `SubstitutionSet` each time it succeeds. Because object creation is a relatively expensive operation in Java, this can introduce inefficiencies into the code. Our reasons for taking this approach include helping the programmer avoid inadvertent changes to the `SubstitutionSet` once we introduce unification into the complex search structures developed in the next chapter. What optimizations could reduce this overhead, while maintaining control over the `SubstutionSet`?

6. Add a class `Function` to define evaluable functions to this model. A reasonable approach would be for the class `Function` to define a default evaluation method that returns the function as a pattern to be unified. Subclasses of `Function` can perform actual evaluations of interest. Test your implementation using functions such as simple arithmetic operations, or an `equals` method that returns `true` or `false`.

7. Representing predicate calculus expressions as Java objects simplifies our implementation, but makes it hard to write the expressions. Write a "front end" to the problem solver that allows a user to enter logical expressions in a friendlier format. Approaches could include a Lisp or Prolog like format or, what is more in the spirit of Java, an XML syntax.

24 A Logic-Based Reasoning System

Chapter Objectives

Predicate calculus representation extended:
- Continued use of meta-linguistic abstraction
- Java interpreter for predicate calculus expressions

Search supported by unification
Proof trees implemented
- Capture logic inferences
- Represent structure of logic-based implications
- Reflect search path for producing proofs

`Tester` class developed
- Tests code so far built

Extensions proposed for user transparency
Java idioms utilized
Implement separation of representation and search
- Use of static structures

Chapter Contents

24.1 Introduction

Chapter 23 introduced *meta-linguistic abstraction* as an approach to solving the complex problems typically found in Artificial Intelligence. That chapter also began a three-chapter (23, 24, and 25) exploration this idea through the development of a reasoning engine for predicate calculus. Chapter 23 outlined a scheme for representing predicate calculus expressions as Java objects, and developed the unification algorithm for finding a set of variable substitutions, if they exist, that make two expressions in the predicate calculus equivalent. This chapter extends that work to include more complex predicate expressions involving the logical operators *and*, ∧, *or*, ∨, *not*, ¬, and implication, ←, and develops a reasoning engine that solves logic queries through the backtracking search of a state space defined by the possible inferences on a set of logic expressions.

24.2 Reasoning in Logic as Searching an And/Or Graph

A *logic-based reasoner* searches a space defined by sequences of valid inferences on a set of predicate logic sentences. For example:

```
likes(kate, wine).
likes(george, kate).
likes(david, kate).
friends(X, Y) ← likes(X, Z) ∧ likes(Y, Z).
```

We can see intuitively that, because both `likes(george, kate)` and `likes(david, kate)` are true, it follows from the "`friends` rule" that `friends(george, david)` is true. A more detailed explanation of this reasoning demonstrates the search process that constructs these conclusions formally. We begin by unifying the goal query, `friends(george, david)`, with the conclusion, or head of the `friends` predicate under the substitutions `{george/X, david/Y}`, as seen in Figure 24.1.

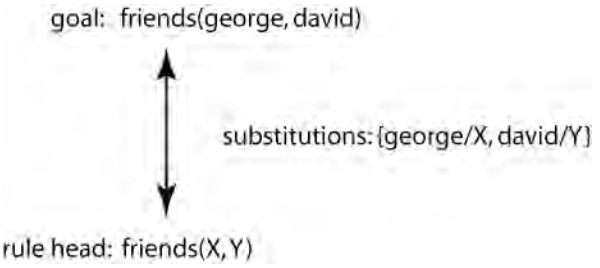

Figure 24.1. The set of variable substitutions, found under unification, by which the two `friends` predicates are identical.

Figure 24.2 illustrates the result of propagating these substitutions through the body of the rule. As the figure suggests, under the inference rule of *modus ponens*, `friends(george, david)` is true if there exists some binding for `Z` such that `likes(george, Z)` and `likes(david, Z)` are true. When viewed in terms of search, this leads to the sub-goal of proving the rule premise, or that the "tail," of the rule is true. Figure 24.2 illustrates this structure of reasoning as a graph. The arc joining the branches between the two `likes` predicates indicates that they are joined by a logical `and`. For the conclusion `friends(george, david)` to be true, we must find a substitution for `Z` under which both `likes(george, Z)` and `likes(david, Z)` are true. Figure 24.2 is an example of a representation called an *and/or* graph (Luger 2009, Section 3.3). And/or graphs represent systems of logic relationships as a graph that can be searched to find valid logical inferences. *And* nodes require that all child branches be satisfied (found to be true) in order for the entire node to be satisfied. *Or* nodes only require that one of the child branches be satisfied.

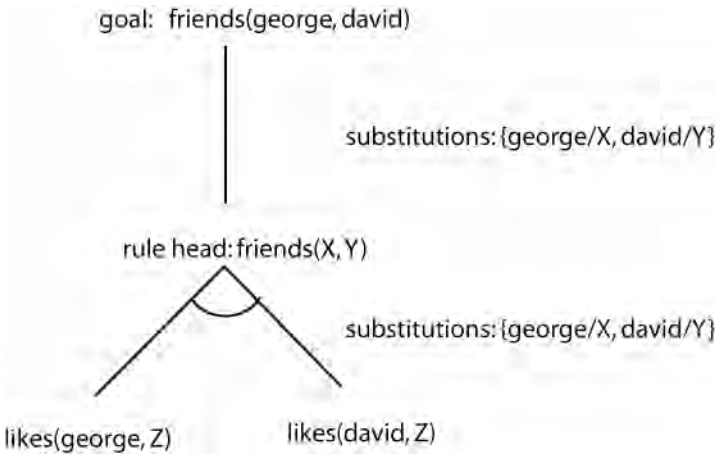

Figure 24.2. Substitution sets supporting the graph search of the `friends` predicate.

As we continue building this graph, the next step is to match the sentence `likes(george, Z)` with the different `likes` predicates. The first attempt, matching `likes(george, Z)` with `likes(kate, wine)` fails to unify. Trying the second predicate, `likes(george, kate)` results in a successful match with the substitution `{kate/Z}`, as in Figure 24.3.

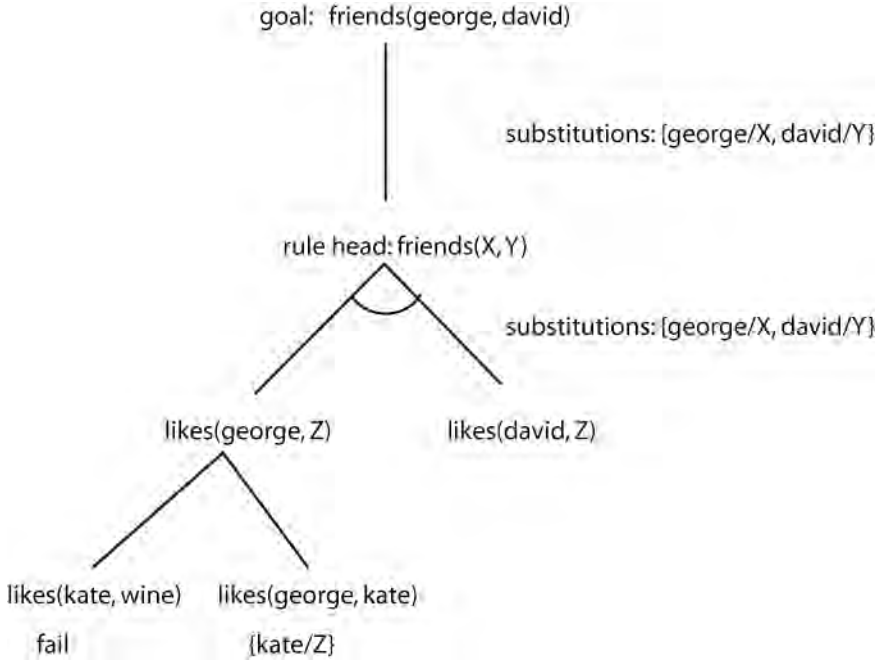

Figure 24.3 Substitution sets supporting the search to satisfy the `friends` predicate.

Note that the branches connecting the goal `likes(george, Z)` to the different attempted matches in the graph are not connected. This indicates an *or* node, which can be satisfied by matching any one of the branches.

The final step is to apply the substitution `{kate/Z}` to the goal sentence `likes(david, Z)`, and to try to match this with the logic expressions. Figure 24.4 indicates this step, which completes the search and proves the initial `friends` goal to be true. Note again how the algorithm tries the alternative branches of the *or* nodes of the graph to find a solution.

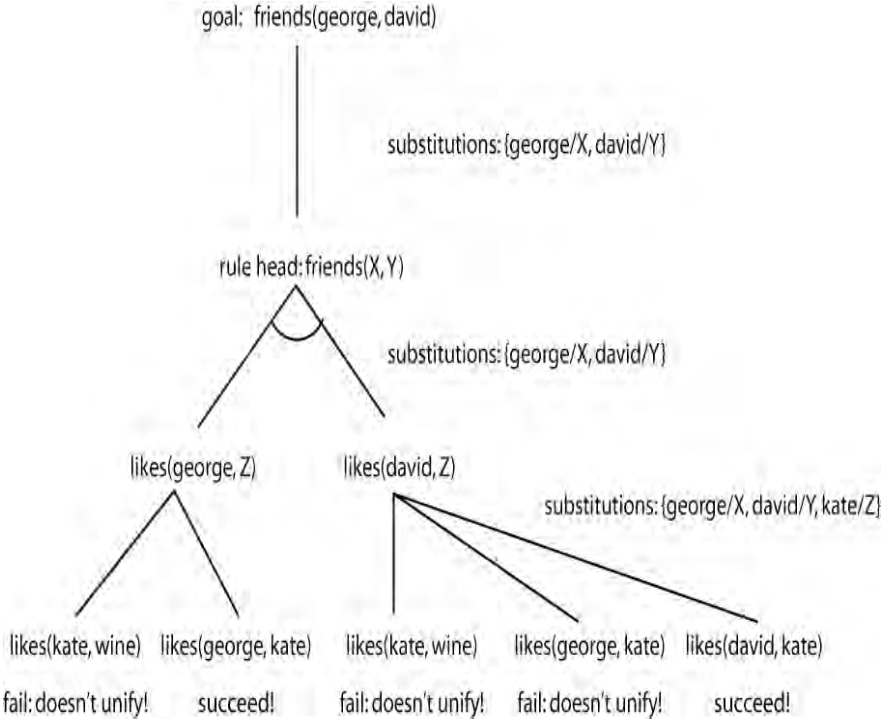

Figure 24.4. A search-based solution of the `friends` relationship.

This process of trying alternative branches of a state space can be implemented as a backtracking search. If a goal in the search space fails, such as trying to match `likes(george, Z)` and `likes(kate, wine)`, the algorithm backtracks and tries the next possible branch of the search space. The basic backtracking algorithm is given in (Luger 2009, Section 3.2) as:

> If some state S does not offer a solution to a search problem, then open and investigate its first child S_1 and apply the backtrack procedure recursively to this node. If no solution emerges from the subtree rooted by S_1 then fail S_1 and apply backtrack recursively to the second child S_2. Continuing on, if no solution emerges from any of the children of S, then fail back to S's parent and apply backtrack to S's first sibling.

Before implementing our logic-based reasoner as a backtracking search of and/or graphs, there is one more concept we need to introduce. That is the notion of a *proof tree*. If we take only the successful branches of our search, the result is a tree that illustrates the steps supporting the conclusion, as can be seen in Figure 24.5. In implementing a logic-based reasoning system, we not only search an and/or graph, but also construct the proof tree illustrating a successful search.

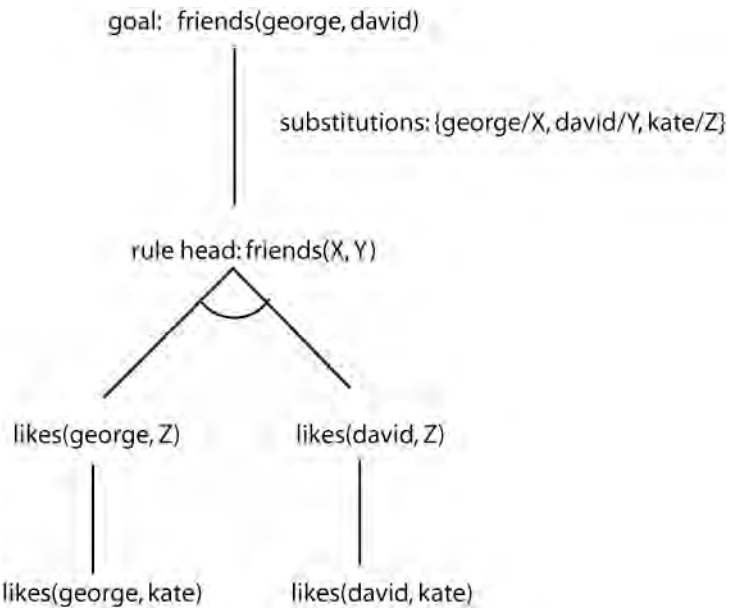

Figure 24.5. A proof tree showing the successful satisfaction of the `friends` predicate.

24.3 The Design of a Logic-Based Reasoning System

The first step in designing a logic-based reasoning system is to create a representation for the logical operators *and*, ∧, *or*, ∨, *not*, ¬, and implication, ←. Figure 24.6 begins this process by adding several classes and interfaces to those described in Figure 23.2.

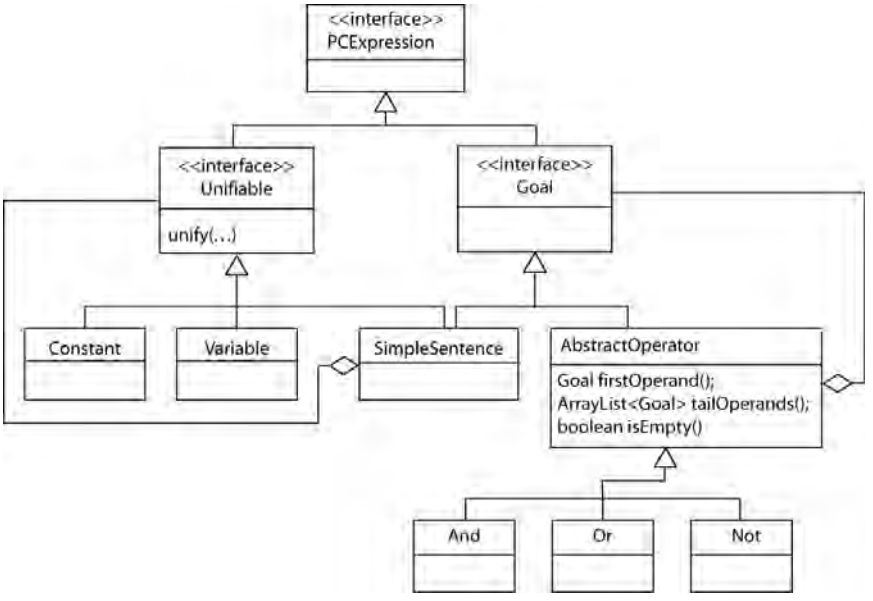

Figure 24.6. Classes and interfaces for a logic-based inference system.

The basis of this extension is the interface, `Goal`. Expressions that will appear as goals in an and/or graph must implement this interface. These include `SimpleSentence`, and the basic logical operators. We will add methods to this interface shortly, but first it is worth looking at a number of interesting design decisions supported by this object model.

The first of these design decisions is to divide `PCExpressions` into two basic groups: `Unifiable`, which defines the basic unification algorithm, and `Goal` which defines nodes of our search space. It is worth noting that, when we were developing this algorithm, our initial approach did not make this distinction, but included both basic unification and the search of logical operators in the `unify` method, which was specified in the top-level interface, `PCExpression`.

We chose to re-factor the code and divide this functionality among the two interfaces because 1) the initial approach complicated the `unify` method considerably, and 2) since the objects `Constant` and `Variable` did not appear in proof trees, we had to treat these as exceptions, complicating both search and construction of proof trees. Note also that `SimpleSentence` implements both interfaces. This is an example of how Java uses interfaces to achieve a form of multiple inheritance.

Another important aspect of this design is the introduction of the `AbstractOperator` class. As indicated in the model of Figure 24.6, an `AbstractOperator` is the parent class of all logical operators. This abstract class defines the basic handling of the arguments of operators through the methods `firstOperand`, `tailOperands`, and `isEmpty`. These methods will enable a recursive search to find solutions to the different operands.

To complete our logical representation language, we need to define Horn Clause rules. Rules do not correspond directly to nodes in an and/or graph; rather, they define relationships between nodes. Consequently, the `Rule` class will be a direct descendant of `PCExpression, as shown in` Figure 24.7, where a rule is a Horn Clause, taking a `SimpleSentence` as its conclusion, or *head*, and a `Goal` as its premise, or *tail*.

This completes the classes defining our logic-based language. The next section gives their implementation, which is fairly straightforward, and Section 24.5 adds new classes for searching the and/or graph defined by inferences on these expressions. This decision to define separate classes for the representation and search reflects common AI programming practice.

24.4 Implementing Complex Logic Expressions

Implementing complex expressions starts with the `Goal` interface. Although Section 24.5 adds a method to this definition, for now, it is a methodless `interface`:

```
public interface Goal extends PCExpression {}
```

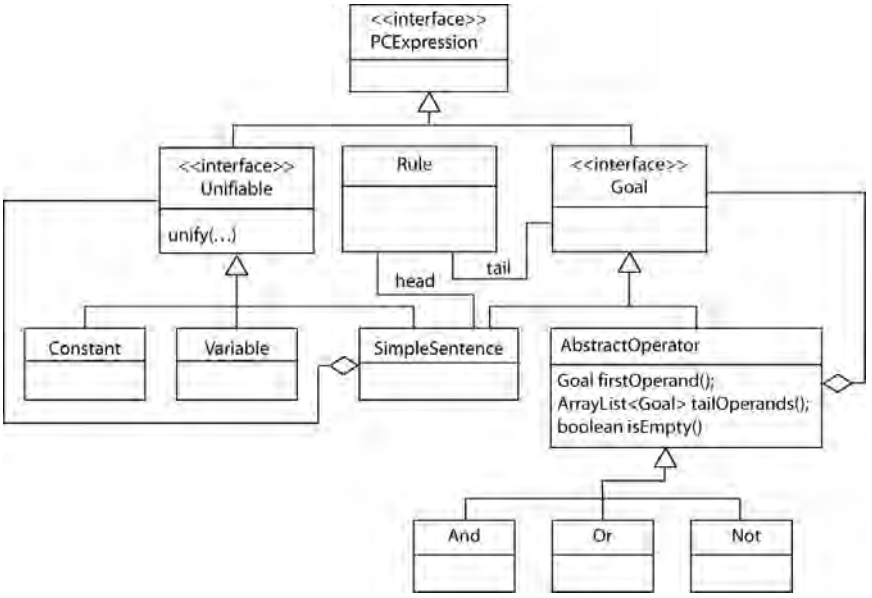

Figure 24.7. A Horn clause `Rule` representation as an instance of `PCExpression`.

Later, we modify `SimpleSentence` to implement this interface, but first, we define a new class, called `AbstractOperator`, that defines the basic methods for accessing the arguments of n-ary operators. In keeping with common Java practice, we implement several patterns for accessing operators, including retrieval of operands by number using the methods `operandCount()` and `getOperand(int i)`. Since we also want to support recursive algorithms for manipulating operands, we implement a *head/tail* approach similar to the `car/cdr` pattern widely used in Lisp. We do this through the methods `firstOperand()`, `getOperatorTail()`, and `isEmpty()`. We also define the `replaceVariables()` method required of all `PCExpressions`, taking advantage of the class' general representation of operands.

Implementation of these methods is straightforward, and we do not discuss it other than to present the code:

```
public abstract class AbstractOperator
        implements Goal, Cloneable
{
    protected ArrayList<Goal> operands;
    public AbstractOperator(Goal... operands)
    {
        Goal[] operandArray = operands;
        this.operands = new ArrayList<Goal>();
        for(int i = 0; i < operandArray.length;i++)
        {
            this.operands.add(operandArray[i]);
        }
    }
```

```
public AbstractOperator(ArrayList<Goal>
        operands)
{
    this.operands = operands;
}
public void setOperands(ArrayList<Goal>
        operands)
{
    this.operands = operands;
}
public int operandCount()
{
    return operands.size();
}
public Goal getOperand(int i)
{
    return operands.get(i);
}
public Goal getFirstOperand()
{
    return operands.get(0);
}
public AbstractOperator getOperatorTail()
        throws CloneNotSupportedException
{
    ArrayList<Goal> tail = new
        ArrayList<Goal>(operands);
    tail.remove(0);
    AbstractOperator tailOperator =
        (AbstractOperator)this.clone();
    tailOperator.setOperands(tail);
    return tailOperator;
}
public boolean isEmpty()
{
 return operands.isEmpty();
}
public PCExpression
        replaceVariables(SubstitutionSet s)
        throws CloneNotSupportedException
```

```
        {
            ArrayList<Goal> newOperands =
                new ArrayList<Goal>();
            for(int i = 0; i < operandCount(); i++)
                newOperands.add((Goal)
                    getOperand(i).
                        replaceVariables(s));
            AbstractOperator copy =
                (AbstractOperator) this.clone();
            copy.setOperands(newOperands);
            return copy;
        }
    }
```

The `And` operator is a simple extension to this class. At this time, our implementation includes just the `toString()` method. Note use of the accessors defined in `AbstractOperator()`:

```
public class And extends AbstractOperator
{
    public And(Goal... operands)
    {
        super(operands);
    }
    public And(ArrayList<Goal> operands)
    {
        super(operands);
    }
    public String toString()
    {
        String result = new String("(AND ");
        for(int i = 0; i < operandCount(); i++)
            result = result +
                getOperand(i).toString();
            return result;
    }
}
```

We leave implementation of `Or` and `Not` as exercises.

Finally, we implement `Rule` as a Horn Clause, having a `SimpleSentence` as its conclusion, or *head*, and any `Goal` as its premise, or *tail*. At this time, we provide another basic implementation, consisting of accessor methods and the `replaceVariables()` method required for all classes implementing `PCExpression`. Also, we allow `Rule` to have a head only (i.e., body = null), as follows from the

definition of Horn Clauses. These rules correspond to simple assertions, such as `likes(george, kate)`.

```
public class Rule implements PCExpression
{
    private SimpleSentence head;
    private Goal body;
    public Rule(SimpleSentence head)
    {
        this(head, null);
    }
    public Rule(SimpleSentence head, Goal body)
    {
        this.head = head;
        this.body = body;
    }
    public SimpleSentence getHead()
    {
            return head;
    }
    public Goal getBody()
    {
            return body;
    }
    public PCExpression
            replaceVariables(SubstitutionSet s)
            throws CloneNotSupportedException
    {
        ArrayList<Goal> newOperands =
            new ArrayList<Goal>();
        for(int i = 0; i < operandCount(); i++)
        newOperands.add((Goal)getOperand(i).
                replaceVariables(s));
        AbstractOperator copy =
            (AbstractOperator)this.clone();
        copy.setOperands(newOperands);
        return copy;
    }
    public String toString()
    {
     if (body == null)
       return head.toString();
```

```
            return head + " :- " + body;
        }
    }
```

Up to this point, the implementation of complex expressions has been straightforward, focusing on operators for manipulating their component structures. This is because their more complex semantics is a consequence of how they are interpreted in problem solvers. The next section discusses the design of the logic-based reasoning engine that supports this interpretation.

24.5 Logic-Based Reasoning as And/Or Graph Search

The basis of our implementation of and/or graph search is a set of classes for defining nodes of the graph. These will correspond to simple sentences, and the operators `And`, `Or`, and `Not`. In this section we define nodes for `And` with simple sentences, leaving `Or` and `Not` as exercises. Our approach is to construct an and/or graph as we search. When the search terminates in success, this graph will be the *proof tree* for that solution. If additional solutions are desired, a call to a `nextSolution()` method causes the most recent subgoal to fail, resuming the search at that point. If there are no further solutions from that subgoal, the search will continue to "fail back" to a parent goal, and continue searching. The implementation will repeat this backtracking search until the space is exhausted.

Figure 24.8 illustrates this search. At the top of the figure we begin with an initial and/or graph consisting only of the initial goal (e.g., `friends(george, X)`). A call to the method `nextSolution()` starts a search of the graph and constructs the proof tree, stopping the algorithm. In addition to constructing the proof tree, each node stores its state at the time the search finished, so a second call to `nextSolution()` causes the search to resume where it left off.

This technique is made possible by a programming pattern known as *continuations*. Continuations have had multiple uses but the main idea is that they allow the programmer to save the program execution at any instant (state) in time so that it can be re-started from that point sometime in the future. In languages that support continuations directly, this is usually implemented by saving the program stack and program counter at the point where the program is frozen. Java does not support this pattern directly, so we will implement a simplified form of the continuation pattern using object member variables to save a reference to the current goal, the current rule used to solve it, and the current set of variable bindings in the tree search. Figure 24.9 shows the classes we introduce to implement this approach.

`AbstractSolutionNode` defines the basic functionality for every node of the graph, including the abstract method `nextSolution()`. `AbstractSolutionNode` and its descendants will implement the ability to search the and/or graph, to save the state of the search, and to resume on subsequent calls to `nextSolution()`.

The class `RuleSet` maintains the logic base, a list of rules. The intention is that all nodes will use the same instance of `RuleSet`, with each instance of `AbstractSolutionNode` maintaining a reference to a particular rule in the set to enable the continuation pattern.

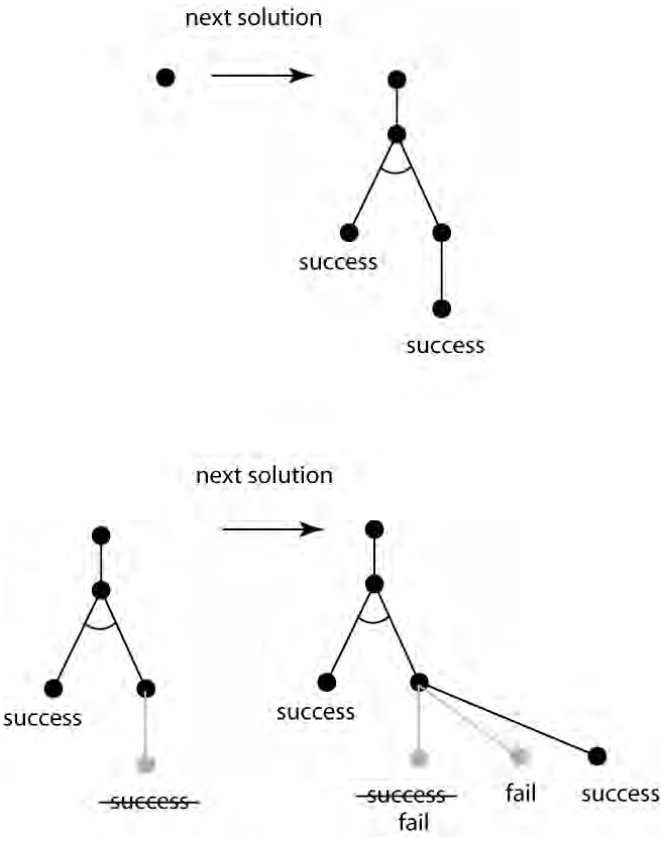

Figure 24.8. An example search space and construction of the proof tree.

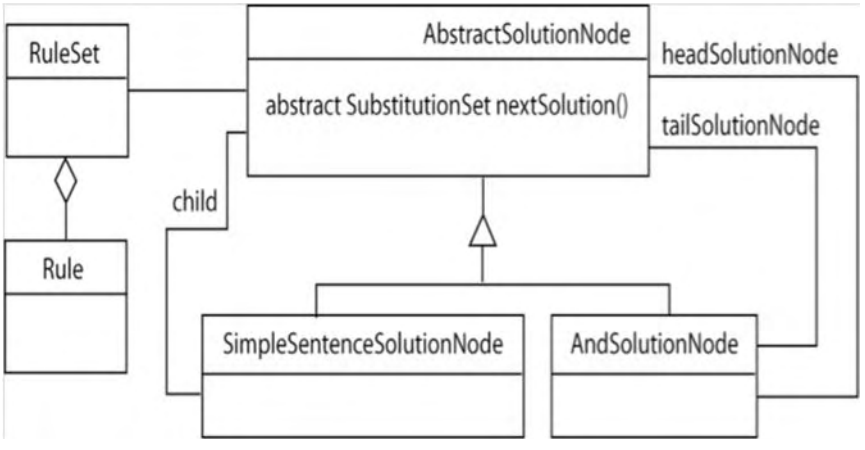

Figure 24.9. The class structure for implementing continuations.

The descendants of `AbstractSolutionNode` maintain references to their children. `SimpleSentenceSolutionNode` represents a simple sentence as a goal, and maintains a reference to its child: the head of a rule. `AndSolutionNode` represents an `and` node, and keeps a

reference to the first branch of the `and` node (the relationship labeled `headSolutionNode`) and the subsequent branches in the `and` node (the relationship labeled `tailSolutionNode`).

We begin implementation with the `RuleSet` class:

```
public class RuleSet
{
    private Rule[] rules;
    public RuleSet(Rule... rules)
    {
        this.rules = rules;
    }
    public Rule getRuleStandardizedApart(int i)
    {
        Rule rule =
            (Rule)rules[i].
                standardizeVariablesApart(
                    new Hashtable<Variable,
                        Variable>());
        return rule;
    }
    public Rule getRule(int i)
    {
        return rules[i];
    }
    public int getRuleCount()
    {
        return rules.length;
    }
}
```

This definition is simple: it maintains an array of rules and allows them to be retrieved by number. The only unusual element is the method `getRuleStandardizedApart(int i)`. This is necessary because the scope of logical variables is the single predicate sentence containing it in a single reasoning step. If we use the same rule again in the search, which is fairly common, we will need to assign new bindings to the variables. A simple way to insure this is to replace the variables in the rule with new copies having the same name. This operation, called "standardizing variables apart" must be defined for all expressions in the rule set. To support this, we will add a new method signature to the interface `PCExpression`. This interface now becomes:

```
public interface PCExpression
{
    public PCExpression
            standardizeVariablesApart(
```

```
                Hashtable<Variable, Variable> newVars);
    public PCExpression
                replaceVariables(SubstitutionSet s);
}
```

The intention here is that the method will be recursive, with each type of `PCExpression` giving it its own appropriate definition. In the method signature, the hash table of pairs of variables keeps track of the substitutions made so far, since a variable may occur multiple times in an expression, and will need to use the same replacement. Defining this requires changes to the following classes. `AbstractOperator` will define it for all n-ary operators:

```
public abstract class AbstractOperator implements
Goal, Cloneable
{
    // variables and methods as already defined
    public PCExpression
                standardizeVariablesApart(
                    Hashtable<Variable,
                    Variable>wVars)
                throws CloneNotSupportedException
    {
        ArrayList<Goal> newOperands =
                new ArrayList<Goal>();
        for(int i = 0; i < operandCount(); i++)
                newOperands.add((Goal)getOperand(i).
                standardizeVariablesApart(newVars));
        AbstractOperator copy =
                (AbstractOperator) this.clone();
        copy.setOperands(newOperands);
        return copy;
    }
}
```

We will also define the method for existing classes `SimpleSentence`, `Constant`, and `Variable`. The definition for Constant is straightforward: each constant returns itself.

```
 public class Constant implements Unifiable
{
    // variables and methods as previously defined
    public PCExpression
        standardizeVariablesApart(
                Hashtable<Variable, Variable> newVars)
    {
         return this;
    }
}
```

The definition for `Variable` is also straightforward, and makes use of the copy constructor defined earlier.

```
public class Variable implements Unifiable
{
        // variables and methods already defined.
    public PCExpression standardizeVariablesApart(
            Hashtable<Variable, Variable>
                newVars)
    {
        Variable newVar = newVars.get(this);
                // Check if the expression already has
                    // a substitute variable.
        if(newVar == null)      // if not create one.
        {
            newVar = new Variable(this);
            newVars.put(this, newVar);
        }
        return newVar;
    }
```

SimpleSentence defines the method recursively:

```
public class SimpleSentence
        implements Unifiable, Goal, Cloneable
{
        // variables and methods already defined.
    public PCExpression
            standardizeVariablesApart(
                Hashtable<Variable, Variable>
                    newVars)
            throws CloneNotSupportedException
    {
        Unifiable[] newTerms =
            new Unifiable[terms.length];
                  //create an array for new terms.
        for(int i = 0; i < length(); i++){
            newTerms[i] =
                (Unifiable)terms[i].
                    standardizeVariablesApart(
                        newVars);
                    // Standardize apart each term.
                    // Only variables will be affected.
        }
```

```
            SimpleSentence newSentence =
                  (SimpleSentence) clone();
            newSentence.setTerms(newTerms);
            return newSentence;
        }
```

Once `RuleSet` has been defined, the implementation of `AbstractSolutionNode` is, again, fairly straightforward.

```
public abstract class AbstractSolutionNode
{
    private RuleSet rules;
    private Rule currentRule = null;
    private Goal goal= null;
    private SubstitutionSet parentSolution;
    private int ruleNumber = 0;
    public AbstractSolutionNode(Goal goal,
            RuleSet rules,
            SubstitutionSet parentSolution)
    {
        this.rules = rules;
        this.parentSolution = parentSolution;
        this.goal = goal;
    }
    public abstract SubstitutionSet nextSolution()
            throws CloneNotSupportedException;
    protected void reset(SubstitutionSet
            newParentSolution)
    {
        parentSolution = newParentSolution;
        ruleNumber = 0;
    }
    public Rule nextRule() throws
            CloneNotSupportedException
    {
        if(hasNextRule())
            currentRule =
                rules.getRuleStandardizedApart(
                      ruleNumber++);
        else
            currentRule = null;
        return currentRule; }
    protected boolean hasNextRule()
    {
        return ruleNumber < rules.getRuleCount();
    }
```

```
    protected SubstitutionSet getParentSolution()
    {
        return parentSolution;
    }
    protected RuleSet getRuleSet()
    {
        return rules;
    }
    public Rule getCurrentRule()
    {
        return currentRule;
    }
    public Goal getGoal()
    {
        return goal;
    }
}
```

The member variable `rules` holds the rule set shared by all nodes in the graph. `RuleNumber` indicates the rule currently being used to solve the goal. `ParentSolution` is the substitution set as it was when the node was created; saving it allows backtracking on resuming the continuation of the search. These three member variables allow the node to resume search where it left off, as required for the continuation pattern.

The variable `goal` stores the goal being solved at the node, and `currentRule` is the rule that defined the current state of the node. `Reset()` allows us to set a solution node to a state equivalent to a newly created node. `NextRule()` returns the next rule in the set, with variables standardized apart. The definition also includes the signature for the `nextSolution()` method. The remaining methods are simple accessors.

The next class we define is `SimpleSentenceSolutionNode`, an extension of `AbstractSolutionNode` for simple sentences.

```
public class SimpleSentenceSolutionNode extends
          AbstractSolutionNode
{
    private SimpleSentence goal;
    private AbstractSolutionNode child = null;
    public SimpleSentenceSolutionNode(
                SimpleSentence goal,
                RuleSet rules,
                SubstitutionSet parentSolution)
            throws CloneNotSupportedException
    {
        super(goal, rules, parentSolution);
    }
```

```
        public SubstitutionSet nextSolution()
        {
             SubstitutionSet solution;
             if(child != null)
             {
                  solution = child.nextSolution();
                  if (solution != null)
                       return solution;
             }
             child = null;
             Rule rule;
             while(hasNextRule() == true)
             {
                  rule = nextRule();
                  SimpleSentence head = rule.getHead();
                  solution = goal.unify(head,
                       getParentSolution());
                  if(solution != null)
                  {
                       Goal tail = rule.getBody();
                       if(tail == null)
                            return solution;
                       child = tail.getSolver
                                 (getRuleSet(),solution);
                       SubstitutionSet childSolution =
                                  child.nextSolution();
                       if(childSolution != null)
                            return childSolution;
                  }
             }
             return null;
        }
        public AbstractSolutionNode getChild()
        {
             return child;
        }
   }
```

This class has one member variable: `child` is the next node, or subgoal in the state space. The method `nextSolution()` defines the use of

these variables, and is one of the more complex methods in the implementation, and we will list the steps in detail.

1. The first step in `nextSolution()` is to test if `child` is `null`. If it is not, which could be the case if we are resuming a previous search, we call `nextSolution()` on the child node to see if there are any more solutions in that branch of the space. If this returns a non-null result, the method returns this solution.
2. If the child node returns no solution, the method sets `child` to `null`, and resumes trying rules in a while-loop. The loop gets each rule from the `RuleSet` in turn, and attempts to unify the goal with the rule head.
3. If the goal matches a rule head, the method then checks if the rule has a tail, or premise. If there is no tail, then this match represents a solution to the goal and returns that substitution set.
4. If the rule does have a tail, the method calls `getSolver()` on the rule tail to get a new child node. This is a new method, which we will discuss shortly.
5. Finally, the method calls `nextSolution()` on the new child node, returning this solution if there is one, and continuing the search otherwise.
6. If the while-loop exhausts the rule set, the node returns `null`, indicating there are no further solutions.

We have not discussed the method `getSolver()`mentioned in step #4. This is a new method for all classes implementing the `Goal` interface that returns the type of solution node appropriate to that goal. By letting each goal determine the proper type of solver for it, we can implement `nextSolution()` in general terms. The revised definition of `Goal`:

```
public interface Goal extends PCExpression
        throws CloneNotSupportedException
{
    public AbstractSolutionNode getSolver(
            RuleSet rules,
            SubstitutionSet parentSolution);
}
```

To complete the search implementation, we define the class, `AndSolutionNode`. Our approach to this implementation is to define a new `And` node for each argument to the `And` operator and the remaining operators. Figure 24.10 illustrates this approach. At the top of the figure is a portion of an And/Or graph for the goal `p ∧ q ∧ r ∧ s`, indicating that the top-level goal will be satisfied by a set of variable substitutions that understands all four of its child goals.

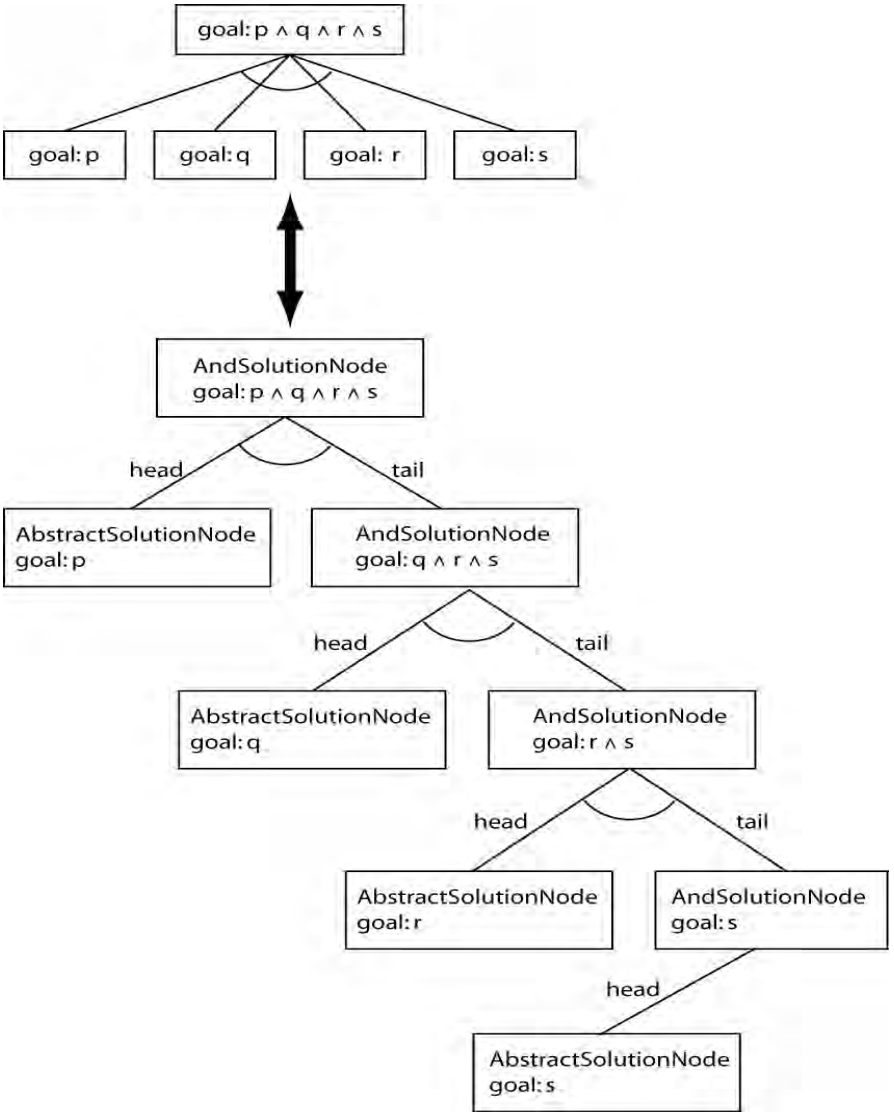

Figure 24.10 A conjunctive goal (top) and the search tree used for its solution.

The bottom of Figure 24.10 indicates the approach we will take. Instead of allowing multiple children at an `and` node, we will make each node binary, consisting of the `and` of the solution for the first operand (the head) and the subsequent operands (the tail). This supports a recursive algorithm that simplifies our code. We leave it to the student to demonstrate (preferably through a formal proof) that the two approaches are equivalent. An additional exercise to implement `and` nodes by using an iterator across a list of child nodes.

`AndSolutionNode` follows the structure of Figure 24.10:

```
public class AndSolutionNode extends
        AbstractSolutionNode
{
    private AbstractSolutionNode
        headSolutionNode = null;
```

```
private AbstractSolutionNode
     tailSolutionNode = null;
private AbstractOperator operatorTail = null;
public AndSolutionNode(And goal,
                RuleSet rules,
                SubstitutionSet parentSolution)
          throws CloneNotSupportedException
{
     super(goal, rules, parentSolution);
     headSolutionNode =
               goal.getFirstOperand().
     getSolver(rules, parentSolution);
     operatorTail = goal.getOperatorTail();
}
protected AbstractSolutionNode
     getHeadSolutionNode()
{
     return headSolutionNode;
}
protected AbstractSolutionNode
     getTailSolutionNode()
{
     return tailSolutionNode;
}
public SubstitutionSet nextSolution()
     throws CloneNotSupportedException
{
     SubstitutionSet solution;
     if(tailSolutionNode != null)
     {
          solution =
               tailSolutionNode.nextSolution();
          if(solution != null) return solution;
     }
     while(solution =
               headSolutionNode.nextSolution())
               != null)
     {
          if(operatorTail.isEmpty())
               return solution;
          else
          {
               tailSolutionNode =
                        operatorTail.getSolver(
                        getRuleSet(), solution);
```

```
                    SubstitutionSet tailSolution =
                              tailSolutionNode.
                              nextSolution();
                    if(tailSolution != null)
                         return tailSolution;
               }
          }
          return null;
     }
}
```

The constructor creates a solution node, `headSolutionNode`, for the first argument of the `And` operator, and also sets the member variable, `operatorTail`, for the rest of the arguments if they exist. Note that it does not create a solution node for the tail at this time. This is an efficiency concern: if there are no solutions to the head subgoal, the entire `and` operator will fail, and there is no need to try the rest of the operators.

As with `SimpleSolutionNode`, the `nextSolution()` method implements the search and the supporting continuation pattern. It begins by testing if `tailSolutionNode` is non-null. This is true only if there are remaining arguments `(operatorTail != null)`, and we have found at least one solution to the head goal. In this case, the continuation must first check to see if there are additional solutions to the tail goal.

When this fails, the algorithm enters a loop of testing for further solutions to the head goal. When it finds a new solution to the head, it checks if there is a tail goal; if not, it returns the solution. If there is a tail goal, it will acquire the child node, a subclass of `AbstractSolutionNode` using the `getSolver` method, and then tries for a solution to the tail goal.

This completes the implementation of the search framework for the `And` operator. We leave implementation of `Or` and `Not` to the reader.

24.6 Testing the Reasoning System

Below is a simple `Tester` class for the reasoning system. It uses a recursive rule for reasoning about ancestor relationships. This is a simple test harness and is not suitable for end users. Finishing the reasoner would involve allowing the representation of rules in a more friendly syntax, such as Prolog, and an interactive query engine. We leave this as an exercise. We also encourage the reader to modify this simple `Tester` to further explore the code.

```
public class Tester
{
    public static void main(String[] args)
    {                              //Set up the knowledge base.
         Constant parent = new Constant("parent"),
              bill = new Constant("Bill"),
              audrey = new Constant("Audrey"),
              maria = new Constant("Maria"),
```

```
        tony = new Constant("Tony"),
        charles = new Constant("Charles"),
        ancestor = new Constant("ancestor");
    Variable X = new Variable("X"),
        Y = new Variable("Y"),
        Z = new Variable("Z");
    RuleSet rules = new RuleSet(
        new Rule(new SimpleSentence(parent,
                bill, audrey)),
        new Rule(new SimpleSentence(parent,
                maria, bill)),
        new Rule(new SimpleSentence(parent,
                tony, maria)),
        new Rule(new SimpleSentence(parent,
                charles, tony)),
        new Rule(new SimpleSentence(ancestor,
                X, Y),
        new And(new SimpleSentence(parent,
                X, Y))),
        new Rule(new SimpleSentence(ancestor,
                X, Y),
        new And(new SimpleSentence(parent,
                X, Z),
        new SimpleSentence(ancestor, Z, Y))));
       // define goal and root of search space.
    SimpleSentence goal =
                new SimpleSentence(ancestor,
                charles, Y);
    AbstractSolutionNode root =
                goal.getSolver(rules,
                new SubstitutionSet());
    SubstitutionSet solution;
                        // print out results.
    System.out.println("Goal = " + goal);
    System.out.println("Solutions:");
    try
    {
        while((solution = root.nextSolution())
                 != null)
        {
            System.out.println("     " + goal.
                    replaceVariables(
                    solution));
        }
    }
    catch (CloneNotSupportedException e)
```

```
            {
                System.out.println(
                    "CloneNotSupportedException:" + e);
            }
        }
    }
```

24.7 Design Discussion

In closing out this chapter, we would like to look at two major design decisions. The first is our separation of representation and search through the introduction of `AbstractSolutionNode` and its descendants. The second is the importance of static structure to the design.

Separating Representation and Search

The separation of representation and search is a common theme in AI programming. In Chapter 22, for example, our implementation of simple search engines relied upon this separation for generality. In the reasoning engine, we bring the relationship between representation and search into sharper focus. Here, the search engine serves to define the semantics of our logical representation by implementing a form of logical inference. As we mentioned before, our approach builds upon the mathematics of the representation language – in this case, theories of logic inference – to insure the quality of our representation.

One detail of our approach bears further discussion. That is the use of the method, `getSolver(RuleSet rules, SubstitutionSet parentSolution)`, which was defined in the `Goal` interface. This method simplifies the handling of the search space by letting search algorithms treat them independently of their type (simple sentence, node, etc). Instead, it lets us treat nodes in terms of the general methods defined by `AbstractSolutionNode`, and to rely upon each goal to return the proper type of solution node. This approach is beneficial, but as is typical of object-oriented design, there are other ways to implement it.

One of these alternatives is through a *factory* pattern. This would replace the `getSolver()` method of `Goal` with a separate class that creates instances of the needed node. For example:

```
Class SolutionNodeFactory
{
    public static AbstractSolutionNode
        getSolver(Goal goal,
                  RuleSet rules,
                  SubstitutionSet parentSolution)
    {
        if (goal instanceof SimpleSentence)
            return new SimpleSentenceSolutionNode(
                      goal, rules, parentSolution);
        if (goal instanceof And)
            return new AndSolutionNode(goal, rules,
                      parentSolution);
    }
}
```

There are several interesting trade-offs between the approaches. Use of the `Factory` sharpens the separation of representation and search. It even allows us to reuse the representation in contexts that do not involve reasoning without the difficulty of deciding how to handle the `getSolver` method required by the parent interface. On the other hand, the approach we did use allows us to get the desired solver without using `instanceof` to test the type of goal objects explicitly. Because the `instanceof` operator is computationally expensive, many programmers consider it good style to avoid it. Also, when adding a new operator, such as `Or`, we only have to change the operator's class definition, rather than adding the new class and modifying the `Factory` object. Both approaches, however, are good Java style. As with all design decisions, we encourage the reader to evaluate these and other approaches and make up their own mind.

The Importance of Static Structure

A more important design decision concerns the *static* structure of the implementation. By static structure, we mean the organization of classes in a program. We call it static because this structure is not changed by program execution. As shown in Figures 24.6, 24.7, and 24.9, our approach has a fairly complex static structure. Indeed, in developing the reasoner, we experimented with several different approaches (this is, we feel, another good design practice), and many of these had considerably fewer classes and simpler static structures. We chose this approach because it is usually better to represent as much of the program's semantic structure as is feasible in the class structure of the code. There are several reasons for this:

1. It makes the code easier to understand. Although our static structure is complex, it is still much simpler than the dynamic behavior of even a moderately complex program. Because it is static, we can make good use of modeling techniques and tools to understand the program, rather than relying on dynamic tracing to see what is going on in program executions.
2. It simplifies methods. A well-designed static structure, although it may be complex, does not necessarily add complexity to the code. Rather, it moves complexity from methods to the class structure. Instead of a few classes with large complex methods, we tend to have more, simpler methods. If we look at the implementation of our logic-based reasoner, the majority of the methods were surprisingly simple: mostly setting or retrieving values from a data structure. This makes methods easier to write correctly, and easier to debug.
3. It makes it easier to modify the code. As any experienced programmer has learned, the lifecycle of useful code inevitably involves enhancements. There is a tendency for these enhancements to complicate the code, leading to increased problems with bugs as the software ages. This phenomenon has been called software entropy. Because it breaks the program functionality down into many smaller methods spread among many classes, good static structure can simplify

code maintenance by reducing the need to make complex changes to existing methods.

This chapter completes the basic implementation of a logic-based reasoner, except for certain extensions including adding the operators for or and not. We leave these as an exercise. The next chapter will add a number of enhancements to the basic reasoner, such as asking users for input during the reasoning process, or replacing true/false values with quantitative measures of uncertainty. As we develop these enhancements, keep in mind how class structure supports these extensions, as well as the implementation patterns we use to construct them.

Exercises

1. Write a method of `AbstractSolutionNode` to print out a proof tree in a readable format. A common approach to this is to indent each node's description c * level, where level is its depth in the tree, and c is the number of spaces each level is indented.

2. Add classes for the logical operators `Or` and `Not`. Try following the pattern of the chapter's implementation of `And`, but do so critically. If you find an alternative approach you prefer, feel free to explore it, rewriting `Or` and `Not` as well. If you do decide on a different approach, explain why.

3. Extend the "user-friendly" input language from exercise 8 of chapter 22 to include `And, ∧`, `Or, ∨`, `Not, ¬`, and `Rule, ←`.

4. Write a Prolog-style interactive front end to the logical reasoner that will read in a logical knowledge-base from a file using the language of exercise 2, and then enter a loop where users enter goals in the same language, printing out the results, and then prompting for another goal.

5. Implement a factory pattern for generating `solutionNodes`, and compare it to the approach taken in the chapter. A factory would be a class, named `solutionNodeFactory` with a methods that would take any needed variables and return an instance of the class `solutionNodes`.

6. Give a logical proof that the two approaches to representing And nodes in Figure 24.10 are equivalent.

7. Modify the `nextSolution()` method in `AndSolutionNode` to replace the recursive implementation with one that iterates across all the operators of an And operator. Discuss the trade-offs between efficiency, understandability, and maintainability in the two approaches.

25 An Expert System Shell

Chapter Objectives

Completing the meta-interpreter for rule systems in Java
Full backtracking unification algorithm
- A goal-based reasoning shell
- An example rule system demonstration

The extended functionality for building expert systems
- Askable predicates
- Response to how and why queries
- Structure presented for addition of certainty factors

Chapter Contents

25.1 Introduction: Expert Systems

In Chapter 24, we developed a unification-based logic problem solver that solved queries through a depth-first, backward chaining search. In this chapter, we will extend those classes to implement two features commonly found in expert-system shells: the ability to attach confidence estimates, or certainty factors, to inferences (see Luger 2009 for more on certainty factors), and the ability to interact with the user during the reasoning process. Since all the classes in this chapter will extend classes from the unification problem solver, readers must be sure to have read that chapter before continuing.

In developing the expert system shell, we have two goals. The first is to explore the use of simple inheritance to extend an existing body of code. The second is to provide the reader with a start on more extensive modifications to the code that will be a valuable learning experience; the exercises will make several suggestions for such extensions.

Certainty Factors

The first extension to the reasoner will be to implement a simplified version of the certainty factor algebra described in Luger (2009). Certainty factors will be numbers between -1.0 and 1.0 that measure our confidence in an inference: -1.0 indicates the conclusion is false with maximum certainty, and 1.0 means it is true with maximum certainty. A certainty value of 0.0 indicates nothing is known about the assertion. Values between -1.0 and 1.0 indicate varying degrees of confidence.

Rules have an attached certainty factor, which indicates the certainty of their conclusion if all elements in the premise are known with complete certainty. Consider the following rule and corresponding certainty factor:

```
If p then q, CF = 0.5
```

This means that, if `p` is true with a confidence of 1.0 (maximum confidence), then `q` can be inferred to be true with a confidence of 0.5. This is the measure of the uncertainty introduced by the rule itself. If our confidence in `p` is less, than our confidence in `q` will be lowered accordingly.

In the case of the conjunction, or "and," of two expressions, we compute the certainty of the conjunction as the minimum of the certainty of the operands. Note that if we limit certainty values to 1.0 (true) and -1.0 (false), this reduces to the standard definition of "and." For the "or" operation, the certainty of the expressions is the maximum of the certainty of its individual operands. The "not" operator switches the sign of the certainty factor of its argument. These are also intuitive extensions of the boolean meaning of those operators.

Certainty factors propagate upward through the inference chain: given a rule, we unify the rule premises with matching subgoals. After inferring the certainties of the individual subgoals, we compute the certainty of the entire rule premise according to the operators for `and`, `or`, and `not`. Finally, we multiply the certainty of the premise by the certainty of the rule to compute the certainty of the rule conclusion.

Generally, certainty factor implementations will prune a line of reasoning if the certainty value falls below a certain value. A common pruning value is if the certainty is less than 0.2. This can eliminate many branches of the search space. We will not include this in the implementation of this chapter, but will leave it as an exercise.

25.2 Certainty Factors and the Unification Problem Solver

Our basic design strategy will be to make minimal changes to the representation of expressions, and to make most of our changes to the nodes of the solution tree. The reasoning behind this approach is the idea that the nodes of the solution tree define the inference strategy, whereas logical expressions simply are a statement about the world that is independent of its truth or reasoning. As a variation on truth-values, it follows that we should treat certainty calculations as a part of the system's inference strategy, implementing them as extensions to descendents of the class `AbstractSolutionNode`. This suggests we take `SimpleSentence` and basic operators to represent assertions independently of their certainty, and avoid changing them to support this new reasoning strategy.

The classes we will define will be in a new package called `expertSystemShell`. To make development of the expert system shell easier to follow, we will name classes in this package by adding the prefix "ES" to their ancestors in the package `unificationSolver` defined in the previous chapter.

Adding Certainty Factors to Expressions

We will support representation of certainty factors as an extension to the definition of `Rule` from the unification problem solver. We will define a new subclass of `Rule` to attach a certainty factor to the basic representation. We define `ESRule` as a straightforward extension of the Rule class by adding a private variable for certainty values, along with

standard accessors:

```
public class ESRule extends Rule
{
    private double certaintyFactor;
    public ESRule(ESSimpleSentence head,
             double certaintyFactor)
    {
         this(head, null, certaintyFactor);
    }
    public ESRule(ESSimpleSentence head, Goal body,
             double certaintyFactor)
    {
         super(head, body);
         this.certaintyFactor = certaintyFactor;
    }
    public double getCertaintyFactor()
    {
         return certaintyFactor;
    }
    protected void setCertaintyFactor(double value)
    {
         this.certaintyFactor = value;
    }
}
```

Note the two constructors, both of which include certainty factors in their arguments. The first constructor supports rules with conclusions only; since a fact is simply a rule without a premise, this allows us to add certainty factors to facts. The second constructor allows definition of full rules. An obvious extension to this definition would be to add checks to make sure certainty factors stay in the range -1.0 to 1.0, throwing an out of range exception if they are not in range. We leave this as an exercise.

This is essentially the only change we will make to our representation. Most of our changes will be to the solution nodes in the proof tree, since these define the reasoning strategy. To support this, we will define subclasses to both `SimpleSentence` and `And` to return the appropriate type of solution node, as required by the interface `Goal` (these are all defined in the preceding chapter). The new classes are:

```
public class ESSimpleSentence extends SimpleSentence
{
    public ESSimpleSentence(Constant functor,
             Unifiable... args)
    {
         super(functor, args);
    }
```

```
    public AbstractSolutionNode getSolver(RuleSet
            rules, SubstitutionSet parentSolution)
    {
        return new
            ESSimpleSentenceSolutionNode(this,
                (ESRuleSet)rules, parentSolution);
    }
}
public class ESAnd extends And
{
    public ESAnd(Goal... operands)
    {
        super(operands);
    }
    public ESAnd(ArrayList<Goal> operands)
    {
        super(operands);
    }
    public AbstractSolutionNode getSolver(RuleSet
        rules, SubstitutionSet parentSolution)
    {
        return new ESAndSolutionNode(this, rules,
            parentSolution);
    }
}
```

These are the only extensions we will make to the representation classes. Next, we will define reasoning with certainty factors in the classes `ESSimpleSentenceSolutionNode` and `ESAndSolutionNode`.

Reasoning with Certainty Factors

Because the certainty of an expression depends on the inferences that led to it, the certainty factors computed during reasoning will be held in solution nodes of the proof tree, rather than the expressions themselves. Thus, every solution node will define at least a goal, a set of variable substitutions needed to match the goal during reasoning, and the certainty of that conclusion. The first two of these were implemented in the previous chapter in the class `AbstractSolutionNode`, and its descendents. These classes located their reasoning in the method, `nextSolution()`, defined abstractly in `AbstractSolutionNode`.

Our strategy will be to use the definitions of `nextSolution()` from the classes `SimpleSentenceSolutionNode` and `AndSolutionNode` defined in the previous chapter. So, for example, the basic framework of `ESSimpleSentenceSolutionNode` is:

```
public class ESSimpleSentenceSolutionNode
    extends SimpleSentenceSolutionNode
    implements ESSolutionNode
{
    private double certainty = 0.0; //default value
    public ESSimpleSentenceSolutionNode(
         ESSimpleSentence goal, ESRuleSet rules,
         SubstitutionSet parentSolution)
    {
         super(goal, rules, parentSolution);
    }
    public synchronized SubstitutionSet
         nextSolution()
         throws CloneNotSupportedException
    {
         SubstitutionSet solution =
              super.nextSolution();
         // Compute certainty factor for the solution
         // (see below)
         return solution;
    }
    public double getCertainty()
    {
         return certainty;
    }
 }
```

This schema, which will be the same for the `ESAndSolutionNode`, defines `ESSimpleSentenceSolutionNode` as a subclass of the `SimpleSentenceSolutionNode`, adding a member variable for the certainty associated with the current goal and substitution set. When finding the next solution for the goal, it will call `nextSolution()` on the parent class, and then compute the associated certainty factor.

The justification for this approach is that the unification problem solver of chapter 24 will find all valid solutions (i.e. sets of variable substitutions) to a goal through unification search. Adding certainty factors does not lead to new substitution sets – it only adds further qualifications on our confidence in those inferences. Note that this does lead to questions concerning logical `not`: if the reasoner cannot find a set of substitutions that make a goal true under the unification problem solver, should it fail or succeed with a certainty of -1.0? For this chapter, we are avoiding such semantic questions, but encourage the reader to probe them further.

We complete the definition of `nextSolution()` as follows

```
public synchronized SubstitutionSet nextSolution()
      throws CloneNotSupportedException
{
    SubstitutionSet solution = super.nextSolution();
    if(solution == null)
    {
         certainty = 0.0;
         return null;
    }
    ESRule rule = (ESRule) getCurrentRule();
    ESSolutionNode child =
         (ESSolutionNode) getChild();
    if(child == null)
    {
         // the rule was a simple fact
         certainty = rule.getCertaintyFactor();
    }
    else
    {
         certainty = child.getCertainty() *
              rule.getCertaintyFactor();
    }
    return solution;
}
```

After calling `super.nextSolution()`, the method checks if the value returned is null, indicating no further solutions were found. If this is the case, it returns null to the parent class, indicating this branch of the search space is exhausted.

If there is a solution, the method gets the current rule which was used to solve the goal, and also gets the child node in the search space. If the child node is null, this indicates a leaf node, and the certainty factor is simply that of the associated rule. Otherwise, the method gets the certainty of the child and multiplies it by the rule's certainty factor. It saves the result in the member variable certainty.

Note that this method is synchronized. This is necessary to prevent a threaded implementation from interrupting the method between computing the solution substitution set, and the associated certainty, as this might cause an inconsistency.

The implementation of the class `ESAndSolutionNode` follows the same pattern, but computes the certainty factor of the node recursively: as the minimum of the certainty of the first operand (the head operand) and the certainty of the rest of the operands (the tail operands).

```
public class ESAndSolutionNode
    extends AndSolutionNode
    implements ESSolutionNode
{
    private double certainty = 0.0;
    public ESAndSolutionNode(ESAnd goal,
         RuleSet rules,
         SubstitutionSet parentSolution)
    {
         super(goal, rules, parentSolution);
    }
    public synchronized SubstitutionSet
         nextSolution()
         throws CloneNotSupportedException
    {
         SubstitutionSet solution =
              super.nextSolution();
         if(solution == null)
         {
              certainty = 0.0;
              return null;
         }
         ESSolutionNode head = (ESSolutionNode)
              getHeadSolutionNode();
         ESSolutionNode tail = (ESSolutionNode)
              getTailSolutionNode();
         if(tail == null)
              certainty = head.getCertainty();
         else
              certainty =
                   Math.min(head.getCertainty(),
                            tail.getCertainty());
         return solution;
    }
    public double getCertainty()
    {
         return certainty;
    }
 }
```

This completes the extension of the unification solver to include certainty factors.

25.3 Adding User Interactions

Another feature common to expert system shells is the ability to ask users about the truth of subgoals as determined by the context of the reasoning. The basic approach to this is to allow certain expressions to be designated as askable. Following the patterns of the earlier sections of this chapter, we will define askables as an extension to an existing class.

Looking at the code defined above, an obvious choice for the base class of askable predicates is the `ESSimpleSentence` class. It makes sense to limit user queries to simple sentences, since asking for the truth of a complex operation would be confusing to users. However, our approach will define `Ask` as a subset of the `Rule` class. There are two reasons for this:

1. In order to query users for the truth of an expression, the system will need to access a user interface. Adding user interfaces to `ESSimpleSentences` not only complicates their definition, but also it complicates the architecture of the expert system shell by closely coupling the interface with knowledge representation classes.
2. So far, our architecture separates knowledge representation syntax from semantics, with syntax being defined in descendents of the `PCExpression` interface, and the semantics being defined in the nodes of the search tree. User queries are a form of inference (may the gods of logic forgive me), and will be handled by them.

As we will see shortly, defining `Ask` as an extension of the `Rule` class better supports these design constraints. Although `Rule` is part of representation, it is closely tied to reasoning algorithms in the solution nodes, and we have already used it to define certainty factors. Our basic scheme will be to modify `ESSimpleSentenceSolutionNode` as follows:

1. If a goal matches the head of a rule, it is true if the premise of the rule is true;
2. If a goal matches the head of a rule with no premise, then it is true;
3. If a goal matches the head of an askable rule, then ask the user if it is true.

Conditions 1 & 2 are already part of the definition of `ESSimpleSentenceSolutionNode`. The remainder of this section will focus on adding #3 to its definition.

Implementing this will require distinguishing if a rule is askable. We will do this by adding a boolean variable to the ESRule class:

```
public class ESRule extends Rule
{
    private double certaintyFactor;
    private boolean ask = false;
                    // constructors and certainty factor
                    // accessors as defined above
```

```
    public boolean ask()
    {
        return ask;
    }
    protected void setAsk(boolean value)
    {
        ask = value;
     }
}
```

This definition sets ask to false as a default. We define the subclass **Ask** as:

```
public class ESAsk extends ESRule
{
    public ESAsk(ESSimpleSentence head)
    {
        super(head, 0.0);
        setAsk(true);
    }
}
```

Note that **ESAsk** has a single constructor, which enforces the constraint that an askable assertion be a simple sentence.

The next step in adding askables to the expert system shell is to modify the method `nextSolution()` of `ESSimpleSentenceSolutionNode` to test for askable predicates and query the user for their certainty value. The new version of `nextSolution()` is:

```
public synchronized SubstitutionSet nextSolution()
  throws CloneNotSupportedException
{
    SubstitutionSet solution = super.nextSolution();
    if(solution == null)
    {
        certainty = 0.0;
        return null;
    }
    ESRule rule = (ESRule) getCurrentRule();
    if(rule.ask())
    {
        ESFrontEnd frontEnd =
            ((ESRuleSet)getRuleSet()).
                getFrontEnd();
        certainty = frontEnd.ask((ESSimpleSentence)
                        rule.getHead(), solution);
        return solution;
    }
```

```
        ESSolutionNode child =
             (ESSolutionNode) getChild();
        if(child == null)
        {
             certainty = rule.getCertaintyFactor();
        }
        else
        {
             certainty = child.getCertainty() *
                          rule.getCertaintyFactor();
        }
        return solution;
    }
```

We will define `ESFrontEnd` in an interface:

```
    public interface ESFrontEnd
    {
        public double ask(ESSimpleSentence goal,
                 SubstitutionSet subs);
    }
```

Finally, we will introduce a new class, `ESRuleSet`, to extend `RuleSet` to include an instance of `ESFrontEnd`:

```
    public class ESRuleSet extends RuleSet
    {
        private ESFrontEnd frontEnd = null;
        public ESRuleSet(ESFrontEnd frontEnd,
             ESRule... rules)
        {
             super((Rule[])rules);
             this.frontEnd = frontEnd;
        }
        public ESFrontEnd getFrontEnd()
        {
             return frontEnd;
        }
    }
```

This is only a partial implementation of user interactions for the expert system shell. We still need to add the ability for users to make a top-level query to the reasoner, and also the ability to handle "how" and "why" queries as discussed in (Luger 2009). We leave these as an exercise.

25.4 Design Discussion

Although the extension of the unification problem solver into a simple expert system shell is, for the most part, straightforward, there are a couple

interesting design questions. The first of these was our decision to, as much as possible, leave the definitions of descendents of `PCExpression` as unchanged as possible, and place most of the new material in extensions to the solution node classes. Our reason for doing this reflects a theoretical consideration.

Logic makes a theoretical distinction between syntax and semantics, between the definition of well-formed expressions and the way they are used in reasoning. Our decision to define the expert system almost entirely through changes to the solution node classes reflects this distinction. In making this decision, we are following a general design heuristic that we have found useful, particularly in AI implementations: insofar as possible, define the class structure of code to reflect the concepts in an underlying mathematical theory. Like most heuristics, the reasons for this are intuitive, and we leave further analysis to the exercises.

The second major design decision is somewhat more problematic. This is our decision to use the nextSolution method from the unification solver to perform the actual search, and compute certainty factors afterwards. The benefits of this are in not modifying code that has already been written and tested, which follows standard object-oriented programming practice.

However, in this case, the standard practice leads to certain cons that should be considered. One of these is that, once a solution is found, acquiring both the variable substitutions and certainty factor requires two separate methods: `nextSolution` and `getCertainty`. This is error prone, since the person using the class must insure that no state changes occur between these calls. One solution is to write a convenience function that bundles both values into a new class (say `ESSolution`) and returns them. A more aggressive approach would be to ignore the current version of nextSolution entirely, and to write a brand new version.

This is a very interesting design decision, and we encourage the reader to try alternative approaches and discuss their trade-offs in the exercises to this chapter.

Exercises

1. Modify the definition of the `nextSolution` method of the classes `ESSimpleSolutionNode` and `ESAndSolutionNode` to fail a line of reasoning if the certainty factor falls below a certain value (0.2 or 0.3 are typical values). Instrument your code to count the number of nodes visited and test it both with and without pruning.

2. Add range checks to all methods and classes that allow certainty factors to be set, throwing an exception of the value is not in the range of -1.0 to 1.0. Either use Java's built-in `IllegalArgumentException` or an exception class of your own definition. Discuss the pros and cons of the approach you choose.

3. In designing the object model for the unification problem solver, we followed the standard AI practice of distinguishing between the representation of well-formed expressions (classes implementing the interface `unifiable`) and the definition of the inference strategy in the

nodes of the solution tree (descendents of `AbstractSolutionNode`). This chapter's expert system shell built on that distinction. More importantly, because we were not changing the basic inference strategy other than to add certainty estimates, we approached the expert system by defining subclasses to `SimpleSentenceSolutionNode` and `AndSolutionNode`, and reusing the existing `nextSolution` method. If, however, we were changing the search strategy drastically, or for other reasons discussed in 25.4, it might have been more efficient to retain only the representation and rewrite the inference strategy completely. As an experiment to explore this option, rewrite the expert system shell without using `AbstractSolutionNode` or any of its descendants. This will give you a clean slate for implementing reasoning strategies. Although this does not make use of previously implemented code, it may allow making the solution simpler, easier to use, and more efficient. Implement an alternative solution, and discuss the trade-offs between this approach and that taken in the chapter.

4. Full implementations of certainty factors also allow the combination of certainty factors when multiple rules lead to the same goal. I.e., if the goal `g` with subsitituions `s` is supported by multiple lines of reasoning, what is its certainty? (Luger 2009) discusses how to compute these values. Implement this approach.

5. A feature common to expert systems that was not implemented in this chapter is the ability to provide explanations of reasoning through How and Why queries. As explained in (Luger 2009), How queries explain a fact by displaying the proof tree that led to it. Why queries explain why a question was asked by displaying the rule that is the current context of the question. Implement How and Why queries in the expert system shell, and support them through a user-friendly front end. This front-end should also allow users to enter queries, inspect rule sets, etc. It should also support askable predicates as discussed in the next exercise.

6. Build a front-end to support user interaction around askable predicates. In particular, it should keep track of answers that have been received, and avoid asking the same question twice. This means it should keep track of both expressions and substitutions that have been asked. An additional feature would be to support asking users for actual substitution values, and adding them to the substitution set.

7. Revisit the design decision to, so far as possible, locate our changes in the solution node classes, rather than descendants of `PCExpression`. In particular, comment on our heuristic of organizing code to reflect the structures implied by logical theory. Did this heuristic of following the structure of theory work well in our implementation? Why? Do you believe this heuristic to be generalizable beyond logic? Once again, why?

26 Case Studies: JESS and other Expert Systems Shells in Java

Chapter Objectives This chapter examines Java expert system shells available on the world wide web

Chapter Contents

26.1 Introduction

In the last three chapters we demonstrated the creation of a simple expert system shell in Java. Chapter 22 presented a representational formalism for describing predicate calculus expressions, the representation of choice for expert rule systems and many other AI problem solvers. Chapter 24 created a procedure for unification. We demonstrated this algorithm with a set of predicate calculus expressions, and then built a simple Prolog in Java interpreter. Chapter 25 added full backtracking to our unification algorithm so that it could check all possible unifications in the processes of finding sets of consistent substitutions across sets of predicate calculus specifications. In Chapter 25 we also created procedures for answering why and how queries, as well as for setting up a certainty factor algebra.

In this chapter we present a number of expert system shell libraries written in Java. As mentioned throughout our presentation of Java, the presence of extensive code libraries is one of the major reasons for the broad acceptance of Java as a problem-solving tool. We have explored these expert shells at the time of writing this chapter. We realize that many of these libraries will change over time and may well differ (or not even exist!) when our readers considers them. So we present their urls, current as of January 2008, with minimal further comment.

26.2 JESS

The first library we present is JESS, the Java Expert System Shell, built and maintained by programmers at Sandia National Laboratories in Albuquerque New Mexico. JESS is a rule engine for the Java platform. Unlike the unification system presented in Chapters 23 and 24, JESS is driven by a lisp-style scripting language built in Java itself. There are advantages and disadvantages to this approach. One main advantage of an independent scripting language is that it is easier to work with for the code builder. For example, Prolog has its own language that is suitable for rule

languages, which makes it easy and clear to write static rule systems.

On the other hand, Prolog is not intended to be embedded in other applications. In the case of Java, rules may be generated and controlled by some external mechanism, and in order to use JESS's approach, the data needs to be converted into text that this interpreter can handle.

A disadvantage of an independent scripting language is the disconnect between Java and the rule engine. Once external files and strings are used to specify rules, standard Java syntax cannot be used to verify and check syntax. While this is not an issue for stand-alone rule solving systems, once the user wants to embed the solver into existing Java environments, she must learn a new language and decide how to interface and adapt the library to her project.

In an attempt to address standardization of rule systems in Java, the Java Community Process defined an API for rule engines in Java. The Java Specification Request #94 defines the javax.rules package and a number of classes for dealing with rule engines. Our impression of this system is that it is very vague and seemingly tailored for JESS. It abstracts the rule system as general objects with general methods for getting/setting properties on rules.

RuleML, although not Java specific, provides a standardized XML format for defining rules. This format can theoretically be used for any rule interpreter, as the information can be converted into the rule interpreter's native representations.

JESS has its own JessML format for defining rules in XML, which can be converted to RuleML and back using XSLT (eXtensible Stylesheet Language Transformations). These formats, unfortunately, are rather verbose and not necessarily intended for being read and written by people.

Web links for using JESS include:

http://www.jessrules.com/- The JESS web site,

http://jcp.org/en/jsr/detail?id=94 - JSR 94: Java™ Rule Engine API,

http://www.jessrules.com/jess/docs/70/api/javax/rules/package-summary.html - javadocs about javax.rules (from JSR 94), and

http://www.ruleml.org/ - RuleML.

26.3 Other Expert System Shells

We have done some research into other Java expert rule systems, and found dozens of them. The following url introduces a number of these (not all in Java):

http://www.kbsc.com/rulebase.html

The general trend of these libraries is to use some form of scripting-language based rule engine. There is even a Prolog implementation in Java! There are many real implementations and issues that these things introduce, including RDF, OWL, SPARQL, Semantic Web, Rete, and more.

Thie following url discusses a high level look at rule engines in Java (albeit

from a couple years ago):

http://today.java.net/pub/a/today/2004/08/19/rulingout.html

Finally, we conclude with a set of links to the seemingly more interesting rule engines. We only picked the engines listed as free, some are open-source, some are not:

http://www.drools.org/

http://www.agfa.com/w3c/euler/

http://jlogic.sourceforge.net/ - a prolog interpreter in Java

http://jlisa.sourceforge.net/ - A Clips-like (NASA rule based shell in C) Rule engine accessible from Java with the power of Common Lisp.

http://mandarax.sourceforge.net/ - this one has some simple straightforward examples on the site, but the javadocs themselves are daunting.

http://tyruba.sourceforge.net/

Related to rule interpreters designed to search knowledge-based specifications, are interpreters intended to transfer knowledge, rules, or general specifications between code modules. These general module translation and integration programs are often described under the topic of the Semantic Web:

http://www.w3.org/2001/SW/

26.4 Using Open Source Tools

The primary advantage these tools have over our simple expert system shell is their range of features. Jess, for example, provides a rule language that frees the programmer from having to declare each rule as a set of nested class instances as in our simple set of tools. An interesting thought experiment would be to consider what it would take to write a parser for a rule language that would construct these class instantiations from a user-friendly rule language. A more ambitious effort, possibly suitable for an advanced undergraduate or masters level thesis project would be to implement such a front end.

In using these tools, the reader should not forget the lessons in extending java classes from the earlier chapter. Inheritance allows the programmer to extend open source code to include additional functionality if necessary. More often, we may simply use these tools as a module in a larger program simply by including the jar files.

In particular, the authors have seen the Jess tool used in a number of large applications at Sandia Laboratories and the University of New Mexico. Typical application architecture uses Jess as an inference engine in a larger system with databases, html front ends using Java Server Faces or similar technologies, and various tools to assist in file I/O, session archiving, etc.

For example, a development team at Sandia Laboratories led by Kevin Stamber, Richard Detry, and Shirley Starks has developed a system called FAIT (Fast Analysis Infrastructure Tool) for use in the National Infrastructure Simulation and Analysis Center (NISAC). FAIT addresses

the problem of charting and analyzing the interdependencies between infrastructure elements to help the Department of Homeland Security respond to hurricanes and other natural disasters. Although there are databases that show the location of generating plants, sub-stations, power lines, gas lines, telecommunication facilities, roads and other infrastructure elements, there are two problems with this data:

1. Interdependencies between elements are not shown explicitly in the databases. For example, databases of electrical power generation elements do not explicitly state which substations service which generating plants, relying on human experts to infer this from factors like co-location, ownership by the same utility, etc.
2. Interactions between different types of utilities, such as the effect of an electrical power outage on telecommunications hubs or gas pumping stations must be inferred from multiple data sources.

FAIT uses Jess to apply rules obtained from human experts to solve these problems. What is especially interesting about the FAIT architecture is its integration of Jess with multiple sources of infrastructure data, its use of a geographic information system to display interdependencies on maps, and its presentation of all this through an html front end.

The success of FAIT is intimately tied to its use of both open-source and commercially purchased tools. If the development team had faced the challenge of building all these components from scratch, the system would have cost an order of magnitude more than it did – if it could have been built at all. This approach of building extremely large systems from collections of independently designed components using the techniques discussed in this section has become an essential part of modern software development.

27 ID3: Learning from Examples

Chapter Objectives

Review of supervised learning and decision tree representation
- Representing decision trees as recursive structures
- A general decision tree induction algorithm

Information theoretic decision tree test selection heuristic

Chapter Contents

27.1 Introduction to Supervised Learning

In machine learning, *inductive learning* refers to training a learner through use of examples. The simplest case of this is rote learning, whereby the learner simply memorizes the training examples and reuses them in the same situations. Because they do not generalize from training data, rote learners can only classify exact matches of previous examples. A further limitation of rote learning is that the learned examples might contain conflicting information, and without some form of generalization, the learner cannot effectively deal with this noise. To be effective, a learner must apply heuristics to induce reliable generalizations from multiple training examples that can handle unseen situations with some degree of confidence.

A common inductive learning task is learning to classify specific instances into general categories. In *supervised learning*, a teacher provides the system with categorized training examples. This contrasts with clustering and similar unsupervised learning tasks where the learner forms its own categories from training data. See (Luger 2009) for a discussion of these different learning tasks. An example of a supervised inductive learning problem, which we will develop throughout the chapter is a bank wanting to train a computer learning system categorize new borrowers according to `credit risk` on the basis of properties such as their `credit history`, current `debt`, their `collateral`, and current `income`. One approach would be to look at the `credit risk`, as determined over time by the actual debt payoff history of data from previous borrowers to provide categorized examples. In this chapter we do exactly that, using the ID3 algorithm.

27.2 Representing Knowledge as Decision Trees

A decision tree is a simple form of knowledge representation that is widely used in both advisors and machine learning systems. Decision trees are recursive structures in which each node examines a property of a collection

of data, and then delegates further decision making to child nodes based on the value of that particular property (Luger 2009, Section 10.3). The leaf nodes of the decision tree are terminal states that return a class for the given data collection. We can illustrate decision trees through the example of a simple credit history evaluator that was used in (Luger 2009) in its discussion of the ID3 learning algorithm. We refer the reader to this book for a more detailed discussion, but will review the basic concepts of decision trees and decision tree induction in this section.

Assume we wish to assign a credit risk of high, moderate, or low to people based on the following properties of their credit rating:

Collateral, with possible values {adequate, none}

Income, with possible values {"0$ to $15K", "$15K to $35K",

"over $35K"}

Debt, with possible values {high, low}

Credit History, with possible values {good, bad, unknown}

We could represent risk criteria as a set of rules, such as "If debt is low, and credit history is good, then risk is moderate." Alternatively, we can summarize a set of rules as a decision tree, as in figure 27.1. We can perform a credit evaluation by walking the tree, using the values of the person's credit history properties to select a branch. For example, using the decision tree of figure 27.1, an individual with credit history = unknown, debt = low, collateral = adequate, and income = $15K to $35K would be categorized as having low risk. Also note that this particular categorization does not use the income property. This is a form of generalization, where people with these values for credit history, debt, and collateral qualify as having low risk, regardless of income.

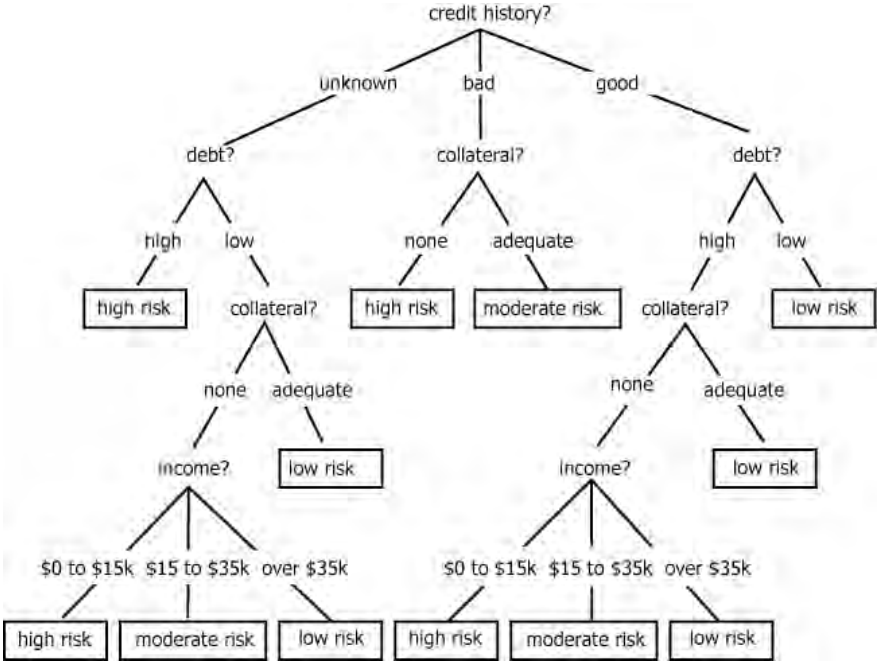

Figure 27.1 A Decision Tree for the Credit Risk Problem (Luger 2009)

Now, assume the following set of 14 training examples. Although this does not cover all possible instances, it is large enough to define a number of meaningful decision trees, including the tree of figure 27.1 (the reader may want to construct several such trees. See exercise 1). The challenge facing any inductive learning algorithm is to produce a tree that both covers all the training examples correctly, and has the highest probability of being correct on new instances.

risk	**collateral**	**income**	**debt**	**credit history**
high	none	$0 to $15K	high	bad
high	none	$15K to $35K	high	unknown
moderate	none	$15K to $35K	low	unknown
high	none	$0 to $15K	low	unknown
low	none	over $35K	low	unknown
low	adequate	over $35K	low	unknown
high	none	$0 to $15K	low	bad
moderate	adequate	over $35K	low	bad
low	none	over $35K	low	good
low	adequate	over $35K	high	good
high	none	$0 to $15K	high	good
moderate	none	$15K to $35K	high	good
low	none	over $35K	high	good
high	none	$15K to $35K	high	bad

A valuable heuristic for producing such decision trees comes from the time-honored logical principle of Occam's Razor. This principle, first articulated by the medieval logician, William of Occam, holds that we should always prefer the simplest correct solution to any problem. In our case, this would favor decision trees that not only classify all training examples, but also that do so, on average, by examining the fewest properties possible. The reason for this is straightforward: the simplest decision tree that correctly handles the known examples is the tree that makes the fewest assumptions about unknown instances. Stating it simply, the fewer assumptions made, the less likely we are to make an erroneous one.

Because omitting properties is a way of generalizing decision trees, and because the order in which the properties are tested determines the ability of the tree to omit properties while still matching all the test data, the order of tests from root down to leaf nodes is the major factor in inducing decision trees. This is captured in the following pseudo code for a recursive algorithm for inducing trees:

```
function induce_tree (example_set, Properties)
begin
  if all entries in example_set are the same class
  then return a leaf node labeled with that class
```

```
    else if Properties is empty
       then return a leaf node with default class
    else
      begin
          select a property, P, and
       make it the root of the current tree
        delete P from Properties
        for each value V of P
          begin
            create a branch of the tree labeled with V
            let partition_V be elements of
             example_set with values V of P
            let branch_V =
           induce_tree (partition_V, Properties)
            attach branch_V to root for value V of P
          end
        endfor
        return current root
      end
    endif
  end
```

This algorithm builds trees in a top-down fashion. It stops when all examples have the same categorization, thereby pruning extraneous branches of the tree. Using this algorithm, production of a simple (i.e., generalized) tree depends upon the order in which properties are selected. This, in turn, depends upon the selection function used to select the property to check in the current node of the tree.

For the decision tree induction, we use the original approach from the ID3 algorithm of (Quinlan 1986) elaborated by Luger (2009, Section 10.3). This approach uses information theory to select the property that gains the most information about the example set. Intuitively, this heuristic should minimize the number of properties the tree checks. We will explain it in detail later. We should note, however, that there are several important extensions of the early ID3 paradigm, differing only in a few operations. For example, C4.5 and C5.0 are Quinlan's (1996) own extensions that overcome a number of the original ID3 weaknesses. We will not implement C4.5/C5.0 here, but we should remember that more sophisticated or domain-specific modifications to the core decision tree induction algorithm may be desired by future developers using this code.

27.3 A Decision Tree Induction Program

Implementing this in Java raises at least two interesting problems. Managing trees, lists of examples, partitioning examples on various properties, and so forth is a challenge for designing data structures. Our example code will not be optimally efficient, but is intended to give the

student opportunities to improve performance by using table lookup and other techniques to reduce time spent scanning lists of examples. The other challenge will be in maintaining the quality of training data. We take a simplified approach of requiring all examples contain legitimate values for all desired properties. Although the machine learning literature is filled with techniques for managing missing or noisy data, this simple assumption will let us investigate a number of interesting Java techniques, such as immutable objects, error checks in constructors, etc.

Figure 27.2 shows the five classes that form the basis of our implementation. `AbstractDecisionTreeNode` defines the basic behaviors of a decision tree. It is a recursive structure, as shown by the use of an assembly link back to itself. `AbstractDecisionTreeNode` will define methods to solve a new instance by walking the tree, and the basic tree induction algorithm mentioned above. The method to evaluate a test property's partition of the example space into subproblems into will be abstract in this class, allowing definition of multiple alternative evaluation heuristics. The class, `InformationTheoreticDecisionTreeNode`, will implement the basic ID3 evaluation heuristic, which uses information theory to select a property that gives the greatest information gain on the set of training examples.

The remaining classes define and manage training examples. An `AbstractProperty` defines properties as <name, value> pairs. It is an abstract class, requiring subclasses define a method to test for legal <name, value> definitions. An `AbstractExample` defines examples as a set of properties and a categorization of those properties: i.e. a single row in the example table given above. Like `AbstractProperty`, it requires subclasses define domain specific checks for the validity of examples. Finally, `ExampleSet` maintains a set of training examples, such as is given in the table above. It enforces checks that all examples are of the same type, provides basic accessors, and also methods to partition an example set on specific properties.

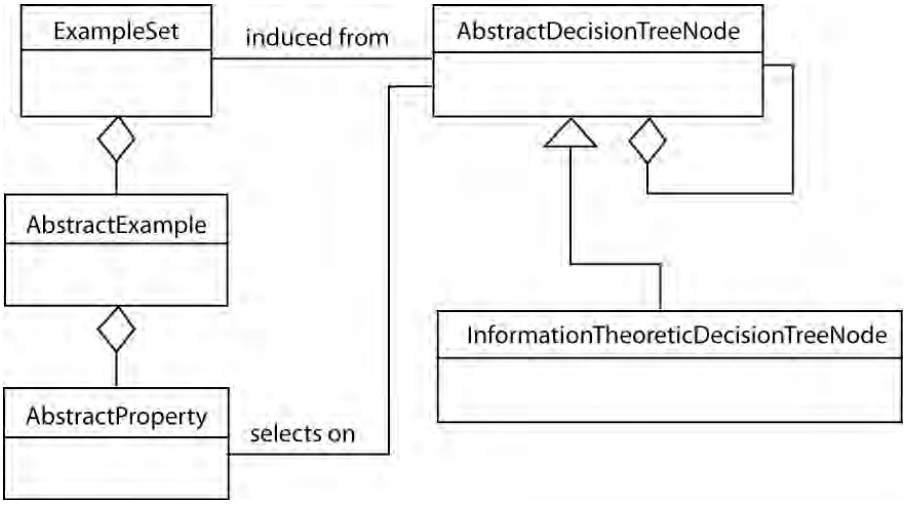

Figure 27.2 Class structure of decision tree nodes and examples

Properties as Immutable Objects

The basic definition of a property is straightforward: it consists of two strings, defining the name and value respectively. A simple initial implementation might be:

```
public class Property
{
  private String name = null;
  private String value = null;
  public Property(String name, String value)
  {
     this.name = name;
     this.value = value;
  }
  public String getName()
  {
     return name;
  }
  public String getValue()
  {
     return value;
  }
}
```

Although this gives the basic structure of the class, and would work in the program, it fails to perform any correctness checks on data values. The first of these the opportunity to perform type checks on property values. Referring to the credit evaluation example, the only values for debt are "high" and "low," and a robust program should check for them.

We can implement this by making `Property` an abstract class that uses an abstract method to test for legal property values. Each property type will be a subclass that defines this method. Our definition then becomes:

```
public abstract class AbstractProperty
{
  private String value = null;
  public AbstractProperty(String name,
        String value)
        throws IllegalArgumentException
  {
     if(isLegalValue(value) == false)
        throw
           new IllegalArgumentException(value +
              "is an illegal Value for Property " +
              getName());
     this.value = value;
  }
```

```
    public final String getValue()
    {
       return value;
    }
    public abstract boolean isLegalValue(String
       value);
    public abstract String getName();
  }
```

This version uses the `islegalValue(…)` method to check for bad values in the constructor, throwing an `IllegalArgumentException` if one is found. Since property is now an abstract class, any property type must define its own subclass defining the abstract methods. Also note that, since the name of a property is the same for all instances of a type, we have made `getName()` an abstract method as well. An example of how a property can implement this is given by this implementation of the debt property:

```
public class DebtProperty extends AbstractProperty {
  public static final String DEBT = "Debt";
  public static final String HIGH = "high";
  public static final String LOW = "low";
  public DebtProperty(String value)
  {
     super(value);
  }
  public boolean isLegalValue(String value)
  {
     return(value.equals(HIGH) ||
        value.equals(LOW));
  }
  public final String getName()
  {
     return DEBT;
  }
}
```

Although simple, the implementation of `AbstractProperty` has another interesting quality. Note that the member variable `value` is private, and we have not provided a set method other than through the constructor. This means that, once an instance of property is created, its value cannot change. This pattern is called an immutable object. Because immutable objects avoid many types of bugs (imagine the effect on the learning algorithm of changing a property value during execution), this should be used where it matches our intent. To reduce the chance that a well-intentioned programmer will change this, we should write code so as to make our intention clear. We can do this by making our get method `final`, to prevent subclasses from violating the immutability pattern, and

also by defining set methods that throw an exception if called. This completes the definition of AbstractProperty as:

```
public abstract class AbstractProperty
{
  private String value = null;
  public AbstractProperty(String value)
     throws IllegalArgumentException
  {
     if(isLegalValue(value) == false)
        throw
        new IllegalArgumentException(value +
        "is an illegal Property Value for " +
        getName());
     this.value = value;
  }
  public abstract boolean isLegalValue(String
     value);
  public abstract String getName();
  public final String getValue()
  {
     return value;
  }
  //Enforcing Immutable object pattern
  public final void setValue(String v)
     throws UnsupportedOperationException
  {
     throw new UnsupportedOperationException();
  }
  //Enforcing Immutable object pattern
  public final void setName(String n)
     throws UnsupportedOperationException
  {
     throw new UnsupportedOperationException();
  }
 }
```

Implementing Examples

Like a property, an example is conceptually simple: it is a collection of properties describing a problem instance and a categorization of that instance. In our credit example, the properties that form an example are debt, collateral, credit history, and income. The example category is a risk assessment. Each row of the example table in section 27.1 would be represented as an example. Like the property class, however, it also presents opportunities for insuring the validity of examples. In this case, we will require that an example consist only of specified properties, and that a

legal example include all properties. Examples also offer an opportunity to use an immutable object pattern, since it makes little sense to allow examples to change during the course of a learning session.

The structure of an example is similar to that of an `AbstractProperty`: it is an abstract class that requires subclasses define methods to support validity checks. We will follow the immutable object pattern, providing access methods but no "add," "set," or other modification methods, and requiring all properties be defined in the constructor.

The class has two member variables. A `category` is a `String` defining the classification of the example. In our credit example, this would be the risk level of high, moderate, or low. The `properties` member variable is a Map that indexes different properties by their name. We define two constructors. The primary constructor does error checks to require that each example contains all legal properties and only legal properties. The single argument constructor allows us to define uncategorized examples. Both of these call the private method, `addProperties` to add the elements of the `propertyList` argument to the properties member variable. This method also checks that the `propertyList` argument contains only legal values and all legal values. The implementation of `AbstractExample` is:

```
public abstract class AbstractExample
{
  private String category = null;
  private Map<String, AbstractProperty> properties
     = new HashMap <String, AbstractProperty> ();
  // Constructor for classified examples
  public AbstractExample(String category,
     AbstractProperty... propertyList)
     throws IllegalArgumentException
  {
     if(isLegalCategory(category) == false)
        throw
        new IllegalArgumentException(category +
        "is an illegal category for example.");
     this.category = category;
     addProperties(propertyList);
  }
  // Constructor for unclassified examples
  public AbstractExample(AbstractProperty...
        propertyList)
        throws IllegalArgumentException
  {
     addProperties(propertyList);
  }
```

```
private void addProperties(AbstractProperty[]
      propertyList)
      throws IllegalArgumentException
{
   Set<String> requiredProps =
      getPropertyNames();
   // check that all properties are legal
   for(int i = 0; i < propertyList.length;
      i++)
   {
      AbstractProperty prop =
         propertyList[i];
   if(requiredProps.contains(
            prop.getName()) == false)
         throw
         new IllegalArgumentException(
         prop.getName() +
         "illegal Property for example.");
      properties.put(prop.getName(), prop);
      requiredProps.remove(prop.getName());
   }
   // Check that all legal properties were used
   if(requiredProps.isEmpty() == false)
   {
      Object[] p = requiredProps.toArray();
      String props = "";
      for (int i = 0; i < p.length; i++)
         props += (String)p[i] + " ";
      throw
         new IllegalArgumentException(
         "Missing Properties in example: " +
         props);
   }
}
public AbstractProperty getProperty(
   String name)
{
   return properties.get(name);
}
public String getCategory()
{
   return category;
```

```
    }
    public String toString()
    {
        // to be defined by reader
    }
    public abstract Set<String> getPropertyNames();
}
```

This implementation of `AbstractExample` as an immutable object is incomplete in that it does not include the techniques demonstrated in `AbstractProperty` to enforce the immutability pattern. We leave this as an exercise.

Implementing `ExampleSet`

`ExampleSet`, along with `AbstractDecisionTreeNode`, is one of the most interesting classes in the implementation. This is because the decision tree induction algorithm requires a number of fairly complex operations for partitioning the example set on property values. The implementation presented here is simple and somewhat inefficient, storing examples as a simple vector. This requires examination of all examples to form partitions, retrieve examples with a specific value for a property, etc. We leave a more efficient implementation as an exercise.

In providing integrity checks on data, we have required that all examples be categorized, and that all examples belong to the same class.

The basic member variables and accessors are defined as:

```
public class ExampleSet
{
  private Vector<AbstractExample> examples =
     new Vector<AbstractExample>();
  private HashSet<String> categories =
     new HashSet<String>();
  private Set<String> propertyNames = null;
  public void addExample(AbstractExample e)
     throws IllegalArgumentException
  {
     if(e.getCategory() == null)
        throw new IllegalArgumentException(
           "Example missing categorization.");
     // Check that new example is of same class
     // as existing examples
     if((examples.isEmpty()) ||
        e.getClass() ==
           examples.firstElement().getClass())
     {
        examples.add(e);
        categories.add(e.getCategory());
```

```
            if(propertyNames == null)
               propertyNames =
                  new HashSet<String>(
                     e.getPropertyNames());
         }
         else
            throw new IllegalArgumentException(
               "All examples must be same type.");
      }
      public int getSize()
      {
         return examples.size();
      }
      public boolean isEmpty()
      {
         return examples.isEmpty();
      }
      public AbstractExample getExample(int i)
      {
         return examples.get(i);
      }
      public Set<String> getCategories()
      {
         return new HashSet<String>(categories);
      }
      public Set<String> getPropertyNames()
      {
         return new HashSet<String>(propertyNames);
      }
               // More complex methods to be defined.
      public int getExampleCountByCategory(String cat)
         throws IllegalArgumentException
      {
                                    // to be defined below.
      }
      public HashMap<String, ExampleSet> partition(
         String propertyName)
         throws IllegalArgumentException
      {
                                    // to be defined below.
      }
   }
```

As mentioned, this implementation is fairly simple. It stores examples as a `Vector`, so most retrieval or partitioning operations will require iterating through this list. The `categories` and `propertyNames` member variables are a convenience, allowing simpler access of these values. Since example sets should not change during a learning session, we could use an immutable object pattern in the `ExampleSet` implementation. This implementation does not, since it would lead to extremely complex constructor implementations. Instead, we implemented an `addExample` method. This method performs simple data integrity checks, requiring that all examples be of the same type, and prohibiting unclassified examples. Reworking this using an immutable pattern is left as an exercise. The remaining methods are straightforward accessors.

`ExampleSet` includes a number of methods to support the induction algorithm. The first of these counts the number of examples that belong to a given category:

```
public int getExampleCountByCategory(String cat)
  throws IllegalArgumentException
{
  Iterator<AbstractExample> iter =
     examples.iterator();
  AbstractExample example;
  int count = 0;
  while(iter.hasNext())
  {
     example = iter.next();
     if(example.getCategory().equals(cat))
           count++;
   }
     return count;
 }
```

A more complex method partitions the example set according to different examples value for a specified property. `Partition` takes as argument a property name, and returns an instance of `HashMap<String, ExampleSet>` where each key is a property value, and each value is an instance of `ExampleSet` containing examples that have that value for the chosen property. `Partition` calls to private methods, `getValues`, which returns a list of values for a property that appear in the example set, and `getExamplesByProperty`, which constructs a new instance of `ExampleSet` where each example has the same value for a property.

```
public HashMap<String, ExampleSet> partition(
  String propertyName)
  throws IllegalArgumentException
{
  HashMap<String, ExampleSet> partition =
     new HashMap<String, ExampleSet>();
```

```
    Set<String> values = getValues(propertyName);
    Iterator<String> iter = values.iterator();
    while(iter.hasNext())
    {
       String val = iter.next();
       ExampleSet examples =
          getExamplesByProperty(propertyName,
             val);
       partition.put(val, examples);
    }
    return partition;
}
private Set<String> getValues(String propName)
{
    HashSet<String>values = new HashSet<String>();
    Iterator<AbstractExample> iter =
       examples.iterator();
    while(iter.hasNext())
    {
       AbstractExample ex = iter.next();
    values.add(ex.getProperty(propName).
          getValue());
    }
    return values;
}
private ExampleSet getExamplesByProperty(
    String propName, String value)
    throws IllegalArgumentException
{
    ExampleSet result = new ExampleSet();
    Iterator<AbstractExample> iter =
       examples.iterator();
    AbstractExample example;
    while(iter.hasNext())
    {
       example = iter.next();
       if(example.getProperty(propName).getValue().
             equals(value))
          result.addExample(example);
    }
    return result;
 }
```

Placing the partitioning algorithm in a method of `ExampleSet`, rather than in the actual decision tree induction algorithm was an interesting design decision. The reason for this choice was a desire to treat `ExampleSet` as an abstract data type, including all operations on it in its class definition.

Although this implementation works, it is inefficient, performing multiple iterations through lists of examples. An alternative approach would construct more complex sets of indices of examples by property and value on construction. Trying this approach and evaluating its effectiveness is left as an exercise.

Implementing Decision Tree Nodes

A decision tree node will define methods to solve problems by walking the tree, as described in section 27.1. We have also chosen to implement the basic induction algorithm in the decision tree class. Justification for this decision was that the inherently recursive nature of the induction algorithm matched the recursive structure of trees, simplifying the implementation. Because the induction algorithm is general, and could be used with a variety of heuristics for evaluating candidate example partitions, we will make the basic implementation of decision trees an abstract class.

The basic definition of `AbstractDecisionTreeNode` appears below. Member variables include `category`, which is set to a categorization in leaf nodes; for internal nodes, its value is not defined. `DecisionPropertyName` is the property on which the node branches; it is undefined for leaf nodes. `Children` is a `HashMap` that indexes child nodes by values of `decisionPropertyName`. Each constructor calls `induceTree` to perform tree induction. Note that the two-argument constructor is protected. Its second argument is the list of unused properties for consideration by the induction algorithm, and it is only used by the `induceTree` method. The remaining methods defined below are straightforward accessors.

```
public abstract class AbstractDecisionTreeNode
{
  private String category = null;
  private String decisionPropertyName = null;
  private HashMap<String,AbstractDecisionTreeNode>
     children = new
     HashMap<String,AbstractDecisionTreeNode>();
  public AbstractDecisionTreeNode (
        ExampleSet examples)
     throws IllegalArgumentException
  {
     induceTree(examples,
        examples.getPropertyNames());
  }
  protected AbstractDecisionTreeNode(ExampleSet
     examples, Set<String> selectionProperties)
```

```
   throws IllegalArgumentException
{
   induceTree(examples, selectionProperties);
}
 public boolean isLeaf()
{
   return children.isEmpty();
 }
public String getCategory()
{
   return category;
}
public String getDecisionProperty()
{
   return decisionPropertyName;
 }
public AbstractDecisionTreeNode getChild(String
   propertyValue)
{
   return children.get(propertyValue);
}
 public void addChild(String propertyValue,
   AbstractDecisionTreeNode child)
{
   children.put(propertyValue, child);
 }
public String Categorize(AbstractExample ex)
{
   // defined below
}
public void induceTree(ExampleSet examples,
      Set<String> selectionProperties)
      throws IllegalArgumentException
{
   // defined below
}
public void printTree(int level)
{
   // implementation left as an exercise
}
protected abstract double
```

```
      evaluatePartitionQuality(HashMap<String,
         ExampleSet> part, ExampleSet examples)
         throws IllegalArgumentException;
   protected abstract AbstractDecisionTreeNode
      createChildNode(ExampleSet examples,
      Set<String> selectionProperties)
      throws IllegalArgumentException;
}
```

Note the two abstract methods for evaluating a candidate partition and creating a new child node. These will be implemented on 27.3.

`Categorize` categorizes a new example by performing a recursive tree walk.

```
public String categorize(AbstractExample ex)
{
   if(children.isEmpty())
      return category;
   if(decisionPropertyName == null)
      return category;

   AbstractProperty prop =
      ex.getProperty(decisionPropertyName);
     AbstractDecisionTreeNode child =
      children.get(prop.getValue());
      if(child == null)
      return null;
   return child.categorize(ex);
}
```

`InduceTree` performs the induction of decision trees. It deals with four cases. The first is a normal termination: all examples belong to the same category, so it creates a leaf node of that category. Cases two and three occur if there is insufficient information to complete a categorization; in this case, the algorithm creates a leaf node with a null category.

Case four performs the recursive step. It iterates through all properties that have not been used in the decision tree (these are passed in the parameter `selectionProperties`), using each property to partition the example set. It evaluates the example set using the abstract method, `evaluatePartitionQuality`. Once it finds the best evaluated partition, it constructs child nodes for each branch.

```
public void induceTree(ExampleSet examples,
       Set<String> selectionProperties)
   throws IllegalArgumentException
{
                        // Case 1: All instances are the same
                        // category, the node is a leaf.
```

```
if(examples.getCategories().size() == 1)
  {
     category = examples.getCategories().
        iterator().next();
     return;
  }
                    //Case 2: Empty example set. Create
                    // leaf with no classification.
    if(examples.isEmpty())
     return;
                    //Case 3: Empty property set; could not classify.
  if(selectionProperties.isEmpty())
     return;
                    // Case 4: Choose test and build subtrees.
                    // Initialize by partitioning on first
                    // untried property.
  Iterator<String> iter =
          selectionProperties.iterator();
  String bestPropertyName = iter.next();
  HashMap<String, ExampleSet> bestPartition =
        examples.partition(bestPropertyName);
  double bestPartitionEvaluation =
         evaluatePartitionQuality(bestPartition,
        examples);
                    // Iterate through remaining properties.
  while(iter.hasNext())
  {
     String nextProp = iter.next();
     HashMap<String, ExampleSet> nextPart =
             examples.partition(nextProp);
     double nextPartitionEvaluation =
             evaluatePartitionQuality(nextPart,
           examples);
                    // Better partition found. Save.
     if(nextPartitionEvaluation >
        bestPartitionEvaluation)
     {
        bestPartitionEvaluation =
           nextPartitionEvaluation;
        bestPartition = nextPart;
        bestPropertyName = nextProp;
     }
  }
                    // Create children; recursively build tree.
  this.decisionPropertyName = bestPropertyName;
```

```
Set<String> newSelectionPropSet =
      new HashSet<String>(selectionProperties);
  newSelectionPropSet.remove(decisionPropertyName);
iter = bestPartition.keySet().iterator();
while(iter.hasNext())
{
   String value = iter.next();
   ExampleSet child = bestPartition.get(value);
   children.put(value,
      createChildNode(child,
         newSelectionPropSet));
}
```

27.4 ID3: An Information Theoretic Tree Induction Algorithm

The heart of the ID3 algorithm is its use of information theory to evaluate the quality of candidate partitions of the example set by choosing properties that gain the most information about an examples categorization. Luger (2009) discusses this approach in detail, but we will review it briefly here.

Shannon (1948) developed a mathematical theory of information that allows us to measure the information content of a message. Widely used in telecommunications to determine such things as the capacity of a channel, the optimality of encoding schemes, etc., it is a general theory that we will use to measure the quality of a decision property.

Shannon's insight was that the information content of a message depended upon two factors. One was the size of the set of all possible messages, and the probability of each message occurring. Given a set of possible messages, $M = \{m_1, m_2 \ldots m_n\}$, the information content of any individual message is measured in bits by the sum, across all messages in M of the probability of each massage times the log to the base 2 of that probability.

$$I(M) = \sum - p(m_i) \log 2\ p(m_i)$$

Applying this to the problem of decision tree induction, we can regard a set of examples as a set of possible messages about the categorization of an example. The probability of a message (a given category) is the number of examples with that category divided by the size of the example set. For example, in the table in section 27.1, there are 14 examples. Six of the examples have high risk, so p(risk = high) = 6/14. Similarly, p(risk = moderate) = 3/14, and p(risk = low) = 5/14. So, the information in any example in the set is:

I(example set) = -6/14 log (6/14) -3/14 log (3/14) -5/14 log (5/14)
= - 6/14 * (-1.222) - 3/14 * (-2.222) - 5/14 * (-1.485)
= 1.531 bits

We can think of the recursive tree induction algorithm as gaining information about the example set at each iteration. If we assume a set of

training instances, C, and a property P with n values, then P will partition C into n subsets, {c_1, c_2, . . . cY}. The information needed to finish inducing the tree can be measured as the sum of the information in each subset of the partition, weighted by the size of that partition. That is, the expected information gain to complete the tree, E, is computed by:

$$E(P) = S\ (|c_i|/|C|) * I(c_i)$$

Therefore, the information gained for property P is:

$$Gain(P) = I(C) - E(P)$$

The ID3 algorithm uses this value to rank candidate partitions.

Implementing Information Theoretic Evaluation

We will implement this in a subclass of `AbstractDecisionTreeNode` called `InformationTheoreticDecisionTreeNode`. This class will implement the two abstract methods of the parent class, along with needed constructors. The `createChildNode` method is called in `AbstractDecisionTreeNode` to create the proper type of child node. `EvaluatePartitionQuality` computes the information gain of a partition. It calls the private methods `computeInformation` and `log2`.

```
public class InformationTheoreticDecisionTreeNode
  extends  AbstractDecisionTreeNode
{
  public InformationTheoreticDecisionTreeNode(
     ExampleSet examples)
     throws IllegalArgumentException
  {
     super(examples);
  }
  public InformationTheoreticDecisionTreeNode(
     ExampleSet examples,
     Set<String> selectionProperties)
     throws IllegalArgumentException
  {
     super(examples, selectionProperties);
  }
   protected AbstractDecisionTreeNode
     createChildNode(
     ExampleSet examples,
     Set<String> selectionProperties)
     throws IllegalArgumentException
  {
     return new
        InformationTheoreticDecisionTreeNode(
           examples, selectionProperties);
     }
```

```
  protected double evaluatePartitionQuality(
         HashMap<String, ExampleSet> part,
         ExampleSet examples)
     throws IllegalArgumentException
  {
     double examplesInfo =
        computeInformation(examples);
     int totalSize = examples.getSize();
     double expectedInfo = 0.0;
     Iterator<String> iter =
        part.keySet().iterator();
     while(iter.hasNext())
     {
        ExampleSet ex = part.get(iter.next());
        int partSize = ex.getSize();
        expectedInfo += computeInformation(ex)
           * partSize/totalSize;
  }
     return examplesInfo - expectedInfo;
  }
  private double computeInformation(
         ExampleSet examples)
     throws IllegalArgumentException
  {
     Set<String> categories =
        examples.getCategories();
     double info = 0.0;
     double totalCount = examples.getSize();
        Iterator<String> iter =
            categories.iterator();
     while (iter.hasNext())
     {
        String cat = iter.next();
        double catCount = examples.
           getExampleCountByCategory(cat);
        info += -(catCount/totalCount)*
              log2(catCount/totalCount);
     }
     return info;
  }
  private double log2(double a)
  {
     return Math.log10(a)/Math.log10(2);
  }
}
```

Exercises

1. Construct two or three different trees that correctly classify the training examples in the table of section 27.1. Compare their complexity using average path length from root to leaf as a simple metric. What informal heuristics would use in constructing the simplest trees to match the data? Manually build a tree using the information theoretic test selection algorithm from the ID3 algorithm. How does this compare with your informal heuristics?

2. Extend the definition of `AbstractExample` to enforce the immutable object pattern using `AbstractProperty` as an example.

3.The methods `AbstractExample` and `AbstractProperty` throw exceptions defined in Java, such as `IllegalArgumentException` or `UnsupportedOperationException` when passed illegal values or implementers try to violate the immutable object pattern. An alternative approach would use user-defined exceptions, defined as subclasses of java.lang.RuntimeException. Implement this approach, and discuss its advantages and disadvantages.

4. The implementation of `ExampleSet` in section 27.2.3 stores component examples as a simple vector. This requires iteration over all examples to partition the example set on a property, count categories, etc. Redo the implementation using a set of maps to allow constant time retrieval of examples having a certain property value, category, etc. Evaluate performance for this implementation and that given in the chapter.

5. Complete the implementation for the credit risk example. This will involve creating subclasses of `AbstractProperty` for each property, and an appropriate subclass of `AbstractExample`. Also, write a class and methods to test your code.

28 Genetic and Evolutionary Computing

Chapter Objectives

A brief introduction to the genetic algorithms
- Genetic operators include
  - Mutation
  - Crossover

An example GA application worked through
- The WordGuess problem

Appropriate object hierarchy created
Generalizable to other GA applications
Exercises emphasize GA interface design

Chapter Contents

28.1 Introduction

The genetic algorithm (GA) is one of a number of computer programming techniques loosely based on the idea of natural selection. The idea of applying principles of natural selection to computing is not new. By 1948, Alan Turing proposed "genetical or evolutionary search" (Turing 1948). Less than two decades later, H.J. Bremmermann performed computer simulations of "optimization through evolution and recombination" (Eiben and Smith 1998). It was John Holland who coined the term, *genetic algorithm* (Holland 1975). However, the GA was not widely studied until 1989, when D.E. Goldberg showed that it could be used to solve a significant number of difficult problems (Goldberg 1989). Currently, many of these threads have come together under the heading *evolutionary computing* (Luger 2009, Chapter 12).

28.2 The Genetic Algorithm: A First Pass

The Genetic Algorithm is based loosely on the concept of natural selection. Individual members of a species who are better adapted to a given environment reproduce more successfully. They pass their adaptations on to their offspring. Over time, individuals possessing the adaptation form a new species that is particularly suited to the environment. The genetic algorithm applies the *metaphor* of natural selection to optimization problems. No claim is made about its biological accuracy, although individual researchers have proposed mechanisms both with and without a motivating basis from nature.

A candidate solution for a genetic algorithm is often called a *chromosome*. The chromosome is composed of multiple *genes*. A collection of

chromosomes is called a *population*. The GA randomly generates an initial population of chromosomes, which are then ranked according to a *fitness function* (Luger 2009, Section 12.1).

Consider an example drawn from structural engineering. Structural engineers make use of a component known as a *truss*. Trusses come in many varieties, the simplest of which should be familiar to anyone who has noticed the interconnected triangular structures found in bridges and cranes. Figure 28.1 is an example of the canonical 64-bar truss (Ganzerli et al. 2003), which appears in the civil engineering literature on optimization. The arrows are loads, expressed in a unit known as a Kip. Engineers would like to minimize the volume of a truss, taken as the cross-sectional area of the bars multiplied by their length.

To solve this problem using a GA, we first randomly generate a population of trusses. Some of these will stand up under a given load, some will not. Those that fail to meet the load test are assigned a severe penalty. The ranking in this problem is based on volume. The smaller the truss volume, after any penalty has been assigned, the more fit the truss. Only the fittest individuals are selected for reproduction. It has been shown that the truss design problem is NP-Complete (Overbay et al. 2006). Engineers have long-recognized the difficulty of truss design, most often developing good enough solutions with the calculus-based optimization techniques available to them (Ganzerli et al. 2003).

By the late nineties, at least two groups were applying genetic algorithms to very large trusses and getting promising results (Rajeev and Krishnamoorthy 1997), (Ghasemi et al. 1999). Ganzerli et al. (2003) took this work a step further by using genetic algorithms to optimize the 64-bar truss with the added complexity of load uncertainty. The point here is not simply that the GA is useful in structural engineering, but that it has been applied in hundreds of ways in recent years, structural engineering being an especially clear example. A number of other examples, including the traveling salesperson and SAT problems are presented in Luger (2009, Section 12.1). The largest venue for genetic algorithm research is *The Genetic and Evolutionary Computation Conference* (GECCO 2007). Held in a different city each summer, the papers presented range from artificial life through robotics to financial and water quality systems.

Despite the breadth of topics addressed, the basic outline for genetic algorithm solvers across application domains is very similar. Search through the problem space is guided by the *fitness-function*. Once the fitness-function is designed, the *GA* traverses the space over many iterations, called *generations*, stopping only when some pre-defined convergence criterion is met. Further, the only substantial differences between one application of the GA and the next is the representation of the chromosome for the problem domain and the fitness function that is applied to it. This lends itself very nicely to an object-oriented implementation that can be easily generalized to multiple problems. The technique is to build a generic *GA* class with specific implementations as subclasses.

WordGuess Example

Consider a simple problem called *WordGuess* (Haupt and Haupt 1998). The user enters a target word at the keyboard. The GA guesses the word. In this case, each letter is a gene, each word a chromosome, and the total collection of words is the population. To begin, we randomly generate a sequence of chromosomes of the desired length. Next, we rank the generated chromosomes for fitness. A chromosome that is identical with the target has a fitness of zero. A chromosome that differs in one letter has a fitness of 1 and so on. It is easy to see that the size of the search space for *WordGuess* increases exponentially with the length of the word. In the next few sections, we will develop an object-oriented solution to this problem.

Suppose we begin with a randomly generated population of 128 character strings. After ranking them, we immediately eliminate the half that is least fit. Of the 64 remaining chromosomes, the fittest 32 form 16 breeding pairs. If each pair produces 2 offspring, the next generation will consist of the 32 parents plus the 32 children.

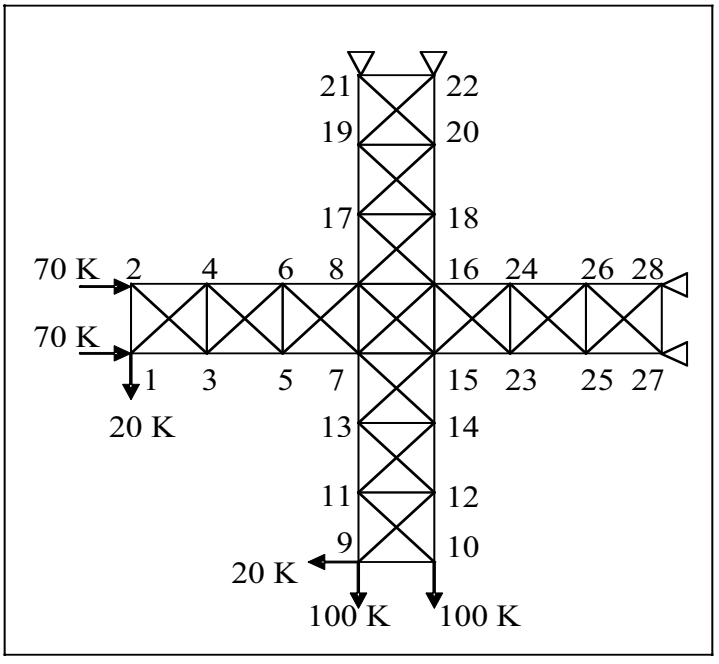

Figure 28.1 A system of trusses to be optimized with a set of genetic operators.

Having decided who may reproduce, we mate them. The GA literature is filled with clever mating strategies, having more or less biological plausibility. We consider two, *TopDown* and *Tournament*. In TopDown, the fittest member of the population mates with the next most fit and so on, until the breeding population is exhausted. Tournament is a bit more complex, and slightly more plausible (Haupt and Haupt 1998). Here we choose a subset of chromosomes from the breeding population. The fittest chromosome within this subset becomes Parent A. We do the same thing again, to find its mate, Parent B. Now we have a breeding pair. We continue with this process until we have created as many breeding pairs as we need.

Mating is how each chromosome passes its genes to future generations. Since mating is an attempt to simulate (and simplify) recombinant DNA, many authors refer to it as *recombination* (Eiben and Smith 2003). As with pairing, many techniques are available. *WordGuess* uses a single technique called *Crossover*. Recall that each chromosome consists of *length*(*chromosome*) genes. The most natural data structure to represent a chromosome is an array of *length*(*chromosome*) positions. A gene—in this case an alphabetic character—is stored in each of these positions. Crossover works like this:

1. Generate a random number n, 0 <= n < length(chromosome). This is called the Crossover Point.
2. Parent A passes its genes in positions 0 … n to Child 1.
3. Parent B passes its genes in positions 0 … n to Child 2.
4. Parent A passes it genes in positions n + 1 … length(chromosome – 1) to the corresponding positions in Child 2.
5. Parent B passes its genes in positions n + 1 … length(chromosome – 1) to the corresponding positions in Child 1

Figure 28.2 illustrates mating with $n = 4$. The parents, PA and PB produce the two children CA and CB.

After the reproducing population has been selected, paired, and mated, the final ingredient is the application of random mutations. The importance of random mutation in nature is easy to see. Favorable (as well as unfavorable!) traits have to arise before they can be passed on to offspring. This happens through random variation, caused by any number of natural mutating agents. Chemical mutagens and radiation are examples. Mutation guarantees that new genes are introduced into the gene pool. Its practical effect for the GA is to reduce the probability that the algorithm will converge on a local minimum. The percentage of genes subject to mutation is a design parameter in the solution process.

The decision of when to stop producing new generations is the final component of the algorithm. The simplest possibility, the one used in *WordGuess*, is to stop either after the GA has guessed the word or 1000 generations have passed. Another halting condition might be to stop when some parameter P percent of the population is within Q standard deviations of the population mean.

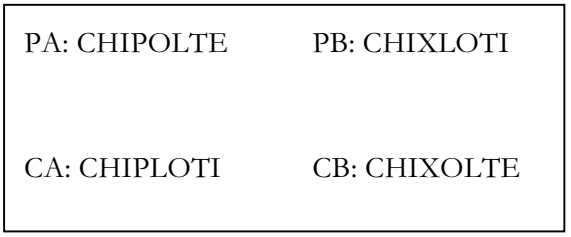

Figure 28.2 Recombination with crossover at the point n = 4.

The entire process can be compactly expressed through the while-loop:

```
GA(population)
{
   Initialize(population);
   ComputeCost(population);
   Sort(population);
   while (not converged on acceptable solution)
   {
      Pair(population);
      Mate(population);
      Mutate(population);
      Sort(population);
      TestConvergence(population);
   }
}
```

28.3 A GA Implementation in Java

`WordGuess` is written in the Java programming language with object-oriented (OO) techniques developed to help manage the search complexity. An OO software system consists of a set of interrelated structures known as classes. Each class can perform a well-defined set of operations on a set of well-defined operands. The operations are referred to as *methods*, the operands as *member variables*, or just *variables*.

The Class Structure

The classes interrelate in two distinct ways. First, classes may inherit properties from one another. Thus, we have designed a class called `GA`. It defines most of the major operations needed for a genetic algorithm. Knowing that we want to adapt `GA` to the problem of guessing a word typed at the keyboard, we define the class `WordGuess`. Once having written code to solve a general problem, that code is available to more specific instances of the problem. A hypothetical inheritance structure for the genetic algorithm is shown in Figure 28.3, where the upward pointing arrows are inheritance links. Thus, `WordGuess` inherits all classes and variables defined for the generic `GA`.

Second, once defined, classes may make use of one another. This relationship is called *compositionality*. `GA` contains several component classes:

- `Chromosome` is a representation of an individual population member.
- `Pair` contains all pairing algorithms developed for the system. By making `Pair` its own class, the user can add new methods to the system without changing the core components of the code.
- `Mate` contains all mating algorithms developed for the system.
- `SetParams`, `GetParams`, and `Parameters` are mechanisms to store and retrieve parameters.

- `WordGuessTst` sets the algorithm in motion.

Finally, class `GA` makes generous use of Java's pre-defined classes to represent the population, randomly generate chromosomes, and to handle files that store both the parameters and an initial population. `GA` is character-based. A Graphical User Interface (GUI) can be implemented with Java's facilities for GUIs and Event-Driven programming found in the `javax.swing` package (see Exercise 28.3).

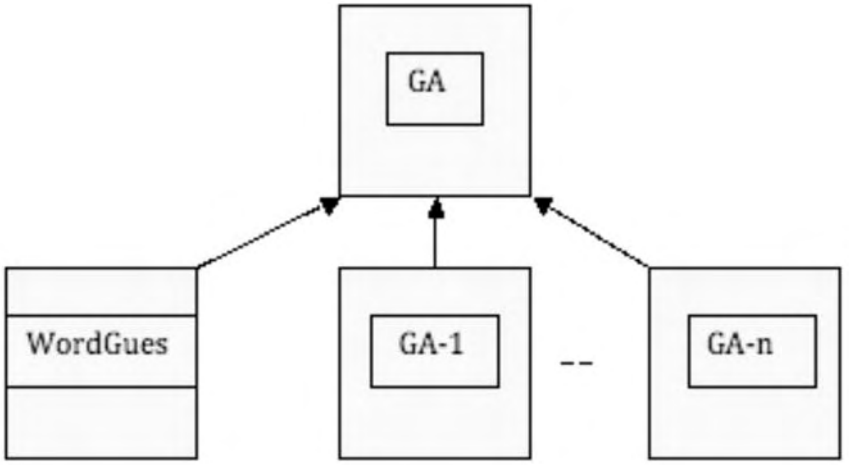

Figure 28.3 The inheritance hierarchy for implementing the GA.

The Class Chromosome

The variables reflect what a class knows about itself. Class `Chromosome` must know how many genes it has, its fitness, and have a representation for its genes. The number of genes and the fitness of the chromosome can be easily represented as integers. The representation of the genes poses a design problem. For `WordGuess`, a character array works nicely. For an engineering application, we might want the chromosome to be a vector of floating point variables. The most general representation is to use Java's class `Object` and have specific implementations, like `WordGuess`, define their own chromosomes (see Exercise 28.4).

The methods describe what a class does. Class `Chromosome` must be able to set and return its fitness, set and return the number of its genes, display its genes, and determine if it is equal to another chromosome. The Java code that implements the class `Chromosome` follows.

```
public class Chromosome
{
  private int CH_numGenes;
  protected int CH_cost;
  private  Object[] CH_gene;
  public Chromosome(int genesIn)
  {
     CH_numGenes = genesIn;
     CH_gene = new char[CH_numGenes];
  }
```

```
    public int GetNumGenes()
    {
       return CH_numGenes;
    }
    public void SetCost(int cost)
    {
       CH_cost = cost;
    }
    public void SetGene(int index, Object value)
    {
       CH_gene[index] = value;
    }
    public boolean Equals(String target)
    {
       for (int i = 0; i < CH_numGenes; i++)
       if (CH_gene[i] != target.charAt(i))
          return false;
       return true;
    }
}
```

Classes Pair and Mate

Chromosomes must be paired and mated. So that we can experiment with more than a single pairing or mating algorithm, we group multiple versions into classes `Pair` and `Mate`. Since pairing and mating are done over an entire population, before we define `Pair` and `Mate` we must decide upon a representation for the population. A population is a list of chromosomes. Java's built-in collection classes are contained in the `java.util` library and known as the *Java Collection Framework*. Two classes, `ArrayList` and `LinkedList` support list behavior. It is intuitively easy to conceive of a population as an array of chromosomes. Accordingly, we use the class `ArrayList` to define a population as follows:

```
ArrayList<Chromosome> GA_pop;
GA_pop = new ArrayList<Chromosome>();
```

The first line defines a variable, `GA_pop` as type `ArrayList`. The second creates an instance of `GA_pop`.

`WordGuess` implements a single paring algorithm, `TopDown`. Tournament pairing is left as an exercise. `Pair` has to know the population that is to be paired and the number of mating pairs. Since only half of the population is fit enough to mate, the number of mating pairs is the population size divided by 4. Here we can see one of the benefits of using pre-defined classes. `ArrayList` provides a method that returns the size of the list. The code for `Pair` follows:

```
public class Pair
{
  private ArrayList<Chromosome> PR_pop;
```

```
    public  Pair(ArrayList<Chromosome> population)
    {
       PR_pop = population;
    }
    public int TopDown()
    {
       return (PR_pop.size() / 4);
    }
}
```

Class `Mate` also implements a single algorithm, `Crossover`. It is slightly more complex than `Pair`. To implement `Crossover`, we need four chromosomes, one for each parent, and one for each child. We also need to know the crossover point, as explained in Section 28.2, the number of genes in a chromosome, and the size of the population. We now present the member variables and the constructor for `Mate`:

```
public class Mate
{
  private Chromosome MT_father,
     MT_mother,
     MT_child1,
     MT_child2;
  private  int   MT_posChild1,
     MT_posChild2,
     MT_posLastChild,
     MT_posFather,
     MT_posMother,
     MT_numGenes,
     MT_numChromes;
  public   Mate(ArrayList<Chromosome> population,
                int numGenes, int numChromes)
  {
     MT_posFather  = 0;
     MT_posMother  = 1;
     MT_numGenes   = numGenes;
     MT_numChromes = numChromes;
     MT_posChild1  = population.size()/2;
     MT_posChild2  = MT_posChild1 + 1;
     MT_posLastChild = population.size() - 1;
     for (int i = MT_posLastChild;
           i >= MT_posChild1; i--)
        population.remove(i);
     MT_posFather = 0;
     MT_posMother = 1;
```

```
    }
                    // Remaining method implemented below.
}
```

`Mate` takes a population of `chromosome` as a parameter and returns a mated population. The `for`-loop eliminates the least fit half of the population to make room for the two children per breeding pair.

`Crossover`, the only other method in `Mate`, is presented next. It implements the algorithm described in Section 28.2. Making use of the `Set/Get` methods of `Chromosome`, `Crossover` blends the chromosomes of each breeding pair. When mating is complete, the breeding pairs are in the top half of the `ArrayList`, the children are in the bottom half.

```
public ArrayList<Chromosome> Crossover(
  ArrayList<Chromosome> population, int numPairs)
{
  for (int j = 0; j < numPairs; j++)
  {
     MT_father =  population.get(MT_posFather);
     MT_mother =  population.get(MT_posMother);
     MT_child1 =  new Chromosome(MT_numGenes);
     MT_child2 = new Chromosome(MT_numGenes);
     Random rnum = new Random();
     int crossPoint = rnum.nextInt(MT_numGenes);
                                          // left side
     for (int i = 0; i < crossPoint; i++)
     {
        MT_child1.SetGene(i,
           MT_father.GetGene(i));
        MT_child2.SetGene(i,
           MT_mother.GetGene(i));
     }
                                          // right side
     for (int i = crossPoint;
            < MT_numGenes;i++)
     {
        MT_child1.SetGene(i,
           MT_mother.GetGene(i));
        MT_child2.SetGene(i,
           MT_father.GetGene(i));
     }
     population.add(MT_posChild1,MT_child1);
     population.add(MT_posChild2,MT_child2);
     MT_posChild1 = MT_posChild1 + 2;
     MT_posChild2 = MT_posChild2 + 2;
```

```
            MT_posFather = MT_posFather + 2;
            MT_posMother = MT_posMother + 2;
        }
        return population;
    }
```

The GA Class

Having examined its subclasses, it is time to look at class **GA**, itself. We never create an instance of `class GA`. **GA** exists only so that its member variables and methods can be inherited, as in Figure 28.3. Classes that may not be instantiated are called *abstract*. The classes higher in the hierarchy are called *superclasses*. Those lower in the hierarchy are called *subclasses*. Member variables and methods designated `protected` in a super class are available to its subclasses.

GA contains the population of chromosomes, along with the various parameters that its subclasses need. The parameters are the size of the initial population, the size of the pared down population, the number of genes, the fraction of the total genes to be mutated, and the number of iterations before the program stops. The parameters are stored in a file manipulated through the classes `Parameters`, `SetParams`, and `GetParams`. We use object semantics to manipulate the files. Since file manipulation is not essential to a GA, we will not discuss it further. The class declaration **GA**, its member variables, and its constructor follow.

```
public abstract class GA extends Object
{
    protected int  GA_numChromesInit;
    protected int  GA_numChromes;
    protected int  GA_numGenes;
    protected double  GA_mutFact;
    protected int  GA_numIterations;
    protected ArrayList<Chromosome> GA_pop;
    public GA(String ParamFile)
    {
        GetParams GP  = new GetParams(ParamFile);
        Parameters P   = GP.GetParameters();
        GA_numChromesInit = P.GetNumChromesI();
        GA_numChromes  = P.GetNumChromes();
        GA_numGenes    = P.GetNumGenes();
        GA_mutFact  = P.GetMutFact();
        GA_numIterations  = P.GetNumIterations();
        GA_pop   = new ArrayList<Chromosome>();
        }
                //Remaining methods implemented below.
}
```

The first two lines of the constructor create the objects necessary to read the parameter files. The succeeding lines, except the last, read the file and

store the results in class **GA**'s members variables. The final line creates the data structure that is to house the population. Since an **ArrayList** is an expandable collector, there is no need to fix the size of the array in advance.

Class **GA** can do all of those things common to all of its subclasses. Unless you are a very careful designer, odds are that you will not know what is common to all of the subclasses until you start building prototypes. Object-oriented techniques accommodate an iterative design process quite nicely. As you discover more methods that can be shared across subclasses, simply push them up a level to the superclass and recompile the system.

Superclass **GA** performs general housekeeping tasks along with work common to all its subclasses. Under housekeeping tasks, we want a super class **GA** to display the entire population, its parameters, a chromosome, and the best chromosome within the population. We also might want it to tidy up the population by removing those chromosomes that will play no part in evolution. This requires a little explanation. Two of the parameters are **GA_numChromesInit** and **GA_numChromes**. Performance of a GA is sometimes improved if we initially generate more chromosomes than are used in the GA itself (Haupt and Haput 1998). The first task, then, is to winnow down the number of chromosomes from the number initially generated (**GA_numChromesInit**) to the number that will be used (**GA_numChromes**).

Under shared tasks, we want the superclass **GA** to create, rank, and mutate the population. The housekeeping tasks are very straightforward. The shared method that initializes the population follows:

```
protected void InitPop()
{
  Random rnum = new Random();
  char letter;
  for (int index = 0;
        index < GA_numChromesInit; index++)
  {
     Chromosome Chrom =
        new Chromosome(GA_numGenes);
     for (int j = 0; j < GA_numGenes; j++)
     {
        letter = (char)(rnum.nextInt(26) + 97);
         Chrom.SetGene(j,letter);
     }
     Chrom.SetCost(0);
     GA_pop.add(Chrom);
  }
}
```

Initializing the population is clear enough, though it does represent a design decision. We use a nested **for** loop to create and initialize all genes

within a chromosome and then to add the chromosomes to the population. Notice the use of Java's pseudo-random number generator. In keeping with the object-oriented design, `Random` is a class with associated methods. `rnum.nextInt(26)` generates a pseudo-random number in the range [0..25]. The design decision is to represent genes as characters. This is not as general as possible, an issue mentioned earlier and addressed in the exercises. We add 97 to the generated integer, because the ASCII position of 'a' is 97. Consequently, we transform the generated integer to characters in the range ['a'..'z'].

Ranking the population, shown next, is very simple using the `sort` method that is part of the static class, `Collections`. A static class is one that exists to provide services to other classes. In this case, the methods in `Collections` operate on and return classes that implement the *Collection Interface*. An interface in Java is a set of specifications that implementing classes must fulfill. It would have been possible to design `GA` as an *Interface* class, though the presence of common methods among specific genetic algorithms made the choice of `GA` as a superclass a more intuitively clear design. Among the many classes that implement the methods specified in the *Collection* interface is `ArrayList`, the class we have chosen to represent the population of chromosomes.

```
protected void SortPop()
{
 Collections.sort(GA_pop, new CostComparator());
}
private class CostComparator
  implements Comparator <Chromosome>
{
  int result;
  public int compare(Chromosome obj1,
     Chromosome obj2)
  {
     result = new Integer(obj1.GetCost()).
        compareTo(new Integer(obj2.GetCost()));
        return result;
  }
}
```

`Collections.sort` requires two arguments, the object to be sorted—the `ArrayList` containing the population—and the mechanism that will do the sorting:

```
Collections.sort(GA_pop, new CostComparator());
```

The second argument creates an instance of a helper class that implements yet another interface class, this time the *Comparator* interface. The second object is sometimes called the *comparator object*. To implement the Comparator interface we must specify the type of the objects to be compared—class `Chromosome`, in this case—and implement its `compare` method. This method takes two chromosomes as arguments,

uses the method `GetCost` to extract the cost from the chromosome, and the `compareTo` method of the *Integer* wrapper class to determine which of the chromosomes costs more. In keeping with OO, we give no consideration to the specific algorithm that Java uses. Java documentation guarantees only that the Comparator class "imposes a total ordering on some collection of objects" (Interface Comparator 2007).

`Mutation` is the last of the three shared methods that we will consider. The fraction of the total number of genes that are to be mutated per generation is a design parameter. The fraction of genes mutated depends on the size of the population, the number of genes per chromosome, and the fraction of the total genes to mutate. For each of the mutations, we randomly choose a gene within a chromosome, and randomly choose a mutated value. There are two things to notice. First, we never mutate our best chromosome. Second, the mutation code in `GA` is specific to genetic algorithms where genes may be reasonably represented as characters. The code for `Mutation` may be found on the Chapter 28 code library.

28.4 Conclusion: Complex Problem Solving and Adaptation

In this chapter we have shown how Darwin's observations on speciation can be adapted to complex problem solving. The GA, like other AI techniques, is particularly suited to those problems where an optimal solution may be computationally intractable. Though the GA might stumble upon the optimal solution, odds are that computing is like nature in one respect. Solutions and individuals must be content with having solved the problem of adaptation only well enough to pass their characteristics into the next generation. The extended example, `WordGuess`, was a case in which the GA happens upon an exact solution. (See the code library for sample runs). This was chosen for ease of exposition. The exercises ask you to develop a GA solution to a known NP-Complete problem.

We have implemented the genetic algorithm using object-oriented programming techniques, because they lend themselves to capturing the generality of the GA. Java was chosen as the programming language, both because it is widely used and, because its syntax in the C/C++ tradition makes it readable to those with little Java or, even, OO experience.

As noted, we have not discussed the classes `SetParams`, `GetParams`, and `Parameters` mentioned in Section 28.3. These classes write to and read from a file of design parameters. The source code for them can be found in the auxiliary materials. Also included are instructions for using the parameter files, and instructions for exercising `WordGuess`.

Chapter 28 was jointly written with Paul De Palma, Professor of Computer Science at Gonzaga University.

Exercises

1. The traveling salesperson problem is especially good to exercise the GA, because it is possible to compute bounds for it. If the GA produces a solution that falls within these bounds, the solution, while probably not optimal, is reasonable. See Hoffman and Wolfe (1985) and Overbay, et al.

(2007) for details. The problem is easily stated. Given a collection of cities, with known distances between any two, a tour is a sequence of cities that defines a start city, C, visits every city once and returns to C. The optimal tour is the tour that covers the shortest distances. Develop a genetic algorithm solution for the traveling sales person problem. Create, at least, two new classes `TSP`, derived from `GA`, and `TSPtst` that sets the algorithm in motion. See comments on mating algorithms for the traveling salesperson problem in Luger (2009, Section 12.1.3).

2. Implement the Tournament pairing method of the class `Pair`. Tournament chooses a subset of chromosomes from the population. The most fit chromosome within this subset becomes Parent A. Do the same thing again, to find its mate, Parent B. Now you have a breeding pair. Continue this process until we have as many breeding pairs as we need. Tournament is described in detail in Haupt and Haupt (1998). Does `WordGuess` behave differently when Tournament is used?

3. As it stands, `GA` runs under command-line Unix/Linux. Use the `javax.swing` package to build a GUI that allows a user to set the parameters, run the program, and examine the results.

4. Transform the java application code into a java applet. This applet should allow a web-based user to choose the `GA` to run (either `WordGuess` or `TSP`), the pairing algorithm to run (Top-Down or Tournament), and to change the design parameters

5. `WordGuess` does not make use of the full generality provided by object-oriented programming techniques. A more general design would not represent genes as characters. One possibility is to provide several representational classes, all inheriting from a modified `GA` and all being super classes of specific genetic algorithm solutions. Thus we might have `CHAR_GA` inheriting from `GA` and `WordGuess` inheriting from `CHAR-GA`. Another possibility is to define chromosomes as collections of genes that are represented by variables of class Object. Using these, or other, approaches, modify `GA` so that it is more general.

6. Develop a two-point crossover method to be included in class `Mate`. For each breeding pair, randomly generate two crossover points. Parent A contributes its genes before the first crossover and after the second to Child A. It contributes its genes between the crossover points to Child B. Parent B does just the opposite. See Haupt and Haupt (1998) for still other possibilities.

29 Case Studies: Java Machine Learning Software Available on the Web

Chapter Objectives This chapter provides a number of sources for open source and free machine learning software on the web.

Chapter Contents 29.1 Java Machine Learning Software

29.1 Java Machine Learning Software

There are many popular java-based open-source machine learning software packages available on the internet. Several important and widely used ones are described below.

Weka Weka is a Java-based open-source software distributed under the GNU General Public License. It was developed at the University of Waikato in Hamilton, New Zealand in 1993.

Weka is a very popular machine learning software that is widely used for data-mining problems. The main algorithms implemented in Weka focus on pattern classification, regression and clustering. Tools for data preprocessing and data visualization are also provided. These algorithms can either be directly applied to a dataset or be called from other Java code. Weka algorithms can also be used as building blocks for implementing new machine learning techniques.

http://www.cs.waikato.ac.nz/ml/weka/

ABLE ABLE is a freely-available Java-toolkit for agent-based machine learning problems developed by the IBM T. J. Watson Research Center in Yorktown Heights, NY.

The ABLE framework provides a library of packages, classes and interfaces for implementing machine learning techniques like neural networks, Bayesian classifiers and decision trees. It also provides a Rule Language for rule-based inference using Boolean and fuzzy logic. The packages and classes can be extended for developing custom algorithms. Support is also provided for reading and writing text and database data, data transformation and scaling and invocation of user-defined external functions.

http://www.alphaworks.ibm.com/tech/able

JOONE JOONE (Java Object-Oriented Neural Engine) is a free java framework for implementing, training and testing machine learning algorithms using artificial neural networks (ANN). The software includes algorithms for feed-forward neural networks, recursive neural networks, time-delay neural networks, standard and resilient back propagation, Kohonen self-

organizing maps, Principal Component Analysis (PCA), and modular neural networks.

JOONE components can be plugged into other software packages and can be extended to design more sophisticated algorithms. It comes with a GUI editor to visually create and test any neural network and a Distributed Test Environment to train many neural networks in parallel and select the best one for a given problem.

http://www.jooneworld.com/

LIBSVM LIBSVM (Library for Support Vector Machines) is an integrated software solution for classification, regression and distribution-estimation using support vector machines (SVM) developed at the National Taiwan University. The source code is freely available with both C++ and Java versions.

The main features of the LIBSVM software are different SVM formulations, efficient multi-class classification, cross-validation for model selection, probability estimates, weighted SVM for unbalanced data and automatic model selection. It also includes a GUI and interfaces for other languages like Python, R, MATLAB, Perl, Ruby and Common Lisp.

http://www.csie.ntu.edu.tw/~cjlin/libsvm/

30 The Earley Parser: Dynamic Programming in Java

Chapter Objectives

Sentence parsing using dynamic programming
- Memoization of subparses
- Retaining partial solutions (parses) for reuse

The chart as medium for storage and reuse
- Indexes for word list (sentence)
- States reflect components of parse
- Dot reflects extent of parsing right hand side of grammar rule

Lists of states make up components of chart
- Chart linked to word list

Java Implementation of an Earley parser
- Context free parser
- Deterministic

Chart supports multiple parse trees
Forward development of chart composes components of successful parse
Backward search of chart produces possible parses of the sentence

Chapter Contents

30.1 Chart Parsing: An Introduction

The Earley parser (Earley 1970) uses dynamic programming to analyze strings of words. Traditional dynamic programming techniques (Luger 2009, Section 4.1) use an array to save (memoize) the partial solutions of a problem for use in the generation of subsequent partial solutions. In Earley parsing this array is called a *chart*, and thus this approach to parsing sentences is often called *chart parsing*.

In Chapter 9, Sections 1 and 2, we first presented the full algorithms along with the evolving chart for Earley parsing. In these sections, we presented pseudo-code, demonstrated the "dot" as a pointer indicating the current state of the parse for each grammar rule, and explicitly considered the state of the chart after each step of the algorithm. We refer to Sections 9.1, 9.2, and Luger (2009, Section 15.2.2) for these specific details if there is any concern about how the chart-parsing algorithm works. We feel that it is also interesting to compare the data representation and control structures used in the declarative Prolog environment, Chapter 9, with what we next present with the object-oriented representations of Java.

30.2 The Earley Parser: Components

We first discuss the data representations required by the Earley parsing algorithm presented in Sections 9.1 and 9.2. Of course, in the present chapter we will be using an object-oriented hierarchy to capture the components of the parser. Consider what the pseudo-code requires:

- **A sentence**. The sentence needs to be in a format that supports pointers to any word located in that sentence so that appropriate grammar rules can be applied.
- **A grammar**. The Earley parser needs a set of (context free) grammar rules that can be applied to interpret the components of the sentence. The parser itself has no knowledge of the parts of speech (POS) or production rules of the grammar.
- **An evolving chart**. The chart is used to save partial solutions (accepted parts of the parse) for later use. Thus, the chart is used to contain the states as they are produced during the algorithm. These states need to be stored in the order of their production and without repeats.
- **The states**. State will capture the current activity of the parser. Thus it will need to be a container for the current rule, which has a left-hand-side, `LHS`, and a right-hand-side, `RHS`. Besides instantiating a particular rule, state must also have the current `(i, j)` pair that presents the seen/unseen parts of the sentence for that rule.

We next consider how each of these constituents of the algorithm is represented as data structures in Java. In Section 30.3 we describe the Earley parser itself that will utilize these components.

The Sentence

First, we consider the set of descriptors that the sentence needs. The primary thing we require is the ability to index into specific word locations in the current sentence. This can be handled two ways: the use of a simple representation, or the use of a class. The simple representation would be a `String` array. This array would enable us easily to index to specific words in the sentence. Everything that we need for the algorithm is present.

We could also use a class to represent each indexed sentence. If `Sentence` is a class, we could incorporate other aspects of the sentence into that class, such as the segmentation of a `String` into individual words. If we were using the Earley algorithm in conjunction with another algorithm (which is often required), we may need to create the `Sentence` class so that we can separate the sentence's parsing from other code.

For this presentation, we use the simple representation of a sentence as a `String` array.

The Grammar

For the grammar rule processing required by the Earley parser we create a class. The application of each rule needs to know the specific rules of a grammar, and which non-terminals are parts of speech. So that both characteristics are easily contained, a `Grammar` class is a good choice. The `Grammar` class will need two important methods: `getRHS(String lhs)`, and `isPOS(String lhs)`:

`getRHS(String lhs)` will return all `RHS`'s of the grammar rules for any left-hand-side, `lhs`. If there are not any such rules, then it will indicate such.

`isPOS(String lhs)` will return `true` or `false` based on whether or not a component of the lhs is a part of speech.

To make the `Grammar` class easier to extend to more complicated grammars, the `Grammar` class itself does not instantiate any rules. It is a basic framework for the two important methods, and defines how the rules are contained and related. To create a specific grammar, the `Grammar` class needs to be extended to a new class. This new class will instantiate all the grammar rules. As just noted, the framework of the grammar rules is a mapping between a LHS and RHS. For each rule there will be only one LHS for each RHS, but it is likely in the full set of grammar rules that a particular LHS will have several possible RHSs. Later in the chapter, the exact framework for this matching is presented.

The Chart

A chart is an ordered list of the successively produced components of the parse. A major requirement is to determine whether any newly produced possible component of the parse is already contained in the chart.

To make it easier to maintain the charts correctly and consistently, we create a `Chart` class. We could have used a simpler structure, like `Vector,` to contain the states of the parse as they are produced, but the code to manipulate the `Vector` would then be distributed throughout the parser code. This dispersed code makes it much harder to make corrections, and to debug. When we create the `Chart` class, the code to manipulate the chart will be identical for all uses, and since the code is all in one place, it will be much easier to debug. Notice that the `Chart` class represents a single chart, not the evolving set of states that are created by the Earley algorithm. Since there is no additional functionality needed beside that already discussed, we make a `Chart` array for the evolving set of chart states, rather than making another class.

The States

A state component for the parser has one left-hand-side, LHS, one right-hand-side, RHS, for each rule that is instantiated, as well as indices from the sentence `String` array, an `(i j)` pair. Because these components all need to be represented, the easiest way to create the problem solving state, is to make a `State` class. Since the `State` class supports the full Earley algorithm, it will require `get` methods for returning the `LHS`, the `RHS`, and the `i j` indices. Also, as seen in the pseudo-code of Section 9.2, we need the ability to get the term after the dot in the `RHS`, as well as the ability to determine whether or not the dot is in the last (terminal) position. Methods to support these requirements must be provided.

Throughout our discussion, `LHS` and `RHS` have been mentioned, but not their implementation. Since the Earley parser uses context-free grammar rules, we create the `LHS` as a `String`. The `RHS` on the other hand, is a sequence of terms, which may or may not include a dot. Due to the fact that it is used in two separate classes, and the additional requirement of dot manipulation, we separated the RHS into its own class.

30.3 The Earley Parser: Java Code

The Earley parser, which manipulates the components described in Section 30.2, will have its own class. This makes it easier to contain and hide the details of the algorithm. The `EarleyParser` class can be implemented in either of two ways.

First, the class could be static. When one wanted to parse a sentence, the static method would be called, and with the grammar rules and the sentence to be parsed as arguments, return a boolean indicating whether the parse was successful. Alternatively, the chart itself could be returned and examined to determine whether there was a successful parse. Second, with the class not static, the `EarleyParser` would be instantiated with a `Grammar`, and then a parse method could be called with a sentence. After the parse method is called, another method would be called to obtain the charts (if the parse method returns a `boolean`). We take this second approach and start with the most basic class, the `RHS` class, and then work our way towards creating the `EarleyParser` class.

The RHS Class

The `RHS` is a `String` array containing a `boolean` that records whether its terms contains a dot, an `int` recording the offset of the dot (this will default to –1, indicating no dot), and a `final static String` containing the representation of the dot. The constructor determines if there is a dot and updates `hasDot` and `dot` accordingly.

```
public class RHS
{
    private String[] terms;
    private boolean hasDot = false;
    private int dot = -1;
    private final static String DOT = "@";
    public RHS (String[] t)
    {
         terms = t;
         for (int i=0; i< terms.length; i++)
         {
              if(terms[i].compareTo (DOT) == 0)
              {
                   dot = i;
                   hasDot = true;
                   break;
              }
         }
    }
                              // Additional methods defined below.
}
```

`RHS` returns its terms, the `String` array, for use by the `EarleyParser`, as well as the `String` prior to and after the dot. This enables ease of queries by the `EarleyParser` regarding the terms in the `RHS` of the grammar rule. For example,. `EarleyParser` gets the term following the dot from `RHS`, and queries the `Grammar` to determine if that term is a part of speech.

```
public String[] getTerms ()
{
    return terms;
}
public String getPriorToDot ()
{
    if(hasDot && dot >0)
        return terms[dot-1];
    return "";
}
public String getAfterDot ()
{
    if(hasDot && dot < terms.length-1)
        return terms[dot+1];
    return "";
}
```

The final procedures required to implement `RHS` are manipulation of and queries concerning the dot. The queries determine whether there is a dot, and where the dot is located, last or first. When a dot is moved or added to a `RHS`, a new `RHS` is returned. This is done because whenever a dot is moved a new `State` must be created for it.

```
public boolean hasDot ()
{
    return hasDot;
}
public boolean isDotLast ()
{
    if(hasDot)
        return (dot==terms.length-1);
    return false;
}
public boolean isDotFirst ()
{
    if(hasDot)
        return (dot==0);
    return false;
}
```

```
public RHS addDot ()
{
    String[] t = new String[terms.length+1];
    t[0] = DOT;
    for (int i=1; i< t.length; i++)
        t[i] = terms[i-1];
    return new RHS (t);
}
public RHS addDotLast ()
{
    String[] t = new String[terms.length+1];
    for (int i=0; i< t.length-1; i++)
        t[i] = terms[i];
    t[t.length-1] = DOT;
    return new RHS (t);
}
public RHS moveDot ()
{
    String[] t = new String[terms.length];
    for (int i=0; i< t.length; i++)
    {
        if (terms[i].compareTo (DOT)==0)
        {
            t[i] = terms[i+1];
            t[i+1] = DOT;
            i++;
        }
        else
            t[i] = terms[i];
    }
    return new RHS (t);
}
```

There are two additional methods that we have not included here. These are overrides methods of `equals(Object o)`, and `toString()`. Equivalence indicates identical terms, and placement of the dot. `toString()` is overridden to make it easier to print during debug, and when the charts are printed. Next we present one of the two classes that contain a `RHS`.

The Grammar Class

The `Grammar` class does not instantiate the rules of a specific grammar. It contains a `HashMap` that links the left-hand-side (`LHS`) of a grammar rule, which is a `String`, to an array of `RHS`s, and a `Vector` of `String`s that are the parts of speech of the grammar.

```
public class Grammar
{
    HashMap<String, RHS[]> Rules;
    Vector<String> POS;
    public Grammar ()
    {
        Rules = new HashMap<String, RHS[]>();
        POS = new Vector<String>();
    }
                        // Additional methods defined below.
}
```

The `Grammar` class supports two methods: one returning all the `RHS`s associated with a LHS, and the second returning if a `String` is a part of speech.

```
public RHS[]  getRHS (String lhs)
{
    RHS[] rhs = null;
    if(Rules.containsKey (lhs))
    {
        rhs = Rules.get (lhs);
    }
    return rhs;
}
public boolean isPartOfSpeech (String s)
{
    return POS.contains (s);
}
```

For `EarleyParser` to function, the `Grammar` class must be extended. To do this we have created `SimpleGrammar` that demonstrates both creation of the rules and how these are added to the rule list.

```
public class SimpleGrammar extends Grammar
{
    public SimpleGrammar ()
    {
        super();
        initialize();
    }
    private void initialize()
    {
        initRules();
        initPOS();
    }
```

```
    private void initRules()
    {
        String[] s1 = {"NP", "VP"};
        RHS[] sRHS = {new RHS(s1)};
        Rules.put ("S", sRHS);
        String[] np1 = {"NP","PP"};
        String[] np2 = {"Noun"};
        RHS[] npRHS = {new RHS(np1),
               new RHS(np2)};
        Rules.put ("NP", npRHS);
        String[] vp1 = {"Verb","NP"};
        String[] vp2 = {"VP", "PP"};
        RHS[] vpRHS = {new RHS(vp1),
               new RHS(vp2)};
        Rules.put ("VP", vpRHS);
        String[] pp1 = {"Prep","NP"};
        RHS[] ppRHS = {new RHS(pp1)};
        Rules.put ("PP", ppRHS);
        String[] noun1 = {"John"};
        String[] noun2 = {"Mary"};
        String[] noun3 = {"Denver"};
        RHS[] nounRHS = {new RHS(noun1),
                 new RHS(noun2),
                 new RHS(noun3)};
        Rules.put ("Noun", nounRHS);
        String[] verb = {"called"};
        RHS[] verbRHS = {new RHS(verb)};
        Rules.put ("Verb", verbRHS);
        String[] prep = {"from"};
        RHS[] prepRHS = {new RHS(prep)};
        Rules.put ("Prep", prepRHS);
    }
    private void initPOS()
    {
        POS.add ("Noun");
        POS.add ("Verb");
        POS.add ("Prep");
    }
}
```

The State class The `State` class contains a `String` representing the `LHS` of the rule, a `RHS` that contains the dotted right-hand-side of the rule, and `int`s describing Seen/UnSeen components. There are `get` methods, and functions for handling the dot.

```
public class State
{
    private String lhs;
    private RHS rhs;
    private int i,j;
    public State (String lhs, RHS rhs, int i, int j)
    {
        this.lhs = lhs;
        this.rhs = rhs;
        this.i = i;
        this.j = j;
    }
    public String getLHS ()
    {
        return lhs;
    }
    public RHS getRHS ()
    {
        return rhs;
    }
    public int getI ()
    {
        return i;
    }
    public int getJ ()
    {
        return j;
    }
    public String getPriorToDot ()
    {
        return rhs.getPriorToDot ();
    }
    public String getAfterDot ()
    {
        return rhs.getAfterDot ();
    }
    public boolean isDotLast ()
    {
        return rhs.isDotLast ();
    }
}
```

Finally, again, the function overrides of `equals(Object o)` and `toString()` are not included. Equivalent states are identified when the LHS, RHS, `i`, and `j` are all identical. `toString()` prints out the `State` in a readable format.

The Chart Class

The `Chart` class contains a `Vector` of `States`. These are the states produced by the `EarleyParser`. The `States` are inserted into the `Vector` in order; this is necessary for the algorithm.

```
public class Chart
{
    Vector<State> chart;
    public Chart ()
    {
        chart = new Vector<State>();
    }
    public void addState (State s)
    {
        if(!chart.contains (s))
        {
            chart.add (s);
        }
    }
    public State getState (int i)
    {
        if(i < 0 || i >= chart.size ())
            return null;
        return (State)chart.get (i);
    }
}
```

`addState(State s)` determines whether `s` is already within the `Chart`. If `s` is not in the `Chart`, `s` is added to the end of `Vector`. Nothing is done when `s` is already in the `Chart`. `getState(int i)` returns the `State` at the `i`-th offset in `Vector`. There are checks to enforce that `i` is a valid offset. `toString()` is overridden in `Chart`, in addition to a `get` function that returns the size of the `Chart`.

30.4 The Completed Parser

We have now completed the design of the components of the Earley parser. Figure 30.1 presents the object hierarchy that supports this design. `EarleyParser`, which implements this design is presented in Section 30.4.1, while Section 30.4.2 describes `main` which presents two sentences and produces their chart parses.

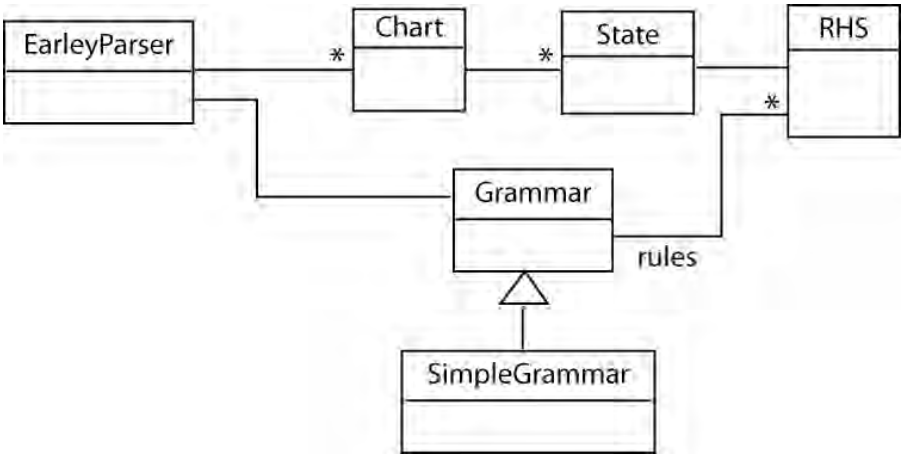

Figure 30.1 The design hierarchy for the EarleyParser class.

The EarleyParser Class

The `EarleyParser` class contains `Grammar` describing the grammar rules for parsing the perspective sentence. It also creates `String`, an array containing the sentence to be parsed, and `Chart`, an array containing the evolving states of the chart. `sentence` and `Chart` will change with each call of `parseSentence(…)`. Note that each of the methods of the EarleyParser class reflect the design components of Sections 30.2 and 30.3 and presented in Figure 30.1.

```
public class EarleyParser
{
    private Grammar grammar;
    private String[] sentence;
    private Chart[] charts;
    public EarleyParser (Grammar g)
    {
        grammar = g;
    }
    public Grammar getGrammar ()
    {
        return grammar;
    }
    public Chart[] getCharts ()
    {
        return charts;
    }
                    // Additional methods defined below.
}
```

`parseSentence(…)` takes the sentence to be parsed, uses it to initialize `Chart` to have the number of words in the sentence + 1 number of chart states, and makes the dummy start state `("$ → @ S", 0, 0)` the first chart state. `parseSentence` then iterates through each of

the charts, and for every `State` in a `Chart`, checks to determine which procedure is called next: `completer(...)` - the dot is last, `scanner(...)` - the term following the dot is a part of speech, or `predictor(...)` - the term following the dot is not a part of speech. After all charts are visited, if the last `State` added to the last `Chart` is a finish state, `("$ → S @", 0, sentenceLength)`, the sentence was successfully parsed.

```
public boolean parseSentence (String[] s)
{
    sentence = s;
    charts = new Chart[sentence.length+1];
    for (int i=0; i< charts.length; i++)
        charts[i] = new Chart ();
    String[] start1 = {"@", "S"};
    RHS startRHS = new RHS (start1);
    State start = new State ("$",startRHS,0,0,null);
    charts[0].addState (start);
    for (int i=0; i<charts.length; i++)
    {
        for (int j=0; j<charts[i].size (); j++)
        {
            State st = charts[i].getState (j);
            String next_term = st.getAfterDot ();
            if (st.isDotLast ())
                                // State's RHS = ... @
                completer (st);
            else
            if(grammar.isPartOfSpeech (next_term))
                    // RHS = ... @ A ..., where A is a part of speech.
                scanner (st);
            else
                predictor (st); // A is NOT a part of speech.
        }
    }
            // Determine whether a successful parse.
    String[] fin = {"S","@"};
    RHS finRHS = new RHS (fin);
    State finish = new State ("$",finRHS,
        0,sentence.length,null);
    State last = charts[sentence.length].getState
        (charts[sentence.length].size ()-1);
    return finish.equals (last);
}
```

We next create the `predictor`, `scanner`, and `completer` procedures. First, the `predictor(State s)` adds all rules for the term after the dot in `s` to the `j`-th slot in `chart`.

```
private void predictor (State s)
{
    String lhs = s.getAfterDot ();
    RHS[] rhs = grammar.getRHS (lhs);
    int j = s.getJ ();
    for (int i=0; i< rhs.length; i++)
    {
         State ns = new State (lhs, rhs[i].addDot (),
              j, j, s);
        charts[j].addState (ns);
    }
}
```

`scanner(State s)` determines whether the part of speech term following the dot in `s` has a `RHS` that contains only the `j`-th word in the sentence. If so, this new state is added to the `(j+1)`-th chart.

```
private void scanner (State s)
{
    String lhs = s.getAfterDot ();
    RHS[] rhs = grammar.getRHS (lhs);
    int i = s.getI ();
    int j = s.getJ ();
    for (int a=0; a< rhs.length; a++)
    {
        String[] terms = rhs[a].getTerms ();
        if (terms.length == 1 &&
             j < sentence.length &&
             terms[0].compareToIgnoreCase
              (sentence[j]) == 0)
         {
              State ns = new State (lhs,
                   rhs[a].addDotLast (), j, j+1, s);
              charts[j+1].addState (ns);
        }
    }
}
```

Finally, `completer(State s)` determines whether any state in the `i`-th chart slot has a term following the dot that is the same as the `LHS` of `s`. If so, new `States` based on those terms found are created with a moved dot, and an updated `j`.

```
private void completer (State s)
{
    String lhs = s.getLHS ();
    for (int a=0; a<charts[s.getI ()].size (); a++)
    {
         State st = charts[s.getI ()].getState (a);
         String after = st.getAfterDot ();
         if(after != null &&
              lhs.compareTo (after)==0)
         {
              State ns = new State (st.getLHS (),
                   st.getRHS ().moveDot (),
                   st.getI (), s.getJ (), s);
              charts[s.getJ ()].addState (ns);
         }
     }
 }
```

After `EarleyParser` completes `parseSentence(…)`, the `getCharts()` method is called. These charts in conjunction with the grammar rules can be used to determine the parse trees of the sentence. We discuss methods for doing this in Section 30.5.

The EarleyParser: A Test Run

We next create methods that contain sentences that test `EarleyParser`. In our example, the included `Main`, contains two sentences and then uses `SimpleGrammar` to parse them. It then prints out the sentences and the associated `Charts` for each.

```
public class Main
{
    public static void main (String[] args)
    {
         String[] sentence1 =
              {"John", "called", "Mary"};
         String[] sentence2 =
              {"John", "called", "Mary",
              "from", "Denver"};
         Grammar grammar = new SimpleGrammar ();
         EarleyParser parser =
              new EarleyParser (grammar);
         test (sentence1,parser);
         test (sentence2,parser);
    }
```

```
    static void test (String[] sent,
         EarleyParser parser)
    {
         StringBuffer out = new StringBuffer ();
         for (int i=0; i<sent.length-1;i++)
              out.append (sent[i]+" ");
         out.append (sent[sent.length-1]+".");
         String sentence = out.toString ();
         System.out.println (
              "\nSentence: \""+sentence+"\"");
         boolean successful =
              parser.parseSentence (sent);
         System.out.println (
              "Parse Successful:" + successful);
         Chart[] charts = parser.getCharts ();
         System.out.println ("");
         System.out.println (
              "Charts produced by the sentence
                   \""+sentence+"\"");
         for (int i=0; i<charts.length; i++)
         {
              System.out.println ("Chart "+ i + ":");
              System.out.println (charts[i]);
         }
    }
}
```

30.5 *Generating Parse Trees from Charts and Grammar Rules (Advanced Section)*

All the topics discussed, as well as all the data structures and algorithms we have designed to this point in Chapter 30, have been used to build a chart that indicates whether or not a set of grammar rules are sufficient for parsing a string of words. In this final section we present some ideas for extracting parse trees from the charts created and the grammar rules that supported them. Thus, we have completed the *forward* component of the *forward/backward* algorithm (Luger 2009, Section 4.1 and Section 15.2.2) used in dynamic programming. We now present some ideas for completing the *backward* component of the dynamic programming algorithm: how we can use the chart and set of grammar rules to extract parse trees. We consider this an advanced topic, and so will present only the main components of an algorithm and leave its design as an exercise. Included with the software is our implementation of the stack-based approach to this problem.

To begin, we must determine how each state in the chart is created. One method for accomplishing this is to list all the ways that each state of the

chart can be produced. Another approach is to record this information when each of the states of the chart is first produced. We will discuss and implement this second method. To record the sources of a `State` we can add to the `State` class a `Vector` of `States`. This `Vector` will contain all `States` that produced this `State`. To maintain this, when a `State` is added to a `Chart`, if the `State` is already within the `Chart`, merge the sources of the `State` within the `Chart` and the one we attempted to add. This approach offers a method to look back through the charts quickly to find the possible parse trees.

It is important to remember that more then one parse tree can often be produced from `Chart`. An example is that the sentence "John called Mary from Denver" has two interpretations (parses): John called Mary, who is from Denver; and John called Mary, and John is in Denver. So however we produce the parse trees, we need to decide if we will attempt to produce all trees, or select only one. With appropriate forethought, it is easy to produce all parse trees.

To generate parse trees, the backward component of the dynamic programming algorithm, we begin with the final state `("$ → S @", 0, sentenceLength+1)`. For each source state for that final state, we iterate through the following:

1. If the source state is the start state `("$ → @ S", 0, 0)`, return the tree created. Otherwise continue.
2. Look at the current state we are evaluating. If the dot is last, then add `LHS` to the tree as the left-most child of the current evaluating node. We add it as the left-most child because we are building the tree right to left. This means that we will find the state with the last word of the sentence before any other preceding words. We will find the right-most child first, and want the subsequent children to be added to the left.
3. If the state's `LHS` is a part of speech, `POS`, then add the `RHS`'s first term as the only child of the node we just added. We are guaranteed that the `LHS` was just added as a child, because for this type of state there is a single term in the `RHS` and a final dot. For example: "`Noun → John @`". We have finished evaluating this state, so move to its source state and continue evaluation. There are two cases for this:
    a. There is only one source. Easy! Use that one source.
    b. There are multiple sources. Now we need to find the one that matches how we have been building the tree. To demonstrate what we mean, consider the more complex sentence "Old men and women like dogs." The ambiguity for this sentence is: only the men are old, or if both the men and women are old. "and" is a conjunction, which is a part of speech so it would match this rule. Here is the problem:

With 3b. we have finished with the state: `("Conj → and @", 2, 3)`. The sources of this state are both `("NP → NP @ Conj NP",`

`0, 2)` and `("NP → NP @ Conj NP", 1, 2)`. Which one should we use? First, notice that the only difference between the two source states, is the location of `i`. (The `i` describes where the start of each rule is.) In `("NP → NP @ Conj NP", 0, 2)`, the start is at the beginning of the sentence. For `("NP → NP @ Conj NP", 1, 2)`, the start is "men". Thus the parse difference between "old (men) and women" and "old (men and women)". Furthermore, we need to consider where the previous states we have used to make the parse tree are looking. If for this particular tree, we had used `("NP → NP Conj @ NP", 0, 3)` then we need to use `("NP → NP @ Conj NP", 0, 2)`. For `("NP → NP Conj @ NP", 1, 3)` we would use `("NP → NP @ Conj NP", 1, 2)`. Therefore we must find the source state that matches the rule ignoring the position of the dot (this should be off by one), and the i.

4. At this point the current state may have been added to the tree, or it may not have. In either case we need to determine whether there are multiple sources for this state, and if there are, we need to iterate across each of them to determine which of the sources are valid. Before we do this, we need to update the current evaluating node. Above, we mentioned a current evaluating node. This is the node we are adding children to. Before we can continue, we need to determine if this node needs to change for the next iteration. There are three cases:
    a. The current state's dot is first. This means there are no more children that need to be added to this node. So we move to the parent of the current evaluating node.
    b. If the current state's `LHS` was just added to the tree and it was not a part of speech, then we will want to be adding the next nodes to the node we just added. So the current evaluating node moves to its left-most child.
    c. Otherwise, continue to use this node. This happens if we have just added a part of speech, `POS`, node and its child.
5. Next, we must iterate through all of these sources, and for each source state that meets the criteria, we attempt to continue building the tree from that state. One of the following must be true for this continuation to be accomplished:
    a. The source state's `LHS` is equal to the current state's term prior to the dot. Remember, we are moving from right to left, and the dot is moving from right to left. So if this is true, then the current state was generated because the source state completed a rule (the dot moved all the way to the right) and the `completer(…)` method was called.
    b. The source state's `RHS`, with dot moved to the right, and `LHS` matches on a state we have already evaluated.

6. If the criteria fail for all source states, then this was a dead end, and no tree is returned. If any of the source states are valid, start at step 1 with that state as the current state, and update the current evaluating node. The accepted trees (there may be no tree possible) are bundled together and returned.

From this algorithm, we can produce the multiple parse trees implicit in the Earley algorithm's successful production of the `Chart`. Example code implementing this algorithm is included with the Chapter 30 support materials.

The Earley parser code as well as the first draft of this chapter was written by Ms Breanna Ammons, MS in Computer Science, University of New Mexico.

Exercises

1. Describe the role of the dot within the right hand side of the grammar rules processed by the Earley parser. How is the location of the dot changed as the parse proceeds? What does it mean that the same right hand side of a grammar rule can have dots at different locations?

2. In the Earley parser the input word list and the states in the state lists have indices that are related. Explain how the indices for the states of the state list are created.

3. Describe in your own words the roles of the `predictor`, `completer`, and `scanner` procedures in the algorithm for Earley parsing. What order are these procedures called in when parsing a sentence, and why is that ordering important? Explain your answers to the order of procedure invocation in detail.

4. Use the Java parser to consider the sentence "John saw the burglar with the telescope". Parse also "Old men and women like dogs". Comment on the different parses possible from these sentences and how you might retrieve them from the chart.

5. Create a `Sentence` class. Have one of the constructors take a `String`, and have it then separate the `String` into words.

6. Code the algorithm for production of parse trees from the completed `Chart`. One method of recording the sources is presented in Section 30.5. You may find it useful to use a stack to keep track of the states you have evaluated for the parse tree.

7. Extend `EarleyParser` to include support for context-sensitive (Luger 2009, Section 15.9.5) grammar rules. What new structures are necessary to guarantee constraints across subtrees of the parse?

31 Case Studies: Java Natural Language Tools Available on the Web

Chapter Objectives

This chapter provides a number of sources for open source and free atural language understanding software on the web.

Chapter Contents

31.1 Java Natural Language Processing Software

There are several popular java-based open-source natural language understanding software packages available on the internet. We describe three of these in Chapter 31. It must be remembered both that these web sites may change over time and that new sites will appear focusing on problems in NLP.

31.2 LingPipe from the University of Pennsylvania

Background

LingPipe is an available Java resource from Alias-I, http://www.alias-i.com/lingpipe/. Alias-i began in 1995 as a collaboration of students at University of Pennsylvania. After competing in different events (including DARPA MUC-6), the group was awarded a research contract under the TIDES (Trans-Lingual Information Detection Extraction and Summarization) program. Starting as Baldwin Language Technologies, the company's name later changed to Alias-i. LingPipe was used in two of Alias-i's products, FactTracker and ThreatTracker. In 2003, LingPipe was released as open source software with commercial licenses available as well. LingPipe contains many tools for linguistic analysis of human language, including tools for sentence-boundary detection; and a part-of-speech tagger and phrase chunker.

LingPipe is easy to download from the website. The download contains demos, documentation and there are models available to download as well. On the website there are tutorials, documentation, and a FAQ. Also there are links to the community of LingPipe consumers. This includes a listing of some commercial customers, as well as research patrons. There is a newsgroup for discussion as well as a blog for being kept up to date on the suite.

We next take a look at some of the tools provided by LingPipe.

Sentence-boundary Detection

To start with, there are tutorials contained in the download for the sentence-boundary detection classes. These tutorials contain example programs that use and extend the *com.aliasi.sentences* classes. If you follow

the tutorials, you will get suggestions for how some of the classes can be extended to do sentence detection for other corpora.

The *AbstractSentenceModel* class contains the basic functionality needed to detect sentences. When extending this class, definitions of possible stops, impossible penultimates, and impossible starts are needed. Possible stops are any token that can be placed at the end of a sentence. This includes '.' and '?'. Impossible penultimates are tokents that cannot precede an end of the sentence. An example would be 'Mr' or 'Dr'. Impossible starts are normally punctuation that should not start a sentence and should be associated with the end of the last sentence. These can be things like end quotes.

The *AbstractSentenceModel* is already extended to the *HeuristicSentenceModel*, which is extended to the *IndoEuropeanSentenceModel*, and the *MedlineSentenceModel*. These last two provide good examples of definitions for the possible stops, impossible penultimates and the impossible starts. From these examples, *HeuristicSentenceModel* can be extended to suit a data set. After creating an example set with known sentence boundaries, running the evaluator contained in the tutorials for the sentences class gives an idea of fallacies of the current model. From the evaluator's output files, corrections can be made to the possible stops, impossible penultimates and impossible starts. Be careful though; when attempting to remove all false positives and all false negatives, the definitions can be come too rigid and cause more errors when run with more then the example set. So try to find a good balance.

Within the download, the *AbstractSentenceModel* is only extended to the *HeuristicSentenceModel*. This does not mean that you must use the *HeuristicSentenceModel*. The *HeuristicSentenceModel* can be used as an example to create a new class that extends only *AbstractSentenceModel*. Therefore if you have a different type of model that you would like to use, extend *AbstractSentenceModel* and try it out.

Part-of-speech Tagger

The part-of-speech (POS) tagger is a little more complicated then the *sentences* classes. To use it the POS tagger must be trained. After it is trained, the tagger can be used to produce a couple of different statistics about its confidence of the tags it applies to input. In the download there are examples of code for the Brown, Genia and MedPost corpora. The classes used in making a POS tagger come from the *com.aliasi.hmm* package. The tagger is a *HmmDecoder* defined by a *HiddenMarkovModel*.

To train a tagger, you need first a corpus or test set that has been tagged. Using this tagged set, the *HmmCharLmEstimator* (in the *com.aliasi.hmm* package) can read the training set and create a *HiddenMarkovModel*. This model can be used immediately, or it can be written out to file and used at a later time. The file can be useful when evaluating different taggers. For each test on different corpora, the exact tagger can be used without having to recreate it each time.

Now that we have a tagger, we can use it to tag input. Within one *HmmDecoder* there are a couple of different ways to tag; all are methods of the decoder you create from the *HiddenMarkovModel*. Based on what kind of information you need, the options are first-best, n-best and confidence-

based. First-best returns only the "best" tagging of the input. N-best returns the first n "best" taggings. Confidence-based results are the entire lattice of forward/backward scores.

Provided in the tutorials is an evaluator of taggers. This uses pre-tagged corpora and trains a little, then evaluates a little. It parses reference taggings, uses the model to tag, and evaluates how well the model did. The reference tags are then added to the training data, and the parser moves on. The arguments to the evaluator will determine how well the model learns and how long it will take. Experiment with this package to see what is appropriate for your own data set.

A tagger produced by this package could be used in other algorithms. Whether as tags needed for the algorithm or as a source to produce a grammar, this POS tagger is useful. A future project might be to create a parse tree from the POS tagger, but that functionality is not within LingPipe. An exercise might be to extend LingPipe to create parse trees.

31.3 The Stanford Natural Language Processing Group

The Stanford NLP group is a team of faculty, postdoctoral researchers, graduate, and undergraduate students, members from both the Computer Science and Linguistics departments. The site http://nlp.stanford.edu describes the team members, their publications, and the libraries that can be downloaded.

Exploring this Stanford website, the reader finds, as of January 2008, six Java libraries available for work in natural language processing. These include a parser as well as a part-of-speech tagger. Although the information contained in the introduction for each package is extensive and contains a set of "frequently asked questions", the code documentation is often sparse without sufficient design documentation. The libraries are licensed under the full GNU Public License, which means that they are available for research or free software projects.

The Stanford Parser

The parser makes up a major component of the Stanford NLP website. There is background information for the parser, an on-line demo, and answers for frequently asked questions. The Stanford group refers to their parser as "a program that works out the grammatical structure of sentences". The software package includes an optimized probabilistic context-free grammar (Luger 2009, Section 15.4).

Within the download of the Stanford parser is a package called edu.stanford.nlp.parser. This parser interface contains two functions: One function determines whether the input sentence can be parsed or not. The other function determines whether the parse meets particular parsing goals, for example, that it is a noun phrase followed by a verb phrase sentence. There are also a number of sub-interfaces provided, including *ViterbiParser*, see Chapter 30, and *KBestViterbiParser*, the first supporting the most likely probabilistic parse of the sentence and the second giving the *K* best parses, where all parses are returned with their "scores".

Within the interface *edu.stanford.nlp.parser* there is a further interface, edu.stanford.nlp.parser.lexparser, which supports parsers for English,

German, Chinese, and Arabic expressions. There are also classes, that once implemented, can be used to train the parser for use on other languages. To train the parser, a training set needs to include systematically annotated data, specifically in the form of a Treebank (a corpus of annotated data that explicitly specifies parse trees). Once trained the parser contains three components: grammars, a lexicon, and a set of option values. The grammar itself consists of two parts, unary (NP = N) and binary (S = NP VP) rewrite rules. The lexicon is a list of lexical (word) entries) preceeded by a keyword followed by a raw count score. The options are persistent variable-value pairs that remain constant across the training and parsing stages.

The Stanford tools also include a GUI for easy visualization of the trees produced through parsing the input streams. The training stages require much more time and memory than using the already trained parser. Thus, for experimental purposes, it is convenient to use the already trained parsers, although there is much that can be learned by stepping through the creation of a parser.

Named-Entity Recognition

A program that performs *named–entity recognition* (NER) locates and classifies elements of input strings (text) into predefined categories. For the Stanford NER the primary categories for classification are *name*, *organization*, *location*, and *miscellaneous*. There are two recognizers, the first classifying the first three of these categories and trained on a corpus created from data from conference proceedings. The second is trained on more specific data, the proceedings from one conference.

Using the text classifiers is straightforward. They can be run either as embedded in a larger program or by command line. When run as part of a program the classifier is read in using a function associated with *CRFClassifier*. This function returns an *AbstractSequenceClassifier* that uses methods to classify the contents of a *String*. An example of one (of the three possible) output formats, called /Cat is: My/O name/O is/O Bill/PERSON Smith/PERSON ./O. /O indicates that the text string is not recognized as a named-entity. There are a number of issues involved in this type classification, for example that at this point Bill Smith is not recognized as the name of a single person but rather as two consecutive PERSON tokens. When working with this type pattern matching it is important to monitor issues in *over-learning* and *under-learning*: when one pattern matching component is created to fit a complex problem situation, another set of patterns may not then be classified properly.

Unfortunately the documentation for the Named-Entity package is minimal. Although it contains a set of JavaDocs they can be both wrong (referring to classes that are not included) or simply unhelpful.

31.4 Sun's Speech API

To this point we have focused on Java-based natural language processing tools analyzing written language. Speech recognition and synthesis are also important components of NLP. To assist developers in these areas Sun Microsystems has created an API for speech. This Java API can be found at http://research.sun.com/speech. From this page there is also a link to a

free speech recognizer developed using this API at Carnegie Mellon University, as well as a speech written by Sun that is based on *Flite*, a speech synthesis engine also developed at Carnegie Mellon University. To run programs written in the Java Speech API needs a compliant speech recognizer and synthesizer, audio hardware for output and a microphone for input.

The API contains three packages: speech, speech recognition, and speech synthesis. The speech package contains several packages and interfaces used by both the recognition and synthesis systems. The main interface is an *Engine* that is the parent interface for all speech systems. The engine contains the procedures for communicating with other classes as well as allocation/deallocation methods for moving between states. These states determine whether the engine has acquired resources sufficient for executing a function. The engine also provides methods to pause and resume play and to access all properties including, listeners, audio, and vocabulary managers. The speech class also contains procedures for listeners as well as a Word class that contains the written and spoken pronunciation forms for words. The collection of words, the vocabulary, is controlled by the *Vocabulary* manager.

The main class of the speech recognizer is *Recognizer*. An instance of recognizer creates listener events and passes them to all registered event listeners much the same way as action listeners work for GUI applications. The events are either accepted or rejected based on sets of grammar rules. There are two forms of grammar rules: rule-based and dictation. Dictation rules offer fewer constraints on what can be said with a resulting higher cost in computational resources an often lesser quality results. Rule-based grammars are constrained to the Java speech grammar format (JSGF) and as a result impose a greater constraint on the recognizer. They also require fewer resources with a reasonable freedom for expressions. A tutorial for the JSGF is located at http://java.sun.com/products/java-media/speech/forDevelopers/JSGF and can be used to create grammars.

Once a grammar is created it is passed to a recognizer and activated. Then the recognizer begins processing and sending out events to all registered listeners. Sample applications are linked to the previously mentioned web site.

The Java speech API also contains a package for synthesis. Analogous to the recognizer, the synthesizer package contains a *Synthesis* class. The synthesizer is able to speak instances of the *Speakable* class in a voice constructed by an instance of the *Voice* class. This class contains both make and female, as well as young, middle-aged, and older voices. It's only task is to output a "speakable" text, as defined by a Java speech markup language (JSML) specification. These specifications can be found at http://java.sun.com/products/java-media/speech/forDevelopers/JSML/.

Demonstrations of the Sun speech synthesizer are available at http://fretts.sourceforge.net/docs/index.php. where the source code can be downloaded. The Sun speech API comes with a wealth of documentation and example source code. Which is fairly transparent and easy to follow.

There are a number of other web-based sources that support tasks in natural language text and speech understanding. These range from phoneme capture, the development of word and language models using probabilistic finite-state acceptors and various forms of Markov models. There are also parsers and recognizers of sentence structures as well as more examples of speech recognizers and synthesizers. There are also a number of tools available for speech to text conversion. Besides tools in Java, many also exist in other languages including C++ and Python.

PART V: Conclusion: Model Building and the Master Programmer

The limits of my language mean the limits of my world…

— Ludwig Wittgenstein, "Tractatus Logico-Philosophicus"

Theories are like nets: He who casts, captures…

— Ludwig Wittgenstein, "Tractatus Logico-Philosophicus"

The best you can do by Friday is a form of the best you can do…

— Charles Eames, Noted Twentieth Century Designer

We have come to the end of our task! In Part V we will give a brief summary of our views of computer language use, especially in a comparative setting where we have been able to compare and contrast the idioms of three different language paradigms and their use in building structures and strategies for complex problem solving. We begin Chapter 32 with a brief review of these paradigm differences, and then follow with summary comments on paradigm based abstractions and idioms.

But first we briefly review the nature of the programming enterprise and why we are part of it.

Well, first, we might say that programming offers monetary compensation to ourselves and our dependents. But this isn't really why most of us got into our field. We authors got into this profession because computation offered us a critical medium for exploring and understanding our world. And, yes, we mean this in the large sense where computational tools are seen as epistemological artifacts for comprehending our world and ourselves.

We see computation as Galileo might have seen his telescope, as a medium for exploring entities, relationships, and invariance's never before perceived by the human agent. It took Newton and his "laws of motion" almost another century fully to capture Galileo's insights. We visualize computation from exactly this viewpoint, where even as part of our own and our colleagues' small research footprint we have explored complex human phenomena including:

- Human subjects' neural state and connectivity, using human testing, fMRI scanning, coupled with dynamic Bayesian networks and MCMC sampling, none of which would be possible without computation.

- Patterns of expressed genes as components of the human genome. These gene expression patterns are assumed to be at the core of protein creation that enables and supports much of the human animal's metabolic system, including cortical activity and communication.
- Real time diagnostics and prognostics on human and mechanical systems. These complex tasks often require various forms of hidden Markov models along with other stochastic tools and languages.
- Understanding human language and voiced speech also requires computational tools, including various stochastic tools and models. Better language tools will require conditioning such systems with realistic models of human understanding and intention.

Of course this list could go on to include many of the exciting tasks that make up the daily challenges of our readers. What is important is that we see computer programming less in terms of the act of building tools, than as a medium for creating and debugging models of the world – as an epistemological medium.

We feel that there are (at least) two consequences of our thinking of computation as an epistemological medium: First, as programmers we are model builders. We use our data structures and search strategies to capture state, relations, and invariance's in our application domains. We come to understand this domain through progressive approximation. And our domains are rarely static, but change and evolve across time. Thus we often require stochastic engines and probabilistic relationships to capture these complex evolving phenomena.

Second, we explore our world by iterative approximation. When we build a model, we make an approximation of some aspect of reality. The quality of our model building is often seen through the lens of failure. As the philosophers of science continue to remind us, good models are falsifiable. It is through their failure points that we begin to appreciate our own failure to comprehend aspects of the phenomena we wish to understand. When our models are carefully designed and crafted, we can then deconstruct them to address these failure points and attempt to expand our understanding. Our increased understanding is then reflected in the next iteration of our model building. Thus the iterative design methodology, whether used by the individual programmer, or as is more often the case, within the collaborating communities of groups of programmers is a critical methodology in coming to understand our application domains.

We urge the reader to keep these ideas in mind in reading the final chapter and its reprise of the book's main themes of language-paradigm-based abstractions and idioms of the master programmer.

32 Conclusion: The Master Programmer

Chapter Objectives This chapter provides a summary and discussion of the primary idioms and design patterns presented in our book.

Chapter Contents

32.1 Language Paradigm-Based Abstractions and Idioms

In the Introduction to this book, we stated that we wanted to do more than simply demonstrate the implementation of key AI algorithms in some of the major languages used in the field. We also wanted to explore the ways that the problems we try to solve, the programming languages we create to help in their solution, and the patterns and idioms that arise in the practice of AI programming have shaped each other. We will conclude with a few observations on these themes.

More than anything else, the history of programming languages is a history of increasingly powerful, ever more diverse abstraction mechanisms. Lisp, the oldest of the languages we have explored, remains one of the most dramatic examples of this progression. Although procedural in nature, Lisp was arguably the first to abstract procedural programming from such patterns as explicit branching, common memory blocks, parameter passing by reference, pointer arithmetic, global scoping of functions and variables, and other structures that more or less reflect the underlying machine architecture. By adopting a model based on the theory of recursive functions, Lisp provides programmers with a cleaner semantics, including recursive control structures, principled variable scoping mechanisms, and a variety of structures for implementing symbolic data structures.

Like Lisp, Prolog bases its abstraction on a mathematical theory: in this case, formal logic and resolution theorem proving. This allows Prolog to abstract out procedural semantics almost completely (the left to right handling of goals and such pragmatic mechanisms as the cut are necessary exceptions). The result is a declarative semantics that allows programmers to view programs as sets of constraints on problem solutions. Also, because grammars naturally take the form of rules, Prolog has not only proven its value in natural language processing applications, as well as a tool for manipulating formal languages, such as compilers or interpreters.

Drawing in part on the lessons of these earlier languages, object-oriented languages, such as Java, offer an extremely rich set of abstractions that support the idea of organizing even the most ordinary program as a model

of its application domain. These abstractions include class definitions, inheritance, abstract classes, interfaces, packages, overriding of methods, and generic collections. In particular, it is interesting to note the close historical relationship between Lisp and the development of object languages. Although Smalltalk was the first "pure" object-oriented language, it was closely followed by many object-oriented Lisp dialects. This relationship is natural, since Lisp laid a foundation for object-orientation through such features as the ability to manipulate functions as s-expressions, and the control over evaluation it gives the programmer. Java has continued this development, and is particularly notable for providing powerful software engineering support through development environments such as Eclipse, and the large number of packages it provides for user data structures, network programming, user interface implementation, web-based implementation, Artificial Intelligence, and other aspects of application development.

In addition to – or perhaps because of – their underlying semantic models, all these languages support more general forms of abstraction. The organization of programs around abstract data types, "bundles" of data structures and operations on them, is a common device used by good programmers – no matter what language they are using. Meta-linguistic abstraction is another technique that is particularly important to Artificial Intelligence programming. The complexity of AI problems clearly requires powerful forms of problem decomposition, but the ill-formed nature of many research problems defies such common techniques as top-down decomposition. Meta-linguistic abstraction addresses this conundrum by enabling programmers to design languages that are tailored to solving specific problems. It tames hard problems by abstracting their key features into a meta language, rather than decomposing them into parts. The general search algorithms, expert system shells, learning frameworks, semantic networks, and other techniques illustrated in this book are all examples of meta-linguistic abstraction.

This diversity of abstraction mechanisms across languages underlies a central theme of this book: the relationship between programming languages and the idioms of their use. Each language suggests a set of natural ways of achieving common programming tasks. These are refined through practice and shared throughout the programmer community through examples, mentoring, conferences, books, and all the mechanisms through which any language idiom spreads. Lisp's use of lists and CAR/CDR recursion to construct complex data structures is one of that language's central idioms; indeed, it is almost emblematic of the language. Similarly, the use of rule ordering in Prolog, with non-recursive terminating statements preceding recursive rules appearing throughout Prolog programs is on of that language's key idioms. Object-oriented languages rely upon a particularly rich set of idioms and underscore the importance of understanding and using them properly.

Java, for example, adopted the C programming language syntax to improve its learnability and readability (whether or not this was good idea continues to be passionately debated). It would be possible for a programmer to write Java programs that consisted of a single class with a static main method

that called additional main methods in the class. This program might function correctly, but it would hardly be considered a good java program. Instead, quality Java programs distribute their functionality over relatively large numbers of class definitions, organized into hierarchies by inheritance, interface definitions, method overloading, etc. The goal is to reflect the structure of the problem in the implementation of its solution. This not only brings into focus the use of programming languages to sharpen our thinking by building epistemological models of a problem domain, but also supports communication among developers and with customers by letting people draw on their understanding of the domain.

There are many reasons for the importance of idioms to good programming. Perhaps the most obvious is that the idiomatic patterns of language use have evolved to help with the various activities in the software lifecycle, from program design through maintenance. Adhering to them is important to gaining the full benefits of the language. For example, our hypothetical "Java written as C" program would lack the maintainability of a well-written Java program.

A further reason for adhering to accepted language idioms is for communication. As we will discuss below, software development (at least once we move beyond toy programs) is a fundamentally social activity. It is not enough for our programs to be correct. We also want other programmers to be able to read them, understand the reasons we wrote the program as we did, and ultimately modify our code without adding bugs due to a misunderstanding of our original intent.

Throughout the book, we have tried to communicate these idioms, and suggested that mastering them, along with the traditional algorithms, data structures, and languages, is an essential component of programming skill.

32.2 Programming as a Tool for Exploring Problem Domains

Idioms are also bound up – along with the related concept of design patterns, also discussed below – with an idea we introduced in the book's introduction: programming languages as tools for thinking. In the early stages of learning to program, the greatest challenges facing the student are in translating a software requirement, usually a homework assignment, into a program that works correctly. As we move into professional-level research or software development, this changes. We are seldom given clear, stable problem statements; rather, our job is to interpret a vague customer need or research goal and project it into a program that meets our needs. The languages we have addressed in this book are the product of many person-decades of theoretical development, experience, and insight. They are not only tools for programming computers, but also for refining our understanding of problems and their solution.

Illustrating this idea of programming languages as tools for thinking has been one of our primary goals in writing this book. Lisp is the oldest, and still one of the best, examples of this. The s-expression syntax is ideally suited for constructing symbolic data structures, and, along with the basic cons/car/cdr operations, provides an elegant foundation for structures as

diverse as lists, trees, frames, networks, and other types of knowledge representation common to Artificial Intelligence. A search of early AI literature shows the power of s-expressions as both a basis for symbolic computing and for communication of theoretical ideas: numerous articles on knowledge representation, learning, reasoning, and other topics use s-expressions to state theoretical ideas as natural science uses algebra.

Prolog continues this tradition with its use of logical representation and declarative semantics. Logic is the classic "tool for thinking," giving a mathematical foundation to the disciplines of clarity, validity, and proof. Subtler is the idea of declarative semantics, of stating constraints on a problem solution independently of the procedural steps used to realize those constraints. This brings a number of benefits. Prolog programs are notoriously concise, since the mechanisms of procedural computing are abstracted out of the logical statement of problem constraints. This concision helps give clear formulation to the complex problems faced in AI programming. Natural language understanding programs are the most obvious example of this, but we also call the reader's attention to the relative ease of writing meta-interpreters in Prolog. This discipline of meta-linguistic abstraction is a quintessential way a language assists in our thinking about hard problems.

Java's core disciplines of encapsulation, inheritance, and method extension also reflect a heritage of AI thinking. As a tool for thinking, Java brings these powerful disciplines to problem decomposition and representation, metalinguistic abstraction, incremental prototyping, and other forms of problem solving. An interesting example of the subtle influence object-oriented programming has on our thinking can be found in comparing the declarative semantics of Prolog with the static structure of an object-oriented program.

Although we have no "hard" data to prove this, our work as both engineers and teachers has convinced us that the more experienced a Java programmer becomes, the more classes and interfaces we find in their programs. Novice programmers seem to favor fewer classes with longer methods, most likely because they lack the rich language of idioms and patterns used by skilled object-oriented designers. Breaking a program down into a larger number of objects brings several obvious benefits, including ease of debugging and validating code, and enhanced reuse. Another benefit of this is a shift of program semantics from procedural code to the static structure of objects and relations in the class structure. For example, a well-designed class hierarchy with the use of overloaded methods can eliminate many if-then tests in the program: the class "knows" which method to use without an explicit test. For this reason, Java programmers frown on the use of operators like `instanceof` to test explicitly for class membership: the object should exploit inheritance to call the proper method rather than use such tests.

The analogy of this to Prolog's declarative semantics is useful: both techniques move program semantics from dynamic execution to static structure. The static structure of objects or assertions can be understood by inspection of code, rather than by stepping through executions. It can be

analyzed and verified in terms of things and relations, rather than the complexities of analyzing the many paths a program can take through its execution. And, it enhances the use of the programming language as a tool for stating theoretical ideas: as a tool for thinking.

32.3 Programming as a Social Activity

As programming has matured as a discipline, we have also come to recognize that teams usually write complex software, rather than a single genius laboring in isolation. Both authors work in research institutions, and are acutely aware that the complexity of the problems modern computer science tackles makes the lone genius the exception, rather than the rule. The most dramatic example of this is open-source software, which is built by numerous programmers laboring around the world. To support this, we must recognize that we are writing programs as much to be read by other engineers as to be executed on a computer.

Software Engineering and AI

This social dimension of programming is most strongly evident in the discipline of software engineering. We feel it unfortunate that many textbooks on software engineering emphasize the formal aspects of documentation, program design, source code control and versioning, testing, prototyping, release management, and similar engineering practices, and downplay the basic source of their value: to insure efficient, clear communication across a software development team.

Both of this book's authors work in research institutions, and have encountered the mindset that research programming does not require the same levels of engineering as applications development. Although research programming may not involve the need for tutorials, user manuals and other artifacts of importance to commercial software, we should not forget that the goal of software engineering is to insure communication. Research teams require this kind of coordination as much as commercial development groups. In our own practice, we have found considerable success with a communications-focused approach to software engineering, treating documentation, tests, versioning, and other artifacts as tools to communicate with our team and the larger community. Thinking of software engineering in these terms allows us to take a "lightweight" approach that emphasizes the use of software engineering techniques for communication and coordination within the research team. We urge the programmer to see their own software engineering skills in this light.

Prototyping

Prototyping is an example of a software engineering practice that has its roots in the demands of research, and that has found its way into commercial development. In the early days, software engineering seemed to aim at "getting it right the first time" through careful specification and validation of requirements. This is seldom possible in research environments where the complexity and novelty of problems and the use of programming as a tool for thinking precludes such perfection. Interestingly, as applications development has moved into interactive domains that must blend into the complex communication acts of human communities, the goal of "getting it right the first time" has been rejected in favor of a prototyping approach.

We urge the reader to look at the patterns and techniques presented in this book as tools for building programs quickly and in ways that make their semantics clear – as tools for prototyping. Metalinguistic abstraction is the most obvious example of this. In building complex, knowledge-based systems, the separation of inference engine and knowledge illustrated in many examples of this book allows the programmer to focus on representing problem-specific knowledge in the development process.

Similarly, in object-oriented programming, the mechanisms of interfaces, class inheritance, method extension, encapsulation, and similar techniques provide a powerful set of tools for prototyping. Although often thought of as tools for writing reusable software, they give a guiding structure to prototyping. "Thin-line" prototyping is a technique that draws on these object-oriented mechanisms. A thin-line prototype is one that implements all major components of a system, although initially with limited complexity. For example, assume an implementation of an expert-system in a complex network environment. A thin-line prototype would include all parts of the system to test communication, interaction, etc., but with limited functionality. The expert system may only have enough rules to solve a few example problems; the network communications may only implement enough messages to test the efficiency of communications; the user interface may only consist of enough screens to solve an initial problem set, and so on.

The power of thin-line prototypes is that they test the overall architecture of the program without requiring a complete implementation. Once this is done and evaluated for efficiency and robustness by engineers and for usability and correctness by end users, we can continue development with a focused, easily managed cycle of adding functionality, testing it, and planning. In our experience, most AI programs are built this way.

Reuse

It would be nearly impossible to write a book on programming without a discussion of an idea that has become something of a holy grail to modern software development: code reuse. Both in industry and academia, programmers are under pressure, not only to build useful, reliable software, but also to produce useful, reusable components as a by-product of that effort. In aiming for this goal, we should be aware of two subtleties.

The first is that reusable software components rarely appear as by-products of a problem-specific programming effort. The reason is that reuse, by definition, requires that components be designed, implemented, and tested for the general case. Unless the programmer steps back from the problem at hand to define general use cases for a component, and designs, builds, tests, and documents to the general cases, it is unlikely the component will be useful to other projects. We have built a number of reusable components, and all of them have their roots in this effort to define and build to the general case.

The second thing we should consider is that actual components should not be the only focus of software reuse. Considerable value can be found in reusing ideas: the idioms and patterns that we have demonstrated in this book. These are almost the definition of skill and mastery in a programmer, and can rightly be seen as the core of design and reuse.

32.4 Final Thoughts

It has been our goal to give the reader an understanding of, not only the power and beauty of the programming languages Prolog, Lisp, and Java, but also of the intellectual depth involved in mastering them. This mastery involves the languages syntax and semantics, the understanding of its idioms of use, and the ability to project those idioms into the patterns of design and implementation that define a well-written program.

In approaching this goal, we have focused on common problems in Artificial Intelligence programming, and reasoned our way through their solution, letting the idioms of language use and the patterns of program organization emerge from that process. The power of idioms, patterns, and other forms of engineering mastery is in their application, and they can have as many realizations, as many implementations as there are problems that they may fit. We hope our method and its execution in this book have helped the student understand the deeper reasons, the more nuanced habits of thinking and perception, behind these patterns. This is, to paraphrase Einstein, less a matter of knowledge than of imagination.

We hope this book has added some fuel to the fires of our readers' imaginations.

Bibliography

Abelson, H. and Sussman, G. J., 1985. *Structure and Interpretation of Computer Programs*. Cambridge, MA: MIT Press.

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I. and Angel, S., 1977. *A Pattern Language*. New York: Oxford University Press.

Bellman, R. E., 1956. *Dynamic Programming*. Princeton, NJ: Princeton University Press.

Brachman, R. J. and Levesque, H. J., 1985. *Readings in Knowledge Representation*, Los Altos CA: Morgan Kaufmann.

Bundy, A., Byrd, L., Luger, G., Melish, C., Milne, R., and Stone, M. 1979. Solving Mechanics Problems Using Meta-Inference. In *Proceedings of IJCAI-1979*, 1017-1027.

Chakrabarti, C., Rammohan, R., and Luger, G. F., 2005. A First-Order Stochastic Prognostic System for the Diagnosis of Helicopter Rotor Systems for the US Navy. In *Proceedings of the 2nd Indian International Conference on Artificial Intelligence*. Pune, India. Elsevier Publications.

Charniak, E., Riesbeck, C. K., McDermott, D.V., and Meehan, J.R., 1987. *Artificial Intelligence Programming*, 2nd ed. Hillsdale, NJ: Erlbaum.

Church, A. (1941). The Calculi of Lambda-Conversion. *Annals of Mathematical Studies* ***6***. Princeton NJ: Princeton University Press.

Clocksin, W. F. and Mellish, C. S., 1984. *Programming in Prolog*. New York, Springer-Verlag.

Clocksin, W. F. and Mellish, C. S., 2003. *Programming in Prolog: Using the ISO Standard*. New York, Springer.

Collins, A. and Quillian, M. R., 1969. Retrieval Time for Semantic Memory. *Journal of Verbal Learning & Verbal Behavior*, 8: 240-247.

Colmerauer, A. H., 1975. *Les Grammaires de Metamorphose*, Groupe Intelligence Artificielle, Universite Aix-Marseille II, France.

Colmerauer, A., H. Kanoui, H., 1973. *Un Systeme de Communication Homme-machine en Francais*. Groupe Intelligence Artificiale, Université Aix-Marseille II, France.

Coplein, J. O. and Schmidt, D. C., 1995. *Pattern Languages of Program Design*. Reading, MA: Addison-Wesley.

Dahl, V., 1977. *Un Système Deductif d'Interrogation de Banques de Donnes en Espagnol*, PhD thesis, Université Aix-Marseille, France.

Dahl, V. and McCord, M.C. 1983. Treating Coordination in Logic Grammars. *American Journal of Computational Linguistics,* 9:69–91.

Darwin, C., 2007. *The Voyage of the Beagle.* Retrieved 3/23/07 from http://www.literature.org/authors/darwin-charles/the-voyage-of-the-beagle.

DeJong, G. and Mooney, R., 1986. Explanation-Based Learning: An Alternative View. *Machine Learning*, 1(2); 145-176.

Dybvig, R. K., 1996. *The Scheme Programming Language*. Upper Saddle River, NJ: Prentice Hall.

Earley, J., 1970. An efficient context-free parsing algorithm. *Communications of the ACM*, 6(8): 451-455.

Eiben, A. E., Smith, J. E., 1998. *Introduction to Evolutionary Computing*. Berlin: Springer.

Evans, E., 1983. *Domain Driven Design: Tackling Complexity in the Heart of Software*. Upper Saddle River NJ: Addison-Wesley.

Feigenbaum, E. A., and Feldman, J., eds., 1963. *Computers and Thought*. New York: McGraw-Hill.

Fikes, R. E., Hart, P. E., and Nilsson, N. J., 1972. Learning and Executing General Robot Plans. *Artificial Intelligence*, 3(4): 251-288.

Fikes, R. E. and Nilsson, N. J., 1971. STRIPS: A New Approach to the Application of Theorem Proving to Artificial Intelligence. *Artificial Intelligence*, 1(2): 189-208.

Forbus, K. D. and deKleer, J. 1993. *Building Problem Solvers*. Cambridge, MA: MIT Press.

Gamma, E., Helm, R., Johnson, R, and Vlissides, J., 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Reading, MA: Addison-Wesley.

Ganzerli, S., De Palma, P., Smith, J. D., and Burkhart, M. F., 2003. Efficiency of Genetic Algorithms for Optimal Structural Design Considering Convex Models of Uncertainty. *Proceedings of the Ninth International Conference on Statistics and Probability in Civil Engineering*, Berkeley: 1003-1010. Rotterdam, NL: Millpress Science Publishers.

Gazdar, G. and Mellish, C., 1989. *Natural Language Processing in PROLOG: An Introduction to Computational Linguistics*. Reading, MA: Addison-Wesley.

GECCO, 2007. *Genetic and Evolutionary Computing Conference, 2007*. Retrieved 3/23/07 from http://www.sigevo.org/gecco-2007.

Goldberg, D. E., 1989. *Genetic Algorithms in Search Optimization and Machine Learning*. New York: Addison-Wesley.

Graham, P. 1993. *On LISP: Advanced Techniques for Common LISP*. Englewood Cliffs, NJ: Prentice Hall.

Graham, P. 1995. *ANSI Common Lisp*. Englewood Cliffs, NJ: Prentice Hall.

Halcolm, J. R. and Shultz, R., 2005. *Tau: A web-deployed hybrid prover for first-order logic with identity with optional inductive proof*. 12 April 2008, http://www.hsinfosystems.com/Tau_JAR.pdf

Hasemer, T. and Domingue, J., 1989. *Common LISP Programming for Artificial Intelligence*. Reading, MA: Addison-Wesley.

Haupt, L. and Haupt, S., 1998. *Practical Genetic Algorithms*. New York: John Wiley and Sons.

Hayes, P., 1977. In Defense of Logic. *Proceedings of IJCAI-77*, Cambridge, MA: MIT Press.

Hermenegildo, M. And the Ciao Development Team, 2007. *An Overview of The Ciao Multiparadigm Language and Program Development Environment and its Design Philosophy*. ECOOP Workshop on Multiparadigm Programming with Object-Oriented Languages MPOOL 2007, July 2007.

Hill, P. and Lloyd, J., 1995. *The Gödel Programming Language*. Cambridge, MA: MIT Press.

Holland, J. H., 1975. *Adaptation in Natural and Artificial Systems*. Ann Arbor MI: University of Michigan Press.

Jurafsky, D., and Martin, J. H., 2008. *Speech and Language Processing (2nd ed)*, Upper Saddle River, NJ: Prentice Hall.

Kedar-Cabelli, S. T. and McCarty, L. T., 1987. Explanation-Based Generalization as Resolution Theorem Proving. *Proceedings of the Fourth International Workshop on Machine Learning*.

King, S. H., 1991. *Knowledge Systems Through Prolog*. Oxford: Oxford University Press.

Kowalski, R., 1979. Algorithm = Logic + Control. *Communications of the ACM* **22**: 424-436.

Kowalski, R., 1979. *Logic for Problem Solving*. Amsterdam: North Holland.

Krzysztof, R. A. and Wallace, M., 2007. *Constraint Logic Programming Using Eclipse*. Cambridge UK: Cambridge University Press.

Lloyd, J. W., 1984. *Foundations of Logic Programming*. New York: Springer Verlag.

Lucas, R., 1996. *Mastering Prolog*. London UK: UCL Press.

Luger, G. F., 2009. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*, Boston MA: Addison-Wesley Pearson.

Maclean, N., 1989. *A River Runs Through It*, Chicago: University of Chicago Press.

Maier, D. and Warren, D. S., 1988. *Computing with Logic: Logic Programming with Prolog*. Boston MA: Addison-Wesley.

Malpas, J., 1987. *Prolog: A Relational Language and its Applications*. Englewood Cliffs NJ: Prentice Hall.

McCarthy, J., 1960. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM* **3**(4).

McCord, M. C., 1982. Using slots and modifiers in logic grammars for natural language. *Artificial Intelligence*, 18:327–367.

McCord, M. C., 1986. Design of a Prolog based machine translation system. *Proceedings of the Third International Logic Programming Conference*, London.

McCune, W. W. and Wos, L., 1997. Otter: The CADE-13 competition incarnations. *Journal of Automated Reasoning*, 18(2): 211-220.

Milner, R., Tofte, M. , Harper , R., and MacQueen, D., 1997. *The Definition of Standard ML (Revised)*. Cambridge *MA: MIT Press*.

Minsky, M., 1975. *A Framework for Representing Knowledge*. In Brachman and Levesque (1985).

Minton, S., 1988. *Learning Search Control Knowledge*. Dordrecht: Kluwer Academic Publishers.

Mitchell, T. M., 1978. Version Spaces: An Approach to Concept Learning. *Report No.STAN-CS-78-711*, Computer Science Dept., Stanford University.

Mitchell, T. M., 1979. An Analysis of Generalization as a Search Problem. *Proceedings of IJCAI*, 6.

Mitchell, T. M., 1982. Generalization as Search, *Artificial Intelligence*, 18(2): 203-226.

Mitchell, T. M., Keller, R. M., and Kedar-Cabelli, S. T., 1986. Explanation-Based Generalization: A Unifying View. *Machine Learning*, 1(1): 47-80.

Mycroft, A. and O'Keefe, R. A., 1984. A Polymorphic Type System for Prolog. *Artificial Intelligence*, 23: 295-307.

Neves J. C. F. M., Luger, G. F., and Carvalho, J. M., 1986. A Formalism for Views in a Logic Data Base. In *Proceedings of the ACM Computer Science Conference*, Cincinnati OH.

Newell, A., 1982. The Knowledge Level. *Artificial Intelligence*, 18(1): 87-127.

Newell, A. and Simon, H. A. 1976. Computer Science as Empirical Inquiry: Symbols and Search. *Communications of the ACM*, 19(3):113–126.

Nilsson, N. J., 1980. *Principles of Artificial Intelligence*. Palo Alto, CA: Tioga.

O'Keefe, R., 1990. *The Craft of PROLOG*. Cambridge, MA: MIT Press.

O'Sullivan, B., 2003. Recent advances in constraints, Joint ERCIM/CologNet International Workshop on Constraint Solving and Constraint Logic Programming. *Lecture Notes in Computer Science 2627*, Berlin: Springer.

Overbay, S., Ganzerli, S., De Palma, P, Brown, A., Stackle, P., 2006. Trusses, NP-Completeness, and Genetic Algorithms. *Proceedings of the 17th Analysis and Computation Specialty Conference*. St. Louis, MO.

Paulson, L. C., 1989. Isabelle: The Next 700 Theorem Provers. *Journal of Automated Reasoning* 5: 383-397.

Pereira, L. M. and Warren, D. H. D., 1980. Definite Clause Grammars for Language Analysis – A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence,* 13:231–278.

Pless, D. and Luger, G. F., 2003. EM learning of product distributions in a first-order stochastic logic language. *Artificial Intelligence and Soft Computing: Proceedings of the IASTED International Conference*. Anaheim: IASTED/ACTA Press. Also available as *University of New Mexico Computer Science Technical Report TR-CS-2003-01*.

Quinlan, J. R., 1986. Induction of Decision Trees. *Machine Learning*, 1(1):81–106.

Quinlan, J, R., 1996. Bagging, Boosting and C4.5. *Proceedings AAAI 96*. Menlo Park CA: AAAI Press.

Rajeev, S. and Krishnamoorthy, C. S., 1997. Genetic Algorithms-Based Methodologies for Design Optimization of Trusses. *Journal of Structural Engineering*, 123 (3): 350-358.

Robinson, J. A., 1965. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM,* **12**: 23-41.

Robinson, J. A. and Voronkov, A., 2001. *Handbook of Automated Reasoning: Volume 1*. Cambridge MA: MIT Press.

Ross, P., 1989. *Advanced Prolog*. Reading, MA: Addison-Wesley.

Roussel, P., 1975. *Prolog: Manuel de Reference et d'Utilisation*. Luminy, France, Groupe d'Intelligence Artificialle, Université d' Aix-Marseille.

Sakhanenko, N., Luger, G. F. and Stern, C. R., 2006. Managing Dynamic Contexts using Failure-Driven Stochastic Models. *Proceedings of FLAIRS Conference*. Menlo Park CA: AAAI Press.

Seibel, P., 2005. *Practical Common Lisp*. Berkeley CA: Apress, Inc.

Shannon, C., 1948. A Mathematical Theory of Communication. Murray Hill NJ: *Bell System Technical Journal*.

Shapiro, S. C., ed., 1987. *Encyclopedia of Artificial Intelligence*. New York: Wiley-Interscience.

Smith, J. B., 2006. *Practical OCaml*. Berkeley CA: Apress, Inc.

Somogyi, Z., Henderson, F., and Conway, T., 1995. Logic Programming for the real world. In *Proceedings of the Eighteenth Australasian Computer Science Conference*, R Kotagiri (Editor), 1995, Australian Computer Science Communications: Glenelg, South Australia. pp. 499-512.

Sowa, J. F., 1984. *Conceptual Structures: Information Processing in Mind and Machine*. Reading MA: Addison-Wesley.

Steele, G. L., 1990, *Common LISP: The Language*, 2nd ed. Bedford, MA: Digital Press.

Sterling, L. and Shapiro, E., 1986. The Art of Prolog. Advanced Programming Techniques. Cambridge MA: MIT Press.

Sussman, G. and Steele, G., 1975. *SCHEME: An Interpreter for Extended Lambda Calculus,* AI Memo 349, MIT Artificial Intelligence Laboratory, Cambridge, Mass.

Tanimoto, S. L., 1990. *The Elements of Artificial Intelligence using Common LISP*. New York: W.H. Freeman.

Touretzky, D. S., 1990. *Common LISP: A Gentle Introduction to Symbolic Computation*. Redwood City, CA: Benjamin/Cummings.

Turing, A., 1948. Intelligent Machinery. A report to the National Physical Laboratory. London.

Van Le, T., 1993. *Techniques of Prolog Programming with Implementation of Logical Negation and Quantified Goals*. New York: Wiley.

Walker, A., McCord, M., Sowa, J. F., and Wilson, W. G., 1987. *Knowledge Systems and Prolog: A Logical Approach to Expert Systems and Natural Language Processing*. Reading, MA: Addison-Wesley.

Warren, D. H. D., Pereira, L. M. and Pereira, F., 1977. Prolog - the language and its implementation compared with LISP. *Proceedings, Symposium on AI and Programming Languages, SIG-PLAN Notices*, **12**(8).

Warren, D. H. D., Pereira, F. and Pereira, L. M., 1979. User's Guide to DEC-System 10 PROLOG. *Occasional Paper 15, Department of Artificial Intelligence*, University of Edinburgh, UK.

Wilensky, R., 1986. *Common LISPCraft*, New York: Norton Press.

Winston, P. H., Binford, T. O., Katz, B, and Lowry, M., 1983. Learning Physical Descriptions from Functional Definitions, Examples, and Precedents. *Proceedings of National Conference on Artificial Intelligence, Washington D.C.*, San Francisco: Morgan Kaufman. 433-439.

Winston, P. H. and Horn, B. K. P., 1984. *LISP*. Reading, MA: Addison-Wesley.

Index