SAMS

22447

The Waite Group

# Artificial Intelligence Programming on the Macintosh

Dan Shafer

[redacted]

# Artificial Intelligence Programming for the Macintosh®

Dan Shafer

**Howard W. Sams & Co.**
A Division of Macmillan, Inc.
4300 West 62nd Street, Indianapolis, IN 46268 USA

FIRST EDITION
FIRST PRINTING—1986

International Standard Book Number: 0-672-22447-X
Library of Congress Catalog Card Number: 86-60027

Edited by: *Don MacLaren*
Designed by: *T. R. Emrick*
Cover Art by: *Ron Troxell*
Computer Art by: *Don Clemons*
Chapter Opening Cartoons by: *Bob Johnson*

*Printed in the United States of America*

# Contents

# *Preface*

Artificial Intelligence (AI) has fascinated me from the first time I heard about it. My first exposure to it came on a tour of the Stanford Artificial Intelligence Laboratory (SAIL). I was entranced at the work the people at SAIL were doing with computer music generation, robotics, and problem-solving. This book is the beginning of the fulfillment of my intense desire to be part of what is happening in that community.

I also wanted this book to fill what seemed to me to be a rather large gap in the literature of AI. I have searched fruitlessly for a book that did not assume the reader was already knowledgeable about programming in an esoteric language such as LISP and aware of such fundamental computer science ideas as heuristics and search mechanisms. I hope this book takes a small step toward filling the need of interesting more and more people in the field of AI.

Few would argue with the proposition that AI is the next frontier in the world of computers. In the past two years, real AI development tools and languages have become available on microcomputers, making hands-on experimentation with and exploration of this fascinating field accessible to millions who otherwise would have had to content themselves with reading about the subject.

The growing importance of AI is enough of a *practical* reason for you to spend your time reading this book, examining the programs it contains, and exploring AI with the book as a guide. But the value of learning more about intelligence—artificial or natural—transcends even the practical benefit of learning about a field that promises to have a tremendous impact on our daily lives.

## Who This Book Is For

On one level, this is an introductory book about AI programming techniques. To read the first part of each chapter in Part I and all of Part II requires no programming background in particular, though a nodding acquaintance with some of the basic ideas of computers and programming would be useful. On another level, the book abounds with examples and complete AI programs written in ExperLogo® from ExperTelligence of Santa Barbara, California. To make full use of these programs, you should have some Logo programming knowledge (though Chapter 10 may give you enough of an introduction to the language), access to an Apple Macintosh® computer, and a copy of ExperLogo®.

If you use ExperLogo® on your Macintosh®, you'll also find that an external drive is extremely helpful and a printer almost a requirement.

## Two Decisions

Two aspects of this book deserve an explanation. Why did I focus on the Macintosh® and why did I choose Logo as the principal language?

The Mac, with its desktop, icons, and windows, comes closer to what most AI researchers view as a "rich and supportive" programming environment than any other machine on the market. As you program the examples in this book, you will be doing so in an environment that is quite similar to that enjoyed by professional AI workers. In addition, the Mac is fun to use and seems more accessible to those new to computers than any other computer.

Logo is an excellent compromise selection between LISP, which is clearly the AI language of choice but is difficult to learn and not readily available on the Mac, and BASIC, which is far more available but not well suited to AI programming. I deal with the issue of language selection at some length in Part II of this book.

Besides, Logo is easy to learn and is gaining increasing acceptance in the educational community as a superb language for the exploration of new ideas. And that is precisely what I wish for you: exploration of new ideas.

DAN SHAFER

# Acknowledgements

As I was developing the ideas for this book, I was fortunate enough to find myself involved with a small group of programmers and AI fans who helped me to focus the content, the direction, and the approach of the book. I am grateful to these people—Chuck Blanchard, John Worthington, Jason Christofilos, Mike VanHorn, and my editor, Robert Lafore—for helping to brainstorm the book and for believing in it.

Along the way, I have had the chance to discuss the book and its contents with some truly helpful, even inspirational, AI workers. A special vote of thanks goes to Steve Lurya of Santa Barbara, a former ExperTelligence programmer who made a particular contribution to Chapter 7 and also helped me grapple with some of the ideas of AI and AI programming and thus helped improve the book as a whole.

ExperTelligence personnel—notably, President Denison Bollay, Marketing VP Tony Uhler, and programmer-engineer Robert Reali—have also been very helpful in aiding my understanding of their fine products, ExperLogo and ExperLisp.

As usual, my editorial associate and research aide Don Huntington made a major contribution to this work. I am grateful to him for tightened sentences, crystallized thinking, and organizational help.

My editor, Robert Lafore, has been a source of continuing support and inspiration as this book has grown from a collection of Mac programs about AI into a cohesive introductory book on AI techniques and languages. In brainstorming sessions and in editing discussions, he has questioned, probed, prodded, and cajoled me into making this book as good as I could possibly make it.

Ken Schieser is a remarkable young man with an excellent grasp of Logo who wrote two of the more intriguing programs in this book: "Micro Blocks World" and the "Intelligent Maze." He is the first person I've met who is as good as he is at programming yet doesn't want to make it a career.

Finally, my family has continued, as they have with all of my previous writing efforts, to be supportive and patient with me. I appreciate their love and closeness—even when that closeness is to the back of my head or through a closed office door—more than words can express.

*This book is dedicated to Albert and Mary Lee Hunton. Your natural common sense and wisdom about how the world and people work far exceed any intelligence—real* or *artifical—I know about. God bless you both.*

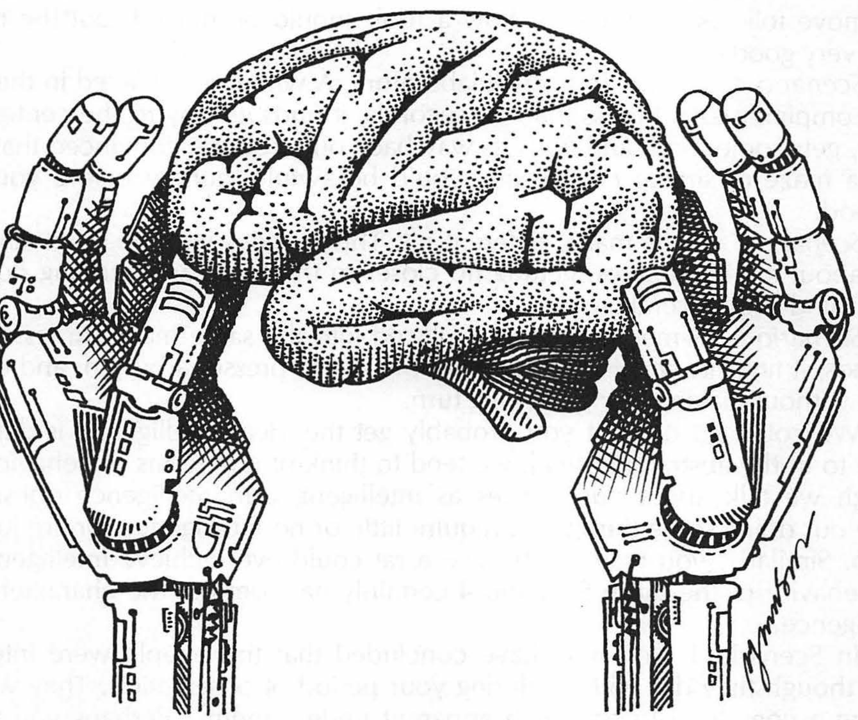
I

# *Artificial Intelligence Programming Techniques*

CE ARTIFICIAL INTELLIGENC
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGENC
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTEL

# 1

# Overview: What Makes a Program Intelligent?



- ○ What Is AI?
- ○ Getting the Computer to Understand Humans
- ○ Basic Research in Search Techniques
- ○ Current Work in Expert Systems
- ○ Importance of Computer Learning

At its root, the difficulty with Artificial Intelligence (AI) as a discipline is the lack of a good definition of the key word, "intelligence." We *think* we know what we mean when we say a particular person, animal, or thing exhibits intelligence. But think of each of the following scenarios and ask yourself whether intelligence is involved in any.

*Scenario 1.* A man and a woman sit across a table from each other. On the table chess pieces are arrayed on a chessboard. A glance reveals no particular pattern to the arrangement of the pieces. You watch the scene for 15 minutes and neither player moves a piece.

*Scenario 2.* Same setting as the first, except that you've now watched for 15 minutes and 1 second and the woman picks up one of her pieces and moves it a few squares. The move follows the rules for how that particular piece should be moved, but the move is not a very good one.

*Scenario 3.* One person is in the room with a chessboard and pieces in front of him. Off to the side, a small computer displays the chessboard and pieces on its screen. The screen reads out a move, "Rook to Queen's Rook 5." The move follows the rules for how a rook should be moved, but the move is not a very good one.

*Scenario 4.* You are in a small laboratory. A white rat is placed in the center of a complex maze. In less than 30 seconds, it finds its way to the center of the maze, gets some food, and finds its way back out. You are convinced that, faced with a maze of similar complexity, you'd be awfully hungry before you found the food.

*Scenario 5.* Same maze, different rat. This rat goes into the maze and blunders about for 10 minutes, getting no closer to the food, and running down the same blind alley several times.

*Scenario 6.* A mechanical rat is placed into the same maze. In less than 15 seconds, it finds its way to the center of the maze, presses a button, and exits the maze without making a single wrong turn.

We could go on. But you probably get the idea. Intelligence is difficult to relate to in the abstract; instead, we tend to think of it in terms of behavior. And, though we talk about our species as intelligent, our intelligence doesn't preclude our doing some things that require little or no intelligence, or are just plain dumb. Similarly, you may not believe a rat could ever achieve intelligence, but the behavior of the rat in Scenario 4 certainly has some of the characteristics of intelligence.

In Scenario 1, you may have concluded that the people were intelligent, even though they did nothing during your period of observation. They were observing a complex situation with apparent understanding. Perhaps you thought that chess is a difficult game and figured that anyone who could understand the game and concentrate that hard upon a single move had to be intelligent. But you didn't really *know* that, did you? It's *possible* that the players had gone to sleep—or were mannequins!

The point is, we can define intelligence by looking at specific behaviors, tangible or presumed, and drawing conclusions. In other words, we can discuss intelligence in the *specifics*, but in the abstract it is a difficult subject, one which continues to elude meaningful definition.

Two interesting points derive from this brief discussion.

First, intelligence quite often appears to be largely a matter of *problem-solving*. Chess players are trying to solve a problem—finding the best move from a huge array of possible and legal ones. The rats in the maze are trying to solve a problem—finding the right path to the food. Alternatively, we could think of these expressions of intelligence in terms of goal attainment. The chess players have a goal of victory and the rats a goal of satisfying hunger. Both ways of looking at intelligence are valid, up to a point, and they encompass a great deal of what we think of as intelligent. Neither presents a complete or comprehensive view of the subject, however.

The second point is that intelligence is a *concept* here, not a measurement. Even a dumb chess move requires intelligence to make. Thus we are not concerned with whether the behavior involved is "wise" or "fruitful," only that it exhibits "intelligence."

## What Has This to Do with Programs?

This book is only indirectly about intelligence in the abstract. My purpose is to enable you to use your Macintosh® to explore the fundamental ideas involved in artificially intelligent computer programs. So why spend time discussing an abstract, or noncomputer, definition of intelligence? I do so because defining a program as *intelligent* is as difficult as trying to define intelligence in humans or laboratory animals. Obviously I consider the programs in this book to be in some sense intelligent, or their inclusion here would be foolish, given our objective.

So I've adopted a somewhat circular definition of intelligence which has been used by many researchers in the field of AI, even though it begs the underlying philosophical question of what constitutes intelligence.

> For our purposes in this book, we will consider a program intelligent if it does something that we would feel exhibited intelligence if a human did the same thing.

This definition spares us the niceties—and the tedium—of an extended lesson on the subject of intelligence and moves us into our main interest: writing programs to explore the ideas behind AI. The fact is, we don't have to write programs that are indisputably intelligent, even by our broad definition, to explore some of the *concepts* underlying AI.

## Major AI Research Themes

Research in AI has taken on greater emphasis and importance during the past few years and the field has undergone an inevitable segmentation. Today, AI work is concentrated in six major categories of discovery: search techniques, natural language processing (NLP), expert systems, pattern recognition, learning, and planning/problem-solving.

We will now turn to a brief discussion of the subject matter and relative importance of each of these areas to gain a good foundation of understanding of AI, though our programs will not deal with all of these areas.

**Search Techniques: Brute Force vs. Finesse**

There are many problems to which computers can be applied that involve the need to search through a number of possible solutions and come up with the best one, the only one, or any one that works. In fact, for a time many of those involved in AI research felt that the idea of *search* was the core on which all else would be built. The concept no longer enjoys such a lofty position, but its importance is not in doubt.

There are dozens of design methods that permit AI programs to carry out searches. We will examine a small subset of those that have been proposed or tried.

The most obvious and effective—but least efficient—search technique is called "brute force." Using this technique to solve the chess problem of which move to make, a computer program would look at every single legal move the computer could make at the time, every single response its human opponent could make, and perhaps every single countermove the computer could make. A value could then be assigned to each such "solution." Then the program could simply sort the results of this massive search into a list and pick the solution with the highest value.

This approach will obviously work. It is, in fact, applied by many computerized chess-playing programs. However, there are at least two fundamental problems with this method. First, it can be extremely time and resource consuming. The design requires a great deal of memory, or disk capacity, for holding the values of all the moves. There will be tens of millions of such moves in any given situation. Second, the brute force technique, as Margaret Boden (1977, 346) has pointed out, "relies on speed of computation, slavish adherence to some exhaustive procedure, and faultless memory, as opposed to intelligent ways of minimizing thinking." In other words, the approach really isn't very intelligent or enlightening.

Using a brute force method leads to something called the "combinatorial explosion." The number of possible points of solution increases geometrically as the number of "moves" or decisions increases. I'll have more to say about this when we explore specific solutions offered by our search-exploration programs. For now it is enough to know that brute force approaches seldom work well for any but the most trivial of problems, particularly in the world of microcomputers.

As a result of combinatorial explosion, AI researchers spend a great deal of time attempting to limit the search space of a problem-solving program.

Broadly speaking, search strategies fall into the two major categories: *depth-first* and *breadth-first*. A "depth-first" search approach follows each possible solution to its ultimate limit before turning to other possible solutions. A "breadth-first" approach examines all possible alternatives in a somewhat cursory fashion and decides on some basis which to follow first. At each level of complexity, a breadth-first approach looks at each possible next step before deciding how to proceed.

Neither search strategy is inherently more efficient than the other. Which strategy will reach the desired goal sooner is a function of how far down the search "tree" the solution appears rather than a function of the strategy chosen.

Figure 1-1 provides an example of how the two search strategies differ from one another. In that figure, we begin at the top of the search tree and place our goal—for example, the food at the center of the laboratory maze—at the point labeled 7. A depth-first search approach moving down the search tree to the end of any given line of investigation and, for the sake of argument, always choosing the right-most path, would find the cheese before a breadth-first search.

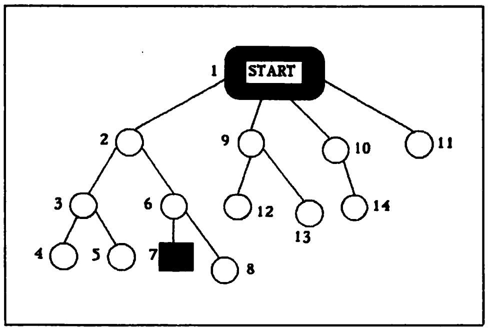


**Figure 1-1. Depth-first search "defeats" breadth-first**

A depth-first search would follow the steps in this order: 1, 2, 3, 4, 3, 5, 3, 2, 6, 7. A total of 10 steps would be required. A breadth-first search would follow the steps in this order: 1, 2, 9, 10, 11, 3, 6, 12, 13, 14, 4, 5, 7—a total of 13 steps. But in Figure 1-2 we put the cheese goal at position 13. The depth-first search now goes through the following steps to find the goal: 1, 2, 3, 4, 3, 5, 3, 2, 6, 7, 6, 8, 6, 2, 1, 9, 12, 9, 13—for a total of 19 steps. The breadth-first search, meanwhile, uses just 9 steps to reach the goal: 1, 2, 9, 10, 11, 3, 6, 12, 13.

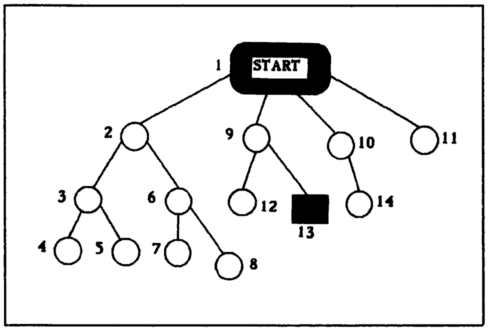


**Figure 1-2. Breadth-first search turns the tables**

Both search strategies will work with a goal-oriented problem—one in which we know where we are trying to go. But both lean more in the direction of brute force than in the direction of finesse. More intelligent search techniques apply principles from the field of computer science known as *heuristics.*

Broadly understood, "heuristics" applies to a computer program being able to adjust its actions as it proceeds to carry out tasks. In other words, the program "learns"—at least in a rudimentary sense—which paths are likely to be fruitful and which are not. The concept of *heuristic search* is one in which information gained is used to explore the problem at various points. The points may be summarized as follows:

1. Determining the "node," or decision point, to expand on next, rather than proceeding in either a depth-first or breadth-first order without regard to the situation.

2. Deciding which subsequent node to move to at a given decision point, rather than blindly following a right-left or left-right strategy.
3. Discerning certain nodes as unprofitable and discarding them from the search tree completely—a process known, not surprisingly, as "pruning."

Such searches are sometimes referred to as "ordered" searches. One example is the concept of a "best-first search" in which heuristics are used only at point 1 to determine which node to analyze next, choosing the most promising on some basis.

One other search concept is worth mentioning. In game searches—in which an opponent tries to thwart the computer program's goal—the computer often uses an approach called "minmax" searching. The term comes from the fact that the computer, using such search techniques, tries to find a proper course of action, assuming that its opponent will attempt to *min*imize the computer's gain and that the computer will always find it advantageous to *max*imize its position.

We will examine searching techniques as we study and explore two programs: "Micro-Logician" (Chapter 3) and "Intelligent Maze" (Chapter 6). This last, a computer vs. human game, uses minmax search techniques.

**Natural Language Processing: Computer Understanding**

Many AI researchers argue persuasively that the fundamental characteristic of a truly intelligent program is that it can understand information fed into it. The problem is that this definition substitutes the difficult word "understand" in place of our initial difficult word, "intelligence."

Dr. Roger C. Schank and his colleagues at the Yale University Artificial Intelligence Laboratory have become well known in AI circles for their strong emphasis on computer understanding. (One of Dr. Schank's books is the source for some of the programs and ideas contained in this book.) Getting computers to understand human language is clearly recognized in the AI community as a goal that is foundational to other AI research. The ultimate role of computer understanding will be to make the computer a supremely useful tool in many fields of human endeavor.

The goal of natural language processing (NLP) is to design computer programs that can process human speech patterns. ("Speech" refers to verbal communication—communication carried out by manipulation of meaningful symbols. *Speech* is used here in its broadest sense; in fact, the word almost always refers to nonoral communication. Voice and speech recognition are not formally part of AI, though the two disciplines are certain to be intertwined with NLP at some point in the future.)

This goal is elusive, though we may be closer to reaching it than we are to reaching many other intermediate goals of AI. The fact is, however, that programs that process human language at all, do so in a very limited fashion. Given the amount of miscommunication that goes on among human beings, it should not surprise us to find that communicating in "natural" language poses many problems and challenges. Take, for example, the simple sentence: "Mary walked up and kissed Kevin."

We understand that this sentence means that a girl named Mary saw a boy named Kevin, walked to where he was, and pressed her lips against his lips (or perhaps his forehead or his cheek?). But a computer, to understand this sentence, would have to know or be told:

- that the phrase "walked up" does not imply an ability to climb walls or thin air;
- that a kiss is something good and connotes a liking for the person being kissed;
- that Mary is a girl's name and Kevin a boy's name and that kissing between members of the opposite sex has a particular connotation.

In the absence of this information, the computer could not answer questions about that sentence that humans wouldn't even have to think about. These questions might include:

- Did Mary hate Kevin?
- Was Mary standing close to Kevin when she kissed him?
- Was Mary driving a car at the time?

Designing a computer program that can understand human speech requires a great deal of design effort and energy. It also requires, so far at least, a willingness to limit things about which the computer will "know"—referred to as the system's "knowledge domain." It is far easier to teach a computer about the words and ideas involved in moving blocks around on a tabletop (as we do in Chapter 5, "Micro Blocks World") than to teach it about human emotions.

One technique for discovering some of the ideas and problems behind NLP is to create programs which generate words, sentences, and poems. We have such a program in this book: "The Digital Poet" (Chapter 4). The "Micro Blocks World" program of Chapter 5 is a microcomputer version of a famous AI program called SHRDLU. Micro Blocks Worlds operates within a limited domain—a set of blocks of various sizes, shapes, and colors and the relationships among their surfaces. It understands English sentences that tell it what to do with those blocks. The program contains elements of search and decision-making, but its real attraction lies in an ability to process and respond to natural-language input.

**Expert Systems: Getting Advice from a Computer**

This major area of AI research combines a number of different areas. It is also, at the moment, the one aspect of AI that is producing commercial products and, as a result, the one about which many people have at least some knowledge. Expert systems are being touted as the next major thrust in microcomputer technology.

An expert system can be thought of most simply as a computer program that offers advice based on its knowledge or experience in the field it is being questioned about.

There are many expert systems, with many different designs, on the market and in the literature of AI. Commercial products which are now on the market, or soon will be, assist human decision-makers in such areas as the following:

- commercial loan analysis and decision-making;
- wine selection;

- disease diagnosis;
- prospecting for mineral deposits;
- tax and investment decisions.

All expert systems have certain components in common. They have a *knowledge base* of information from which they draw conclusions and make recommendations. Many times, the knowledge base contains rules about, or examples of, decisions. Expert systems also have *inference engines*—programs at the heart of the systems—which enable the computer to analyze its knowledge base and determine what decision-making rules are being followed. In that way it can respond to the question being asked of it. The term "inference engine" refers to the fact that it draws conclusions from information (an inference is a conclusion drawn from facts). The term "engine" has long been used in computing circles to symbolize a somewhat stand-alone device or section of programming code which "drives" the program or system of which it is a part.

We will look at two programs in the field of expert systems. Chapter 7 presents an implementation, in ExperLogo®, of the popular expert systems programming language, Prolog. The program is known as "Prologo" and it is capable of drawing some relatively sophisticated conclusions about knowledge bases we provide for it. It is completely functional and can be used as a model to create an expert system on any subject we wish. Chapter 8 integrates the inference engine (which is what Prologo and its "big brother," Prolog, really are) into two full-blown, if small-scale, knowledge bases. As Prologo manipulates these knowledge bases and applies facts and rules intelligently, it displays many of the earmarks of an expert system.

Our expert system will *not* do two things that some AI researchers would say are requisite to a true expert system. It will not be able to explain how it reached its conclusions, and it will not be able to learn from experience. In other words, if it reaches a conclusion and we indicate that its advice is incorrect or incomplete, it will not avoid making the same mistake the next time. As a matter of fact, most commercial expert systems don't do either of these things yet, either.

## Pattern Recognition

Early in its development, AI was blessed with a number of researchers who realized that human intelligence was often based on patterns and relationships. We could look at a situation that we had not encountered previously and still understand it in great detail by comparing it with other similar experiences.

The ideas behind true pattern-matching of this kind are extremely complex. The fundamental idea in pattern-matching is that events of a certain type—sentences, for example—tend to fall into groups of patterns. A question often has a verb followed by a noun followed by a noun phrase, as shown in Figure 1-3. A declaratory sentence, which simply imparts information, usually has a noun phrase first, then a verb, and sometimes a noun phrase after the verb. Figure 1-4 shows this structure.

If a computer were given those patterns and information about vocabulary (what words are nouns, noun phrases, and verbs, for our present example), it could use pattern-matching to determine the type of sentence "handed" it by

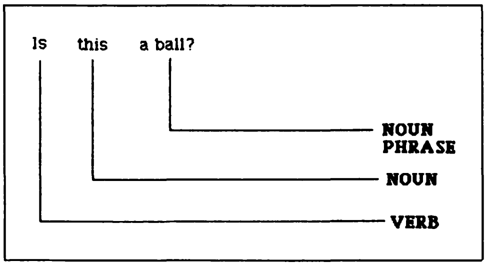


**Figure 1-3. A question's pattern**

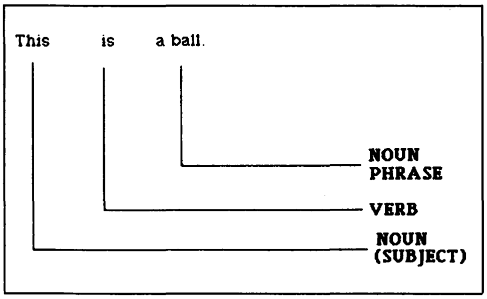


**Figure 1-4. A declaratory sentence's pattern**

the user. It could do that, *provided* the sentence fit into one of the patterns it "knew" about.

Thus our hypothetical computer program could determine the type of each of these sentences:

- Are you a programmer? (question)
- John hit the ball. (declaratory)
- Is she going to the dance? (question)

but would be totally confused by these:

- Watch out!
- Apparently, John hit the ball.
- John and Rick were chasing the bull across the green fields.

Extending the pattern descriptions would, of course, enable the program to identify these sentence structures as well.

We will see pattern-matching emerge as part of the concept behind the programs in Chapters 2–6. It is subtly present in other programs as well.

The most prevalent use of pattern-matching is in the field of computer vision systems, where its application to the robot technology is being widely discussed.

**Learning**

It is frequently argued that a computer program that cannot learn is not really, in the final analysis, intelligent. By learning, the AI community usually means that the program is capable of extending its own knowledge, understanding, and skills—within a limited domain.

In the field of expert systems, perhaps more than any other area of AI research, the ability of the computer program to learn from its experiences and from information given by the user is a characteristic most designers would dearly love to develop.

The question of how people learn is incredibly complex. Researchers have been able to determine that most learning takes place in one of four ways.

First, people may acquire new information by *rote learning*. In school, basic skills—for example, multiplication—are usually taught by rote learning. No processing, thinking, or analysis is required. The learner merely memorizes incoming information for later retrieval and use.

Another method by which people learn is advice taking, or learning by being told. Classroom lectures are often sources of advice-giving learning. Limited processing is required to understand the information being provided and to connect it with previous information, but fundamentally the learner is not required to perform a detailed analysis simply to gain the knowledge.

Learning by example is the most common method of teaching skills and the least used method of teaching ideas. We can learn to tie a shoe, for example, by watching someone else tie many shoelaces. We can learn how to position a telescope by watching an astronomer. We can learn by reading hundreds of chess games how to play the game better. The process of learning and extending knowledge from examples is referred to as *induction*.

Finally, the little-known method of learning by analogy must be briefly considered. People learn a great deal by analogy. The concept is simple: if we know how to do task X and we can see even superficial similarities between it and task Y, we may be able to learn a great deal about task Y simply by analogizing what we know about task X. Not much serious work has been published in the field of AI research on applying this idea to systems and programs.

**Planning/Problem-Solving**

In one sense, all computer programs that are artificially intelligent solve a problem. But in a narrower sense, a great deal of AI research has the ultimate goal of designing programs that analyze a problem, develop an approach to its solution, monitor their progress in solving the problem, and adjust to changing conditions. In the general sense, all the programs in this book are in the planning and problem-solving category. In the narrow sense, however, none of the programs specifically deal with this concept.

AI programs that fall into the problem-solving category of research are goal directed; they have objectives. A plan is a course of action, usually an ordered collection of interrelated goals and subgoals, designed to attain an objective. In an earlier example, the objective was to reach the cheese in the center of the maze and the plan would spell out the steps to be taken to get to the cheese.

In the absence of a plan, programs that merely solve problems are likely to lead to the wrong results, even though their user might consider them extremely valuable. "If you don't know where you're going, you'll probably end up someplace." A problem-solving program that isn't based on a planned approach to the problem may find itself solving the wrong problem or, more likely, approaching the right problem from a wrong direction.

The concepts behind planning programs are so complex and cumbersome that I have largely left them out of this book. Refer to the annotated bibliography in Appendix C for sources of meaningful information on this subject.

## Language Issues in AI

The second major thrust of this book is to provide exposure to the primary languages used in AI programming and research. Chapters 9–13 are devoted to the subject. Chapter 9 contains an overview of the language issues involved. Following that introduction, we examine in some detail each of three key languages: Logo, LISP, and Prolog. We conclude that section of the book with some hints for programmers who are familiar with BASIC, providing some insight and direction into how these programs could be converted to that popular programming language.

The purpose of each chapter on a specific language is to provide the reader with background material and a short refresher course in the basic ideas, commands, structures, and uses of the language. In this way, these chapters serve the purpose of introducing concepts to those who are unfamiliar with the languages at all and affording an opportunity to refresh their skills to those who once used these languages but who have grown "rusty" from lack of use.

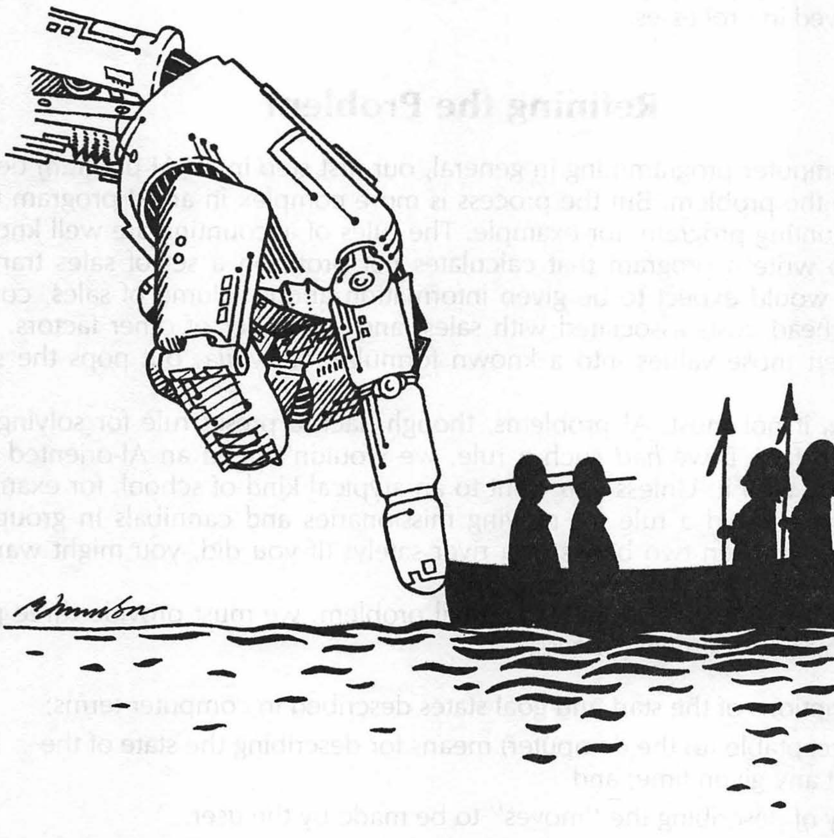
## Conclusion

The broad, slippery, and intriguing field of Artificial Intelligence can be divided into six major areas of research and activity. We will examine four of these in some depth—search, natural language processing, pattern recognition, and expert systems. The other two concepts—learning and planning/problem-solving—are woven into the programs but are not discussed in great depth.

CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELL
ELLIGENCE ARTIFICIAL IN
L INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELL
ELLIGENCE ARTIFICIAL INT
L INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELL
ELLIGENCE ARTIFICIAL INT
L INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELL

# 2

# Classic Missionary-Cannibal Problem



- ○ **Refining Problem Definition**
- ○ **Describing Start and Goal States**
- ○ **Describing Moves in a Problem Set**
- ○ **Choosing Representative Data Structures Appropriately**
- ○ **Basic Search Decision: Brute Force vs. Finesse**

If you have any interest in logic puzzles, you know of the problem posed (and solved) by the program in this chapter. The scenario is simple enough. Three missionaries and three cannibals are on the left bank of a river. They have a boat that will hold two people; any combination of missionaries and cannibals can operate the boat. The objective is to move everyone from the left bank of the river to the right bank of the river.

There's a catch: if the cannibals ever outnumber the missionaries on either bank, they will do their thing and convert the missionaries into a meal.

Solving the problem requires two steps which play critical roles in the world of AI programming: precisely defining the rules of the problem and searching for the solution. The two steps are common to many kinds of problem-solving programs we might design, so we will discuss the basic ideas of both processes as they relate to the missionary-cannibal problem. Along the way, we'll develop a better understanding of the general steps and considerations involved in processes.

## Refining the Problem

As with computer programming in general, our first step in an AI program design is to refine the problem. But the process is more complex in an AI program than in an accounting program, for example. The rules of accounting are well known. If asked to write a program that calculates net profit on a set of sales transactions, you would expect to be given information about volume of sales, cost of sales, overhead costs associated with sales, and a number of other factors. You would insert those values into a known formula and, *voila*, out pops the solution.

Many, if not most, AI problems, though, lack a preset rule for solving the problem. In fact, if we *had* such a rule, we wouldn't need an AI-oriented program to deal with it. Unless you went to an atypical kind of school, for example, you haven't learned a rule for moving missionaries and cannibals in groups of one or two between two banks of a river safely! (If you did, you might want to skip this chapter.)

To deal with the missionary-cannibal problem, we must provide three parts of the problem-refinement process:

- descriptions of the start and goal states described in computer terms;
- an acceptable (to the computer) means for describing the state of the problem at any given time; and
- a way of describing the "moves" to be made by the user.

### Start and Goal State Description

AI professionals use the term "state"—a particular condition or circumstance—to talk about problem-solving programs. An AI problem is in a particular state at any given moment. Any problem can be broken down into discrete and unique states, similar to that shown in Figure 2-1.

The object of problem-solving programs is to transform the initial, or starting, state into the solution, or goal, state. The problem's solution is then described in terms of the start state, the set of intermediate states through which it passes, and the goal state. Each state describes the "space" of the problem.

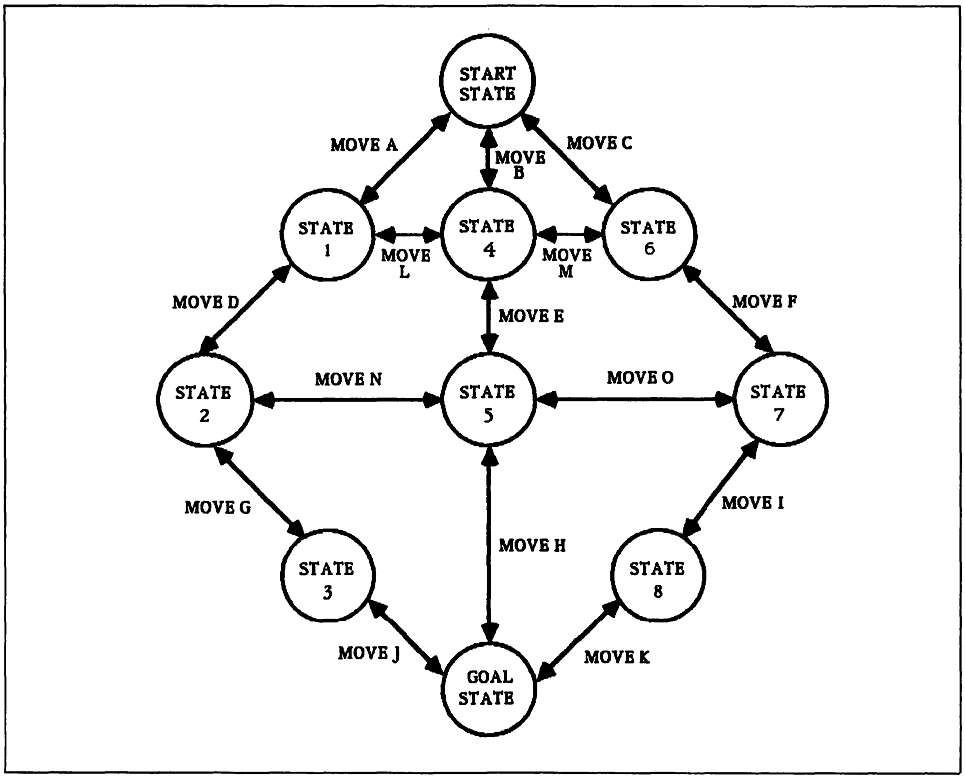


**Figure 2-1. A problem and its states**

Each step between states constitutes a "move." In Figure 2-1, for example, the combination of Start State-Move A-Move D-Move G-Move J-Goal State represents one solution to the problem.

One other point should be made: real-life problems and typical complex AI problems will involve more than one start state and more than one goal state. In other words, the start state may be a function of what has gone on elsewhere in the system and the solution may be described in terms of reaching any of several alternative goal states.

This may seem pretty abstract, so let's look at a concrete example that is often used in teaching basic AI courses and concepts.

***Three-Coins Problem*** This classic puzzle begins with three coins (see Figure 2-2). One coin is placed tails up and the other two are placed heads up. The objective is to position the coins same side up—all heads or all tails. That would be easy—just turn each of the heads coins so the tails up or, easier yet, turn only the coin which had tails showing—except for the additional rule that this must be accomplished in *exactly three* moves.

Figure 2-3 shows a typical state-space diagram for the three-coins problem. Note that it is a more specific version of the general diagram shown in Figure 2-1. We use "H" to mean "heads" and "T" to mean "tails." (This brings up the question of symbolic representation, about which I will have more to say shortly.) Note, too, that there are only three moves possible in the problem we've described. We label them "1," "2," and "3," meaning, respectively, flip over the first coin, flip over the second coin, and flip over the third coin.

Figure 2-2. Three coins in the problem

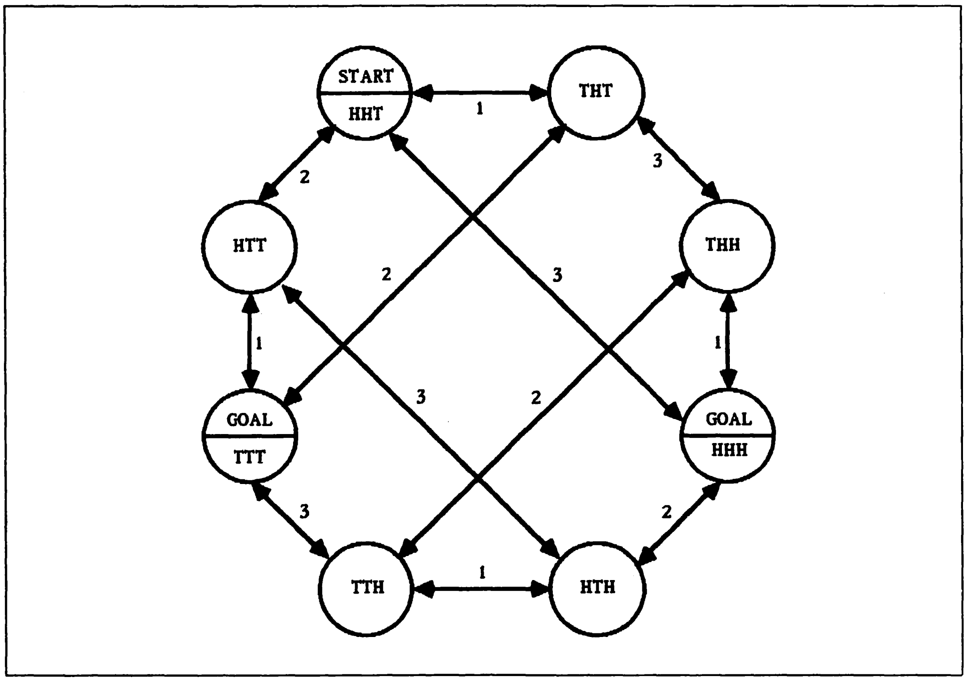


Figure 2-3. State-space diagram of three-coins problem

From each state of the problem's state-space, three moves are possible. If you examine Figure 2-3, you'll find that all three moves lead to three additional states for each state.

Beginning with the start state in the upper left corner of the diagram, if we choose Move 2—i.e., if we turn the middle coin over—we reach the state called HTT. From there, flipping the first coin over—i.e., choosing Move 1—leads to the goal of TTT. Unfortunately, we did not reach that goal in three moves, but in two, so our solution fails to meet a requirement that is outside the space-state diagram of the problem. Similarly, if we start from the start state and choose Move 3, we reach the HHH goal state in one step.

You may have noticed that each state in a state-space diagram can be reached from any state to which it can be converted. In other words, there is a one-move relationship between connected states. In Figure 2-3, for example, we can move from the start state to the HTT state by using Move 2. Similarly, if we found ourselves in the HTT state at some point in our problem and wanted to return to the start state, we could apply Move 2 and do so. This move-for-move correspondence is a key feature of a space-state diagram.

Now a final examination of Figure 2-3 reveals that one way to solve the problem is to apply Moves 1, 3, and 1 in that sequence. There are five other solutions to the problem. Can you find them? If you want that challenge, stop now and try to find them, because I'll provide the answer in the next paragraph.

Four of the five solutions are readily apparent. The new sequences are 1-1-3, 2-2-3, 3-1-1, 2-3-2, and 3-2-2. The other, less obvious solution, is the sequence 3-3-3, a solution which meets the criteria of our problem definition, even if it is a little maddening!

**Describing the State of the Problem**

Now that we can define the problem's start, intermediate, and goal states in a state-space diagram, the next task is to decide how the computer should store the information about the state it is in at any given point. This step involves the concept of *symbolic representation*. A complete discussion is beyond the scope of this book, but we will discuss the concept in a broader sense than the missionary-cannibals problem encompasses.

The field of knowledge representation is of great interest to AI researchers because of the emphasis on natural language processing and expert systems as areas of commercially viable applications of AI. Processing natural language and drawing conclusions from information "known" about a given situation both require the computer to represent this knowledge.

Storing information in a computer requires a data structure. There are many such data structures available. Some depend on the programming language being used but others are more or less common to most programming languages. Some of the more widely available and useful types of data structures include:

- *Lists*. These are collections of individual elements grouped into single data structures. Logo typically encloses these in square brackets. We might have a list called BIRDS represented as [ROBIN CANARY EGRET GULL MACAW PARAKEET].
- *Property Lists*. A special type of list, AI frequently uses property lists. They are almost always part of LISP and Logo languages. This type of list consists of paired attributes and values. For example, a property list called ROBIN might have the attributes SIZE, HABITAT, and KEY_COLOR filled in with the values MEDIUM, NORTH_AMERICA, and RED. Our property list would then look like this: [SIZE MEDIUM HABITAT NORTH_AMERICA KEY_COLOR RED].
- *Arrays*. Strictly speaking, a list is an array and so, by extension, is a property list. What I mean here, though, are arrays that have two or more dimensions. The most common array, perhaps, is the two-dimensional matrix. Figure 2-4 extends the idea of a property list into a two-dimensional array called BIRD_PROPERTIES.
- *Records*. The Macintosh® often stores information on a disk as a collection of one or more records. Each record can contain several different kinds of information. As with arrays and unlike property lists, the attributes—those portions of the data structure that tell us what individual pieces of information

mean or what they relate to—are not stored in the record itself. Instead, the computer program must be told what each item, or field, in a record represents. The record for the robin would look like this:

**MEDIUM,NORTH_AMERICA,RED.**

• *Files*. A file is a collection of records. Each row of the two-dimensional array, BIRD_PROPERTIES, in Figure 2-4 would become a single record in a file called (perhaps) BIRD_ATTRIBUTES. Each record would look like the one described for the robin.

| *Species* | *Size* | *Habitat* | *Key_Color* |
|---|---|---|---|
| ROBIN | MEDIUM | NORTH_AMERICAN | RED |
| CANARY | SMALL | WORLD | YELLOW |
| EGRET | LARGE | SOUTHEAST_TROPICS | VARIES |
| GULL | LARGE | WATER_SALT | WHITE |
| MACAW | LARGE | JUNGLES | VARIES |
| PARAKEET | SMALL | WORLD | VARIES |

**Figure 2-4. An array of bird information**

There are no right or wrong data structures. Rather, selection of the appropriate data structure is a function of several factors, including the kind of data to be stored, the amount of information to be available, the programming language being used, and the need for permanence of the information.

The goal of knowledge representation is to find the most natural way to mirror information considered by the programmer to be important in solving the problem. At the same time, the representation should make efficient use of storage space while granting rapid access to the information when it is needed by the program. Not surprisingly, these goals sometimes conflict with one another. As one writer puts it, "In AI programs, data structures tend to become large and complex. But complex data structures are inefficient, so the tendency is to sacrifice some naturalness and convenience in order to make do with simpler data structures." (Barr 1981, 2:34)

***Describing Three-Coin States*** In making a decision about representation of the knowledge needed by the computer at each step to solve the three-coins problem, we may conclude:

1. The information needed is minimal—one of two states in each of three coins.
2. There is no need for permanence (once the problem is solved, it is solved, and generating the data structures again is quite easy).
3. There are no complex values to be stored.

As we are programming in Logo, the list is the easiest structure to use for such storage. So we define each state as a list containing three elements, each of which would be either an "H" or a "T." The eight possible states shown in Figure 2-3 are then represented as lists which look like this: [H H T] [H T T] [T T T], and so on.

**Describing the Moves**

The problems of describing the moves to be made in the course of solving a problem by a computer program are similar to those involved in making knowledge representation decisions. The goals are naturalness of expression (particularly if the user is to inform the system of the moves rather than having the computer figure them out itself) and economy of programming space and memory. Again, the two are often inconsistent with one another.

Ideally, if the user were to enter information into the computer and have the program determine the consequence of the move—in other words, generate the next state—we would permit the user to type full sentences. "Flip over the first coin" would be the kind of input we would permit. As we will see later, we can in fact achieve something close to such natural language entry, but only at the expense of programming time and placing a sizable burden on the Mac's memory.

On the other hand, we *could* simply require the user to type in one of three numbers to correspond to the moves defined earlier: where "1" means "Flip over the first coin," "2" flips the second coin, and "3" the third coin. Some people would consider this to be perhaps *too* sparse. But if we used this kind of move description language for the computer itself, and permitted the computer to generate the moves and solve the problem, we would find such cryptic descriptions satisfactory.

The question of move description becomes intricate and interesting even in the context of this relatively simple problem. For example, we could define the moves to be variations on the Logo MAKE primitive. We would begin by defining the start state as follows:

```
MAKE START_STATE [H H T]
```

We could then define a move procedure called FLIP which would invert the value of an element of the current state's description from H to T or vice versa. Another move procedure called STAY would define a no-change situation. In that event, the move we've called "1" would be programmed something like this:

```
MAKE STATE [FLIP STAY STAY]
```

This is more cumbersome than other ways we've discussed. The point is that there are a great many ways—some of them quite creative—to express movement from one state to another.

## Applying the Ideas to the Missionary-Cannibal Problem

So much for the theoretical discussion. Let's return to the missionary-cannibals problem. (From now on, we'll use M-C instead of missionary-cannibals.) First, we'll consider the state-space representation issue. Next, we'll turn our attention to the question of symbolic representation of the knowledge contained in the program while it is running. Finally, we'll look briefly at the issue of move description since the computer will be trying to find the solution to the problem in our program.

**Describing the M-C States**

Figure 2-5 shows all 32 possible combinations, including those that result in the loss of the problem's solution because of the cannibals' propensity to eat missionaries—which can arise on either bank of the river.

| | ***Left Bank*** | | | | ***Right Bank*** | |
|---|---|---|---|---|---|---|
| ***Miss.*** | ***Cann.*** | ***Boat*** | | ***Miss.*** | ***Cann.*** | ***Boat*** |
| 3 | 3 | 1 | Start① | 0 | 0 | 0 |
| 3 | 2 | 1 | 2 | 0 | 1 | 0 |
| 3 | 1 | 1 | 3 | 0 | 2 | 0 |
| 3 | 0 | 1 | 4 | 0 | 3 | 0 |
| 2 | 3 | 1 | 5 | 1 | 0 | 0 |
| 2 | 2 | 1 | 6 | 1 | 1 | 0 |
| 2 | 1 | 1 | 7 | 1 | 2 | 0 |
| 2 | 0 | 1 | 8 | 1 | 3 | 0 |
| 1 | 3 | 1 | 9 | 2 | 0 | 0 |
| 1 | 2 | 1 | 10 | 2 | 1 | 0 |
| 1 | 1 | 1 | 11 | 2 | 2 | 0 |
| 1 | 0 | 1 | 12 | 2 | 3 | 0 |
| 0 | 3 | 1 | 13 | 3 | 0 | 0 |
| 0 | 2 | 1 | 14 | 3 | 1 | 0 |
| 0 | 1 | 1 | 15 | 3 | 2 | 0 |
| 0 | 0 | 0 | Goal⑯ | 3 | 3 | 1 |
| 3 | 2 | 0 | 17 | 0 | 1 | 1 |
| 3 | 1 | 0 | 18 | 0 | 2 | 1 |
| 3 | 0 | 0 | 19 | 0 | 3 | 1 |
| 2 | 3 | 0 | 20 | 1 | 0 | 1 |
| 2 | 2 | 0 | 21 | 1 | 1 | 1 |
| 2 | 1 | 0 | 22 | 1 | 2 | 1 |
| 2 | 0 | 0 | 23 | 1 | 3 | 1 |
| 1 | 3 | 0 | 24 | 2 | 0 | 1 |
| 1 | 2 | 0 | 25 | 2 | 1 | 1 |
| 1 | 1 | 0 | 26 | 2 | 2 | 1 |
| 1 | 0 | 0 | 27 | 2 | 3 | 1 |
| 0 | 3 | 0 | 28 | 3 | 0 | 1 |
| 0 | 2 | 0 | 29 | 3 | 1 | 1 |
| 0 | 1 | 0 | 30 | 3 | 2 | 1 |

**Figure 2-5. All possible states of M-C problem**

The list of possible moves that can be made within the constraints of the problem definition is relatively small. In fact, there are only five combinations, as shown in Figure 2-6. Note that the issue of missionaries being eaten in the boat never arises because only two people can be in the boat at a time. Note, too, that we have purposely omitted the boat from each scenario; you can't move without it, so it is unnecessary to include it in each description.

By combining the possible states in Figure 2-5 with the possible moves in Figure 2-6, we could generate a state-space diagram of the M-C problem. A

| Move | Number of Missionaries | Number of Cannibals |
|---|---|---|
| A | 0 | 2 |
| B | 0 | 1 |
| C | 1 | 1 |
| D | 1 | 0 |
| E | 2 | 0 |

**Figure 2-6. All possible moves in M-C problem**

small portion of the diagram that would result is reproduced as Figure 2-7. Each circle represents a state, and has three sets of numbers. The top number corresponds to the state number in Figure 2-5, the second and third describe the situations on the left bank and the right bank. They consist of three values. The first designates the number of missionaries, the second the number of cannibals, and the third whether the boat is present ("1") or not ("0").

If even that small part of the state-space diagram in Figure 2-7 looks like spaghetti, it's because the problem permits a larger number of possible states and moves than the three-coins problem. We have actually illustrated less than one-fourth of the total diagram required to define the M-C problem completely!

The point is not whether we can—or should—construct the complete state-space diagram of a problem, but that we should create at least the concept of the diagram so that we can determine the variety and number of states with which our program has to deal. This makes our decisions about knowledge representation more straightforward and fact based. Let's now turn our attention to those decisions.

**Choosing Representation for the M-C Problem**

Analysis of the possible states of the M-C problem gives at least two ways to represent the state of the problem at any moment.

First, we could simply use three numbers, as we did in Figure 2-5 and in the state-space diagram itself, to represent the state. Thus we would represent two missionaries and two cannibals on the left bank and the other missionary and cannibal on the right bank with the boat by two lists:

```
MAKE LEFT_BANK [2 2 0]
MAKE RIGHT_BANK [1 1 1]
```

Second, we could represent each missionary by the letter "M" in a list, each cannibal by a "C," and the boat by a "B." The same situation would then be represented:

```
MAKE LEFT_BANK [M M C C]
MAKE RIGHT_BANK [M C B]
```

The first method, using numbers, has two advantages. First, it takes up less space and, therefore, less computer memory. Second, we can completely describe any situation by describing only one of the banks. Thus knowing that the left bank is represented as [2 2 0], for example, we can calculate that the right bank's representation for the same setup is [1 1 1]. This method has a disadvan-

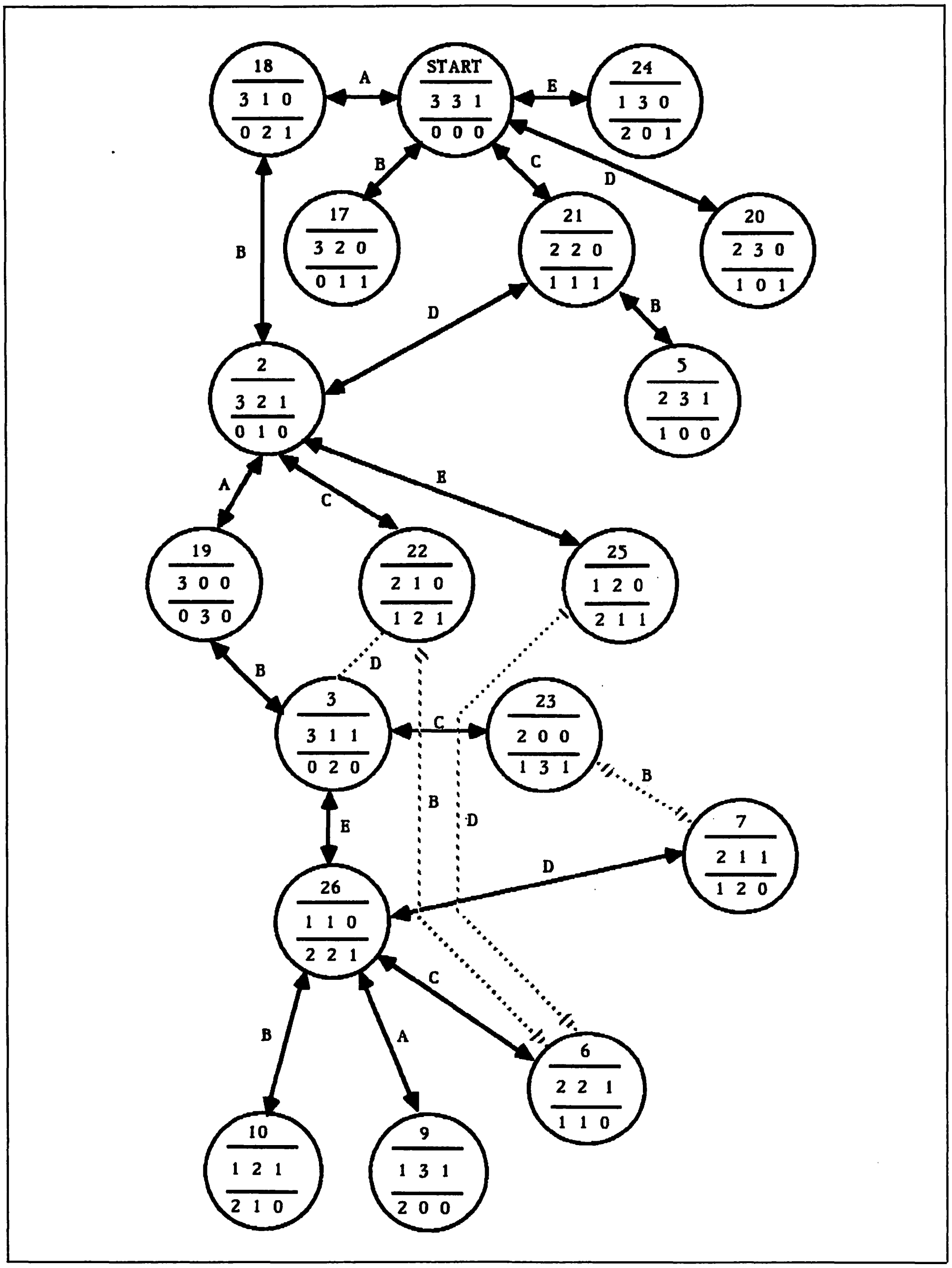


**Figure 2-7. Partial state-space diagram of M-C problem**

tage, though. If we are going to allow the program to move missionaries and cannibals by using the letters "M" and "C" (for reasons of clarity and naturalness), it will be cumbersome to use numbers to represent the states. If, for example, the state of the system is represented as [2 2 0] and the program generates the move [M C], the program will have to determine where the boat is (in this case, it's on the right bank), calculate how many Ms, Cs, and Bs are present (one of each), count the number of Ms and Cs in the move (one of each), subtract each of the moved values from the number in the starting position (yielding, temporarily, a state on the right bank of [0 0 1]), and then perform similar calculations to update the left bank status, including the boat, when the move is complete.

The second method—using a list of letter symbols—is more cumbersome than using numbers. It also does not permit an easy calculation to determine the state of the other side of the bank. At the very least, we'd have to count the number of Ms and Cs involved in the move and on each bank and then perform the mathematical processing to subtract and add Ms and Cs and the B to and from the appropriate banks. That's a fairly indirect way to approach the problem. Using a list, we can simply delete and add new components to each side as events (moves) take place in the problem.

It turns out we need *two* representations of the states of the system. One is the *current* state. That need is best served by using letter symbols to represent each bank. Logo's impressive list-manipulating instructions can manage the current state of affairs and change it. The representation is more natural than the numeric representation, though not as natural as one that uses the words "missionary," "cannibal," and "boat" rather than letters M, C, and B. It also permits direct movement and display of the situation, since we merely work with existing lists.

The second need for representation in the system is to remember the previously experienced states. If the computer didn't know which situations it had already encountered, it could end up trying the same paths and steps repeatedly. Conceivably, it might never get around to trying a path that yields a solution! This need is best served by numeric representation because any one state on the left bank corresponds to one and only one situation on the right bank, so we need only store information about the state of one bank. In addition, since this information will not be used to generate moves but only to check on their uniqueness in the current solution process, it need not be natural at all.

As you will see in our discussion of the program, both approaches represent the state of the problem at a given point in time.

**Describing Moves in the M-C Program**

In large part, deciding how to describe moves inside the Logo version of the M-C program depends on the choice of how to represent data in the system. If we had chosen to use arrays (a viable option in this case) or property lists (probably not workable in this problem), we would have approached the description of moves differently.

Because we chose to use list representations of the states of the system, our method of describing moves is to manipulate those lists.

As you will see when you type in the program and run it, movements are defined by a process made up of the following steps:

1. Select a move from the list of the five legal moves. (We do this selection sequentially to ensure no duplication.)
2. Determine where the boat is. This defines the bank from which the movement must take place.
3. Find out if the move is possible. The move is defined as a list of objects to be moved. For example, [M C B] means to move a missionary and a cannibal. If

no missionaries or no cannibals are on the bank where the boat is, the move is rejected.

4. Remove items in the list to be moved from the list of objects on the bank and add them to the other bank.

The entire moving process consists of manipulating lists—creating them, checking them against one another, deleting information from them, and adding information to them. This describes the entire process by which moves are handled and represented in the M-C program.

This direct method of representing moves is natural; each list presents exactly the items to be moved rather than pointing to them or encoding them somehow. Given the limited nature of the program and the problem to be solved, the method is satisfactory. If there were 25 or 30 different types of objects to be taken into account in a move and hundreds of possible states, we might well wish to use a process that would result in faster execution of the program.

## Using Search Techniques

The second phase of program design involves the selection and use of appropriate search techniques. Searching is a critical part of AI programming. It finds its way into all types of AI systems, whether expert systems, natural language processors, or problem-solving programs.

One authoritative book offers this observation about searching and its importance: "At one time AI researchers believed that the problem of search was the central problem of AI. A parser would search through the possible syntactic structures of a sentence; a game player, through the possible legal moves in a game, etc. People now tend to emphasize the fact that programs with sufficient knowledge of their domains can avoid searching large spaces, but it is recognized that in some cases one will still have to resort to search." (Cherniak et al. 1980, 257). That comment is certainly still valid.

### What Is a Search?

It may seem trivial to define "search." After all, who hasn't lost something and ended up spending time searching for it? Well, a computer search has some things in common with our human searches, but it also has a key difference or two worth noting.

Like a human search, a computer search begins with an initial state and has a goal state. In fact, we could generate a state-space diagram for a human search like the whimsical Figure 2-8. If we did that, we might draw curious looks from our friends and neighbors. But in a sense we do unconsciously draw such a diagram though we take a lot of mental shortcuts in the process. As part of the search process we actually generate each intermediate step between the initial state and the goal state.

"I think I'll look in the garage," defines an intermediate state of determining whether the lost object is among those objects stored in the garage. (If your garage is like mine, that intermediate state might be more accurately labeled "Chaos" instead of "Garage.") We don't, of course, generate the garage itself, but we do create the idea of searching there. In a computer sense, we generate the *state* of looking in the garage.

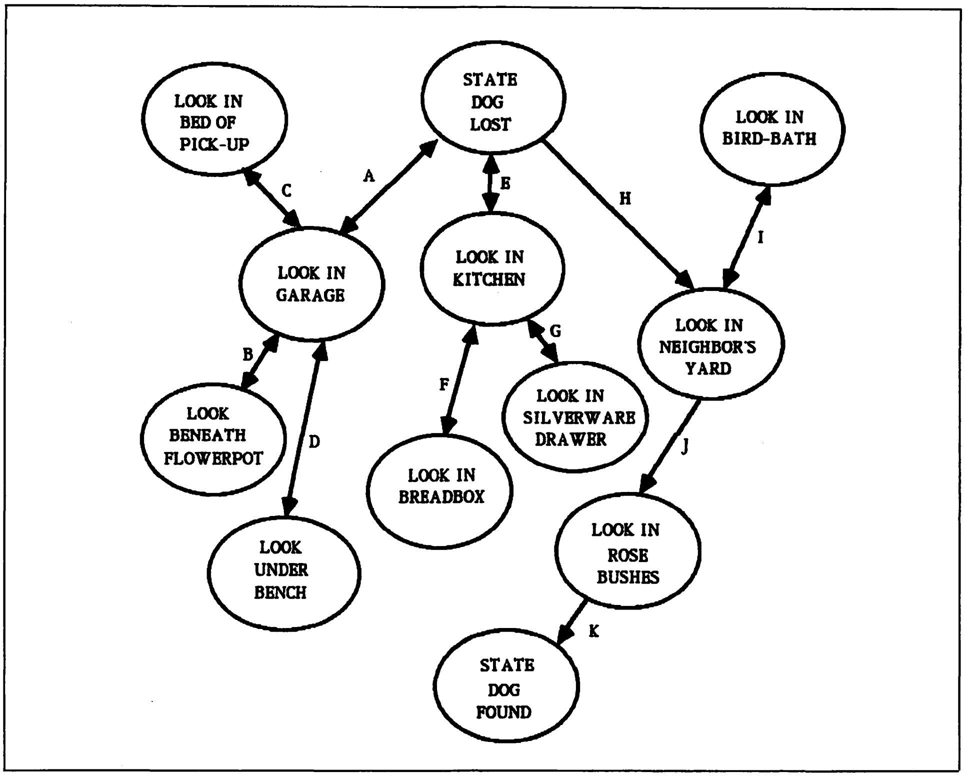


**Figure 2-8. State-space diagram of dog search**

A computer program engaged in a search does the same thing. It begins with a start state and has in its structures a definition of the goal state. In between, it generates new states based on rules, parameters, and understandings it has been given in its programming design, and it examines each to see if it is the solution or if it leads to another step toward the solution.

A human search, though, is more intelligent than any computer search. Humans draw inferences from circumstances which would require huge amounts of memory and mammoth programs to make available in a useful way to a computer. For example, if your dog is a Doberman, the chances of it hiding under a flowerpot in the garage are pretty slim (no pun intended). Storing the information in a computer program that a Doberman is a type of dog and that it is larger than a flower pot, for example, won't stop the program from searching in a breadbox, which is, after all, larger than a flower pot. Computer searches, even those that apply machine intelligence concepts, are more inclined to use techniques and processes that humans would discard immediately. This, among other things, prevents computer programs from being as intelligent as we might expect.

## Types of Search: An Overview

We will return to the subject of searching several times in this book, each time adding more to our fund of knowledge about the subject. But for the moment, let us take a brief look at some key search techniques which are widely utilized in AI programming.

***The "Brute Force" Method*** The Missionary-Cannibals program in this chapter uses the "brute force" search method. It simply tries every possible path at every possible point in the state-space diagram of the problem until it reaches a solution, breathes a heavy sigh, and quits.

Brute force search is, as you can imagine, inefficient. Many blind alleys are pursued. With no means of ensuring that the program didn't repeat unsuccessful routes and patterns, the program could theoretically search forever without finding a solution. Our M-C program computer "recognizes" that it has tried a particular path previously without success. Although this clearly constitutes a useful modification of the brute force search method, it hardly approaches the addition of intelligence to the search process.

***Intelligent Search Concepts*** Search techniques which use intelligence often undertake their assignments using such approaches as:

- "best-first" searching, in which evaluation takes place at each state in the state-space and findings of the search to that point are compared with findings from other partial searches; the most promising route is then pursued for one more level, and so on
- "depth-first" searching, in which the program follows a given path to its ultimate conclusion and, if unsuccessful, backtracks to the next earliest level and checks the next path until it finally finds a solution
- "breadth-first" searching, in which each possible path is followed to one level of depth in the search space and then each is pursued one additional level, and so on, so that the conclusion is reached when the goal state is discovered in a horizontal movement of the search process.

Chapter 3 presents the "Micro-Logician" program and we will discuss more about search techniques and how they differ from one another then. As with knowledge representation, there are no "right" and "wrong" search techniques. Selection of a search technique derives from a host of factors including size of the state space to be searched, the language being used, machine limitations, and need for speed.

### Search in the M-C Problem

In designing the M-C program, we realized that the number of states to be searched is relatively small and certainly manageable within the constraints of the Mac's 512K memory. Speed is not significant, since our primary objective is to learn about AI programming techniques and not to create commercially salable programs. So we used a brute force method, modified to avoid repeating previously tried paths.

We will describe the search processing itself when we reach that part of the program description.

## The Missionary-Cannibal Program

Figure 2-9 is a box diagram of the program called, appropriately enough, SAVE_MISSIONARIES. The main procedure calls the SETUP_PROBLEM routine, which initializes some key variable information and returns control to the main procedure. The SHOW__STATUS procedure is then called to display the starting situation: three missionaries, three cannibals, and the boat, all on the left bank. The program then calls the main "workhorse" procedure, TRY, which determines whether or not a solution has been found (one always is, since we have designed it that way, but for debugging purposes, leaving the alternative possibility in the program is a good idea.)

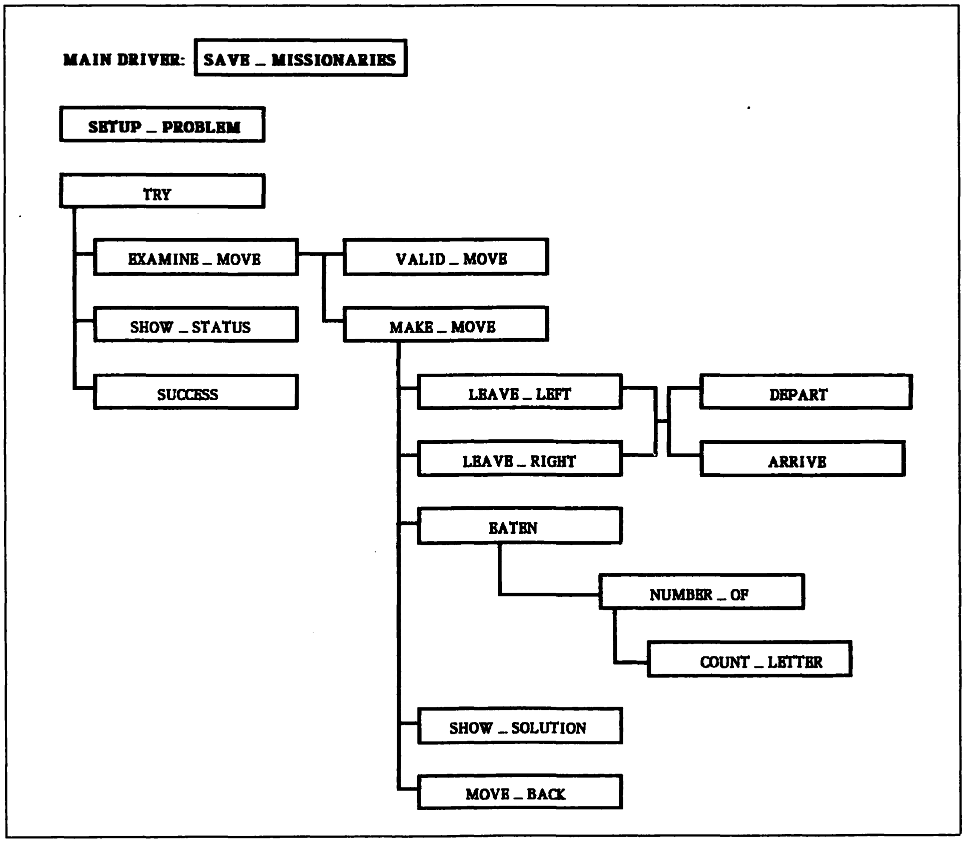


**Figure 2-9. Box diagram of missionaries and cannibals**

## SETUP__PROBLEM Procedure

This procedure initializes variables describing each of the banks at starting position (BANKL and BANKR, for left and right bank, respectively). It then sets up a variable called ALL__MOVES, which is a list of lists, each element of which is one of the five possible combinations of legal moves that can be made in a given situation. Finally, it sets up the variable BEEN__HERE, which keeps track of each position the program encounters to ensure that blind alleys aren't continually tried.

## SHOW__STATUS Procedure

Besides displaying the left and right bank status at the end of each move, SHOW__STATUS affords a convenient place to update the variable BEEN__HERE. We simply place the contents of the left bank into the list variable BEEN__HERE.

**TRY: Second Main Driver Procedure**

Once everything is set up, the main procedure passes control to a collection of procedures. The main driver, TRY, is called with the ALL_MOVES variable so that it is passed a list of all legal moves. TRY goes through the list one move at a time using the EXAMINE_MOVE procedure, which we will soon discuss, to determine if a move is legal and, if so, what its effect will be.

After each move, TRY checks to see if a solution has been found. TRY calls the SUCCESS procedure, which outputs TRUE if a solution *has* been found (i.e., the left bank is empty) and nil if the problem has not been solved. If a solution has not been found, TRY displays the status of the left and right banks at that point using SHOW_STATUS.

When a move has been successfully made with no missionaries being eaten and no solution being found, TRY calls itself again with the list of legal moves called ALL_MOVES to work through another sequence of potential moves from the now-changed state of the problem.

**EXAMINE_MOVE Procedure**

The procedure EXAMINE_MOVE first calls another procedure, VALID_MOVE (discussed in the next section) to determine if the move is legal. If so, VALID_MOVE prints "true"; otherwise, it passes NIL to the calling procedure, EXAMINE_MOVE. If the move isn't valid for the circumstances, EXAMINE_MOVE stops and control returns to TRY.

Similarly, if the move creates a situation that the program has previously seen or if the procedure EATEN indicates that a missionary has been lost, the program calls the procedure MOVE_BACK, which retracts the move. Control then returns to TRY.

If the move is legal and does not result in a lost missionary, then the program sets a variable called SOLUTION to be "true." This variable is used throughout the program as a means of loop control, not to indicate that a solution has necessarily been found (though it would also be used in that situation).

**VALID_MOVE Procedure**

This procedure determines which bank will be the departure bank by identifying the boat's location. It then checks to see if the move being tested is legal by ensuring that the number of missionaries and cannibals being moved is less than or equal to the number of missionaries and cannibals on the bank at the moment. Note that we have used the construction:

```
IF AND
    (OR  ((NUMBER_OF 'M :MOVE) < (NUMBER_OF 'M :BANK))
         ((NUMBER_OF 'M :MOVE) = (NUMBER_OF 'M :BANK)))
    (OR  ((NUMBER_OF 'C :MOVE) < (NUMBER_OF 'C :BANK))
         ((NUMBER_OF 'D :MOVE) = (NUMBER_OF 'C :BANK)))
```

instead of the ExperLogo® construction:

```
IF AND (OR  ((NUMBER_OF 'M :MOVE) ≤ (NUMBER_OF 'M :BANK))
            ((NUMBER_OF 'C :MOVE) ≤ (NUMBER_OF 'C :BANK))))
```

This is because most Logo implementations do not include a single less-than-or-equal-to symbol like ExperLogo's ≤.

***EATEN Test Procedure*** The EATEN procedure uses similar AND/OR logic to that in VALID__MOVE to ensure that the number of cannibals on one bank is not greater than the number of missionaries. (Note that it must also check to be sure there aren't "no missionaries," because then it wouldn't matter if there were more cannibals; they'd have nothing to eat!)

**LEAVE__LEFT and LEAVE__RIGHT**
The LEAVE__LEFT and LEAVE__RIGHT procedures are identical except for the arguments sent to the next-level procedures, DEPART and ARRIVE. Each sets up a variable called TEMP__BANK to hold information during processing. This makes it possible to move things around without disturbing variables that may be needed for further evaluation.

After completing this setup, these procedures call on the DEPART procedure, which builds a list of who is left on the bank from which the boat is leaving, places that in the variable TEMP__BANK, and returns control to the calling procedure. There, TEMP__BANK's contents are swapped into the departure bank's list so that it contains an updated description of who is left on the bank following the departure.

TEMP__BANK is now reinitialized to be an empty list and the procedure ARRIVE is called. This procedure is more complex than DEPART because it must keep track of the *order* of missionaries and cannibals in the boat when moving from the right bank to the left so that the variable BEEN__HERE will contain accurate and pattern-matchable information about what has gone before.

Figure 2-10 reproduces the listing which appears in the ExperLogo® Listener Window (the Text Window in other Logos) as the program solves this puzzle. It is self-explanatory.

## Exploring AI with Missionaries and Cannibals

The program we have been discussing here cannot be greatly modified since its knowledge domain is limited to "knowing" about moving cannibals and missionaries around. But a couple of things might make the program more *interesting*.

For one thing, a random number generator could pick our moves instead of going through the same sequence each time. Over time, this should result in the program finding many solutions to the problem. Another approach would be to place the lists in the variable ALL__MOVES in another order to see how that affects the decision making of the program.

If you're interested in seeing in greater depth how the program draws its conclusions, change the program so that when it runs into an illegal move or one that results in a missionary's demise, it prints a message indicating what move it tried and what the result was. This will, of course, make the program run more slowly but it may prove helpful in trying to figure out how the program works.

You might also try changing the number of missionaries and cannibals or the number of people the boat can hold—though the results will be unpredictable with some combinations. When you make these changes, be sure to think about what a state-space diagram would look like in identifying the possible legal moves.

```
SAVE_MISSIONARIES
The left bank is host to:  M M M C C C B
The right bank is host to:  nil
move C C B from left bank to right bank.

The left bank is host to:  M M M C
The right bank is host to:  C C B
move C B from right bank to left bank.

The left bank is host to:  M M M C C B
The right bank is host to:  C
move C C B from left bank to right bank.

The left bank is host to:  M M M
The right bank is host to:  C C C B
move C B from right bank to left bank.

The left bank is host to:  M M M C B
The right bank is host to:  C C
move M M B from left bank to right bank.

The left bank is host to:  M C
The right bank is host to:  C C M M B
move M C B from right bank to left bank.

The left bank is host to:  M M C C B
The right bank is host to:  M C
move M M B from left bank to right bank.

The left bank is host to:  C C
The right bank is host to:  M M C M B
move C B from right bank to left bank.

The left bank is host to:  C C C B
The right bank is host to:  M M M
move C C B from left bank to right bank.

The left bank is host to:  C
The right bank is host to:  M M M C C B
move C B from right bank to left bank.

The left bank is host to:  C C B
The right bank is host to:  M M M C
move C C B from left bank to right bank.

The left bank is host to:  nil
The right bank is host to:  M M M C C C B
I DID IT!
I DID IT!
```

**Figure 2-10. How the problem gets solved**

## Summary: What We've Learned about AI Programming

In this chapter, we've taken our first steps in learning about Artificial Intelligence and the considerations behind intelligent programs. We've covered a great deal of new material here, including:

- defining the state of a problem and the goals associated with its solutions;
- making decisions about how to represent the knowledge contained in a problem's "world" system;
- accurately and concisely describing moves from one state to another in ways which are consistent with programming language syntax.

In the next chapter, "Micro Logician," we will further explore the ideas of searching.

```
              {Missionaries & Cannibals ©1985, The Waite Group}
                        {Logo program by Ken Schieser}

{Main calling procedure}
  TO SAVE_MISSIONARIES
    SETUP_PROBLEM
    SHOW_STATUS
    TRY :ALL_MOVES
    IF EQUALP :SOLUTION :FOUND
      [PR [I DID IT!]]
      [PR[SOLUTION NOT FOUND!]]
END

{Sets initial conditions}
TO SETUP_PROBLEM
  {FOUND is a boolean variable. It holds a value of true throughout the program. Its
main purpose is to make the program readable.}
  MAKE FOUND 'TRUE
  MAKE BANKL [M M M C C C B]
  MAKE BANKR []
  MAKE ALL_MOVES [[C C B][C B][M B][M C B][M M B]]
  MAKE BEEN_HERE[]
END
{Shows status, adds contents of left bank to BEEN_HERE, initializes loop control
variable}
TO SHOW_STATUS
  PR SE [The left bank is host to:] :BANKL
  PR SE [The right bank is host to:] :BANKR
  {Loop control variable see TO TRY & TO EXAMINE}
  MAKE SOLUTION NIL
  MAKE BEEN_HERE LPUT :BANKL :BEEN_HERE
END
```

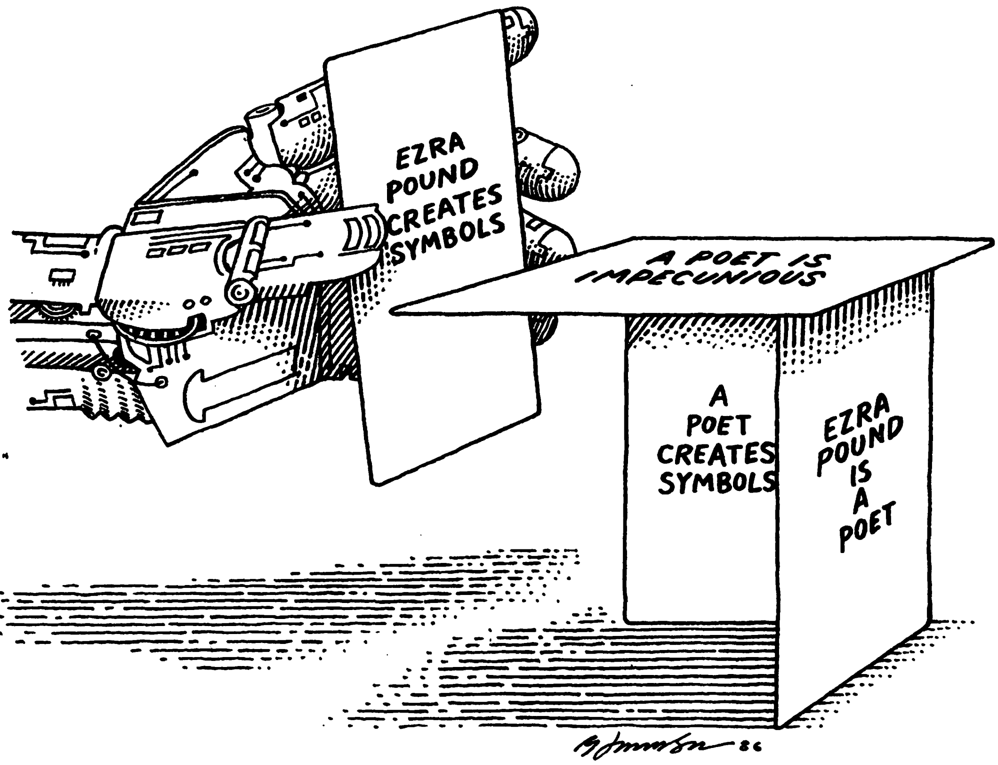
```
{Prints out the correct move}
TO SHOW_SOLUTION
  IF MEMBERP 'B :BANKR
    [MAKE BANK1 [left bank] MAKE BANK2 [right bank.]]
    [MAKE BANK1 [right bank] MAKE BANK2 [left bank.]]
  PR (SE [Move] :MOVE [from] :BANK1 [to] :BANK2)
  PR<<>>
END

{Recursively test all moves}
TO TRY :MOVES
  IF EMPTYP :MOVES [STOP]
  {The variable MOVE, used in all lower level procedures, is made to be the first set of
the variable MOVES (or ALL_MOVES, since TRY is called with the argument
ALL_MOVES)}
  MAKE MOVE FIRST :MOVES
  EXAMINE_MOVE
  {Test to see if a solution has been found—if not TRY is called recursively without the
first member of the set MOVES. This continues until the solution is found or MOVES
becomes the empty set. The stop is needed to prevent complex recursion.}
  IF EQUALP :SOLUTION NOT :FOUND [TRY BF :MOVES STOP]
  {If the program has made it past the above test, a solution has been found. At this
point SHOW_STATUS is called—printing out the new banks & reinitializing the loop
control variable}
  SHOW_STATUS
  {If the left bank is empty, the second level is stopped; control is passed back to the
main level}
  IF SUCCESS [MAKE SOLUTION :FOUND STOP]
  {If the left bank is not empty, TRY is called again with an argument of :ALL_MOVES
  TRY :ALL_MOVES
END

TO SUCCESS
  IF EMPTYP :BANKL [OP 'TRUE][OP NIL]
END

{Third level}
TO EXAMINE_MOVE
  {If the move contains characters that are not members of the bank to be moved from,
control is passed back to second level}
  IF NOT VALID_MOVE [STOP]
  MAKE_MOVE
  {Test for repetition}
  IF MEMBERP :BANKL :BEEN_HERE [MOVE_BACK STOP]
  {Test for more cannibals on either bank}
  IF EATEN [MOVE_BACK STOP]
  {If either of the preceding tests pass, control is handed back to second level. If not,
the loop control variable FOUND is made "true" and the solution is printed out}
```

```
  MAKE SOLUTION :FOUND
  SHOW_SOLUTION
END
{Function: Outputs "true" or "nil"} TO VALID_MOVE
  {A variable BANK is made equal to the bank with the boat}
  IF MEMBERP 'B :BANKL [MAKE BANK :BANKL][MAKE BANK :BANKR]
    IF AND
    (OR ((NUMBER_OF 'M :MOVE)<(NUMBER_OF 'M :BANK))
         ((NUMBER_OF 'M :MOVE)=(NUMBER_OF 'M :BANK)))
    (OR ((NUMBER_OF 'C :MOVE)<(NUMBER_OF 'C :BANK))
         ((NUMBER_OF 'C :MOVE)=(NUMBER_OF 'C :BANK)))
      [OP 'TRUE][OP NIL]
END
{Function: Outputs "true" or "nil"}
TO EATEN
  IF OR
  (AND((NUMBER_OF 'M :BANKL)>0)
       ((NUMBER_OF 'M :BANKL)<(NUMBER_OF 'C :BANKL)))
  (AND((NUMBER_OF 'M :BANKR)>0)
       ((NUMBER_OF 'M :BANKR)<(NUMBER_OF 'C :BANKR)))
  [OP 'TRUE][OP NIL]
END

  {Function: Outputs a number}
  TO NUMBER_OF :LETTER :COLLECTION
    MAKE NUM 0
    COUNT_LETTER :LETTER :COLLECTION
    OP :NUM
END

  {Counts letters in collection}
  TO COUNT_LETTER :LETTER :COLLECTION
    IF EMPTYP :COLLECTION [STOP]
    IF EQUALP FIRST :LETTER FIRST :COLLECTION [MAKE NUM :NUM + 1]
    COUNT_LETTER :LETTER BF :COLLECTION
  END

  {Same as MAKE_MOVE, purpose: readability}
  TO MOVE_BACK
    MAKE_MOVE
END

  TO MAKE_MOVE
    IF MEMBERP 'B :BANKL [LEAVE_LEFT][ LEAVE_RIGHT]
END

TO LEAVE_LEFT
  MAKE TEMP_BANK []
  DEPART :BANKL :MOVE
  MAKE :BANKL :TEMP_BANK
  MAKE TEMP_BANK[]
  ARRIVE :BANKR :MOVE
  MAKE BANKR :TEMP_BANK
END

  TO LEAVE_RIGHT
  MAKE TEMP_BANK []
  DEPART :BANKR :MOVE
  MAKE BANKR :TEMP_BANK
  MAKE TEMP_BANK []
  ARRIVE :BANKL :MOVE
  MAKE BANKL :TEMP_BANK
END

{Places only those members of B that are not in M into TEMP_BANK}
TO DEPART :B :M
  IF EMPTYP :B [STOP]
  IF EMPTYP :M
    [MAKE TEMP_BANK LPUT FIRST :B :TEMP_BANK
    DEPART BF :B :M STOP]
  IF EQUALP FIRST :B FIRST :M
    [DEPART BF :B BF :M STOP]
  MAKE TEMP_BANK LPUT FIRST :B :TEMP_BANK
  DEPART BF :B :M
END

{The order of arriving to the right is not important, but it is crucial when arriving to the
left. ARRIVE places all the Ms together, Cs together, and puts the boat on the end}
TO ARRIVE :B :M
  IF EMPTYP :M [STOP]
  IF EMPTYP :B
    [MAKE TEMP_BANK LPUT FIRST :M :TEMP_BANK
    ARRIVE :B BF :M STOP]
  IF EQUALP FIRST :B FIRST :M
    [MAKE TEMP_BANK LPUT FIRST :M :TEMP_BANK
    MAKE TEMP_BANK LPUT FIRST :B :TEMP_BANK
    ARRIVE BF :B BF :M STOP]
  MAKE TEMP_BANK LPUT FIRST :B :TEMP_BANK
  ARRIVE BF :B :M
END
```

CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI

# 3

# Micro-Logician



- Backward-Chaining in Search Strategies
- Elementary Pattern-Matching in Natural Language Processing
- Basic Logic Processing
- Using Limits to Language Entries for Manageable Programs

The idea of searching is so important that we have included this program as another example of how it is done. This chapter adds one significant concept to the Missionaries and Cannibals program in Chapter 2: backward-chaining. Along the way, we'll present some ideas about natural language processing, which will be a primary topic of several of our programs.

## What You'll Learn

As you read this chapter, type in the program it discusses. When you run and analyze that program, you'll appreciate the importance of selecting the right approach to searching through the possible solutions to a problem. More specifically, you'll understand the significance of the differences between forward- and backward-chaining, which relate to how we search for a solution to a problem.

The ideas of forward- and backward-chaining have gained prominence as expert systems have become commercially available. Promoters of specific expert system development tools tout their products as forward-chaining, backward-chaining, or combination products, which do chaining of both kinds. As we will see, the differences between these kinds of chaining are technical rather than qualitative.

## The Program

This chapter will analyze a program called "Micro-Logician." The program seems intelligent because it accepts information and then answers questions about that information that it can only know about indirectly. In other words, it appears to draw conclusions and inferences from facts without having been told the conclusions.

In handling inquiries about fact patterns, the program performs backward-chaining so that it can determine where, if at all, in its "knowledge base" it has the data needed to answer the question.

This chapter will also demonstrate the concept of pattern-matching as it is used in natural language processing. We'll see how we can determine content by examining the pattern and structure of input from the user. (See Figure 3-1.) (We will discuss the idea of pattern-matching using such templates as predetermined sentence patterns in more detail when we reach the subject of natural language later in our study. The current program simply demonstrates that idea.)

### What the Program Does

Micro-Logician permits the user to enter three kinds of input:

1. statements of fact (for example, "The house is large.")
2. questions (for example, "Is a robin a bird?")
3. general inquiries (for example, "What do you know about television?")

The program uses statements of fact provided by the user without trying to evaluate their truth or practical value. The statements of fact form the basis for building a set of *property lists* in the program's memory. Property lists are an important characteristic of Logo and of other languages which are best suited for AI programming tasks. They provide a convenient way for the program to "remember" information passed to it by the user.

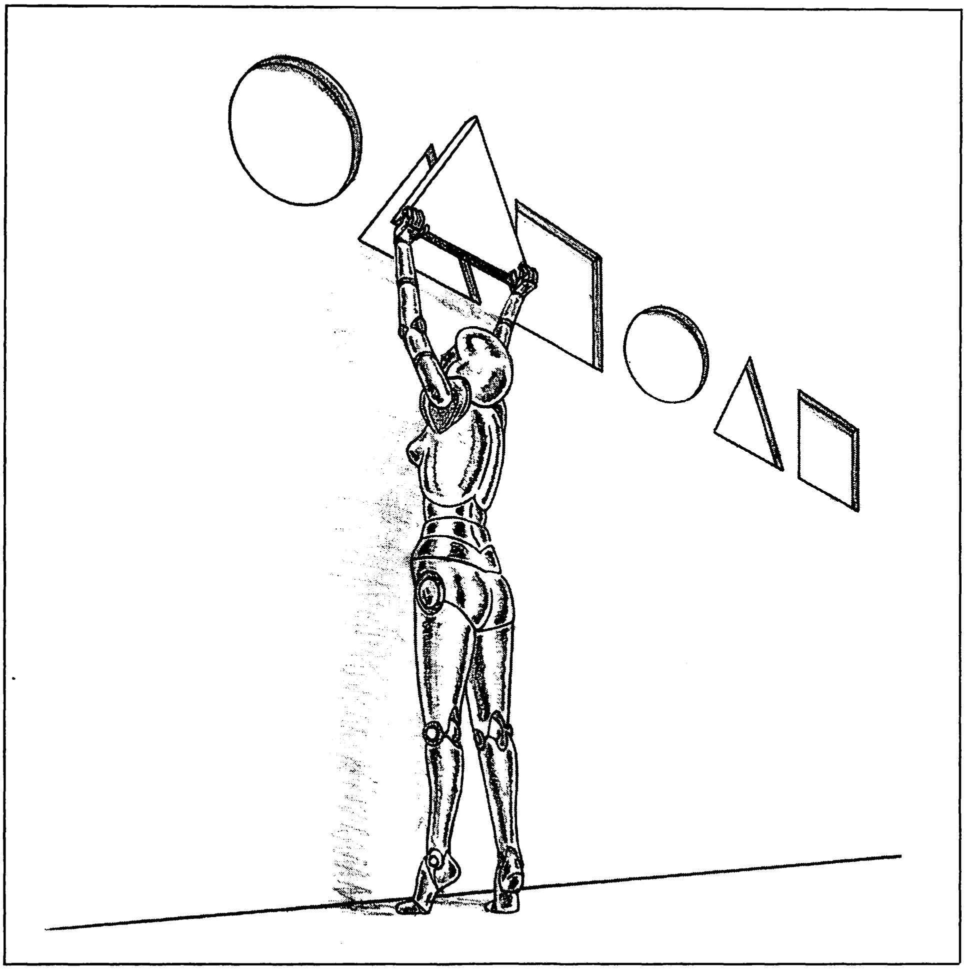


**Figure 3-1. Pattern-matching: fitting incoming ideas to predefined structures**

If the user types in a question, the computer recognizes it as a question, searches through its property lists to see if it knows anything about the subject, and, if so, tries to reach a conclusion as to whether the question being asked should be answered "yes" or "no" or "I don't know."

When the user types in a sentence starting with the word "Inquire"—for example, "Inquire about taxes"—the program searches through its list of topics to see if it knows anything about the subject. If it does, it performs a "memory dump," simply providing a list of all the facts it has been told about the subject.

The program is terminated by the user typing "Quit" at the outermost procedural block when being asked for an entry.

**Narrowing the Domain of Entries**

Ideally, a program like Micro-Logician would accept all kinds of statements and questions from its user. Obviously, it can understand such simple statements as "The box is blue." But it should also be able to handle complex sentences and thoughts like, "The big box over in the corner, which belongs to Steve and has a pink bow on it, is blue." After all, both sentences involve the same essential data—there is a box and its color is blue.

We will see in our discussions of programs that are part of the AI discipline of natural language processing that such a capability would place a huge demand on the resources of the Macintosh® or any other desktop personal computer. We must, therefore, be willing to live with something less than total comprehension.

Micro-Logician doesn't limit the *subjects* about which the computer will be told; the subject is under the user's control. The program will, though, limit the kinds of sentences to which it will respond.

***Factual Statements*** Statements of fact will always require the following format:

SUBJECT IS PREDICATE.

The subject must be simple, rather than compound or complex. It may, however, include an article (a, an, or the). The following will be acceptable subjects in a factual statement entered into Micro-Logician:

*A writer* is often out of cash.

*Computing* is a fine art.

*The box* is in the corner.

The following subjects will not work:

*Writers* are often out of cash. (The subject is plural. Since it requires a verb other than "is," it won't be recognized.)

*Computing with a Macintosh®* is a fine art. (The compound subject won't be understood.)

*The big box* is in the corner. (The adjective "big" makes the subject unacceptable to Micro-Logician.)

***Singularly Important*** Only the verb "is" will be understood by our micro version of this logical program. This simplifies the programming though it limits the input that the program can understand and use.

***Verbal Freedom*** The predicate—everything after the word "is"—can be convoluted or complex and the program will still accept it. The program, however, may not *understand* it as we intended. An example or two will clarify what we mean here.

Look at the sentence:

The programmer is tall and thin.

Anyone listening to that sentence understands that the word "and" means that the programmer is *both* tall and thin. If we heard that sentence and were then asked, "Is the programmer thin?" we would answer "yes." But Micro-Logician doesn't understand words like "and" to have logical value; as far as the program is concerned, logical connectives are just words. Thus, if we ask Micro-Logician, after entering the above sentence, "Is the programmer thin?" it will insist he is not. If we ask, on the other hand, "Is the programmer tall and thin?" the program will agree that he is both of those things.

***A Concluding Mark*** Every sentence or question typed into Micro-Logician *must* close with a period, question mark, or other punctuation. The punctuation need not be *correct*, but must be present.

**Logical Questions**

With that discussion behind us, we can quickly examine the format requirement of the other two types of sentences with which Micro-Logician has been programmed to cope. A logical question must take the form:

```
IS SUBJECT PREDICATE?
```

The first word *must* be "is." The subject, therefore, must be simple. The program will consider everything after the subject to be a predicate defining the character trait of the subject.

**General Inquiries**

We have adopted the word INQUIRE to signal Micro-Logician that what follows is an inquiry about the information it has stored on the subject.

The required form for such an inquiry is:

```
INQUIRE SUBJECT.
```

The subject may contain an article (though it almost never makes sense to include one) but otherwise must be simple. No predicate containing the trait involved is required since the inquiry elicits *all* available information about the subject.

For example, we could type in the following query:

```
INQUIRE WRITER.
```

The program would then tell us every fact it knows about writers. Assuming we had stored the information, it would respond:

```
WRITER IS AN INTELLECTUAL.
WRITER IS POOR.
WRITER IS LONELY.
```

## How Micro-Logician Works

As shown in the box diagram in Figure 3-2, Micro-Logician is divided into six Level 2 procedures: SET.UP, CLEAN.UP, SCANNER, ADD.A.FACT, INQUIRY, and LOGIC.FINDER.

Micro-Logician lacks the "pure" main program that most of the other programs in this book have. That is because we have two different kinds of setup or housekeeping chores to do at the beginning and we want to call one of the sets only once. So the start of the program calls for us to type in the command SET.UP. That routine in turn calls the main driver routine MICRO.LOGIC. In a sense, MICRO.LOGIC is the main procedure.

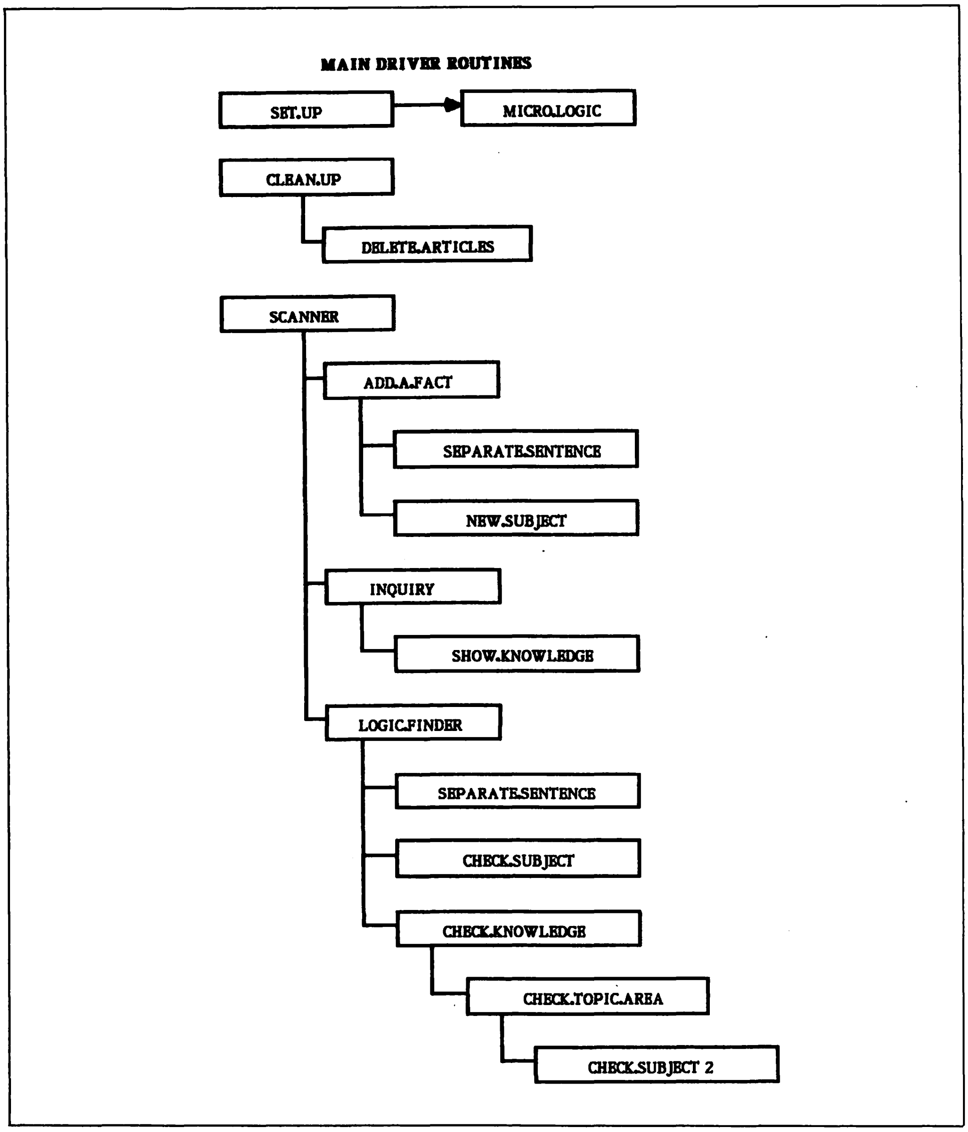


**Figure 3-2. Box diagram of Micro-Logician**

## SET.UP

The SET.UP procedure is the Micro-Logician initialization routine. It initializes variables which will be needed through the rest of the program. Specifically, the Logo implementation of the SET.UP routine defines the articles "a," "an," and "the" as articles for later deletion and initializes the list called TOPICS to be the empty list. Ultimately, this list will store topics the program "knows" about based on subsequent user input. The SET.UP procedure is called by the user typing in its name.

The program is kept running by the MICRO.LOGIC procedure, which is called by other procedures. We can restart the program during a run of Micro-Logician by recalling SET.UP, which reinitializes the topic list. We can also call the MICRO.LOGIC routine in order to *continue* processing after some kind of halt without the program "forgetting" all it had been told.

**CLEAN.UP**

After SET.UP and before CLEAN.UP, we invoke the main procedure, MICRO.LOGIC. This routine takes care of our request for user input and stores the routine to check to see if the user is done. It would be equally acceptable to design another procedure—perhaps called GET.INPUT—to handle input and at the same time check to see if the user is finished. Because of the way Logo handles the STOP command, however, our current method is the more straightforward and easier to implement.

The MICRO.LOGIC procedure also initializes a temporary holding variable called SENT2 which must be reset each time through the program's main procedures.

The CLEAN.UP procedure puts the English-like data entered by the user into a more usable form. The procedure strips the punctuation mark at the end of the sentence and then scans the sentence for articles and removes them. When this CLEAN.UP procedure is completed, the sentence:

THE BOX IS A CUBE.

has been transformed into:

BOX IS CUBE.

This enables the program to store information efficiently about the subject "box."

**SCANNER**

The procedure SCANNER determines which of the three types of sentence—factual statement, logical question, or general inquiry—has been entered by the user.

Since we have arbitrarily fixed the formats of each type of input, SCANNER has a quite simple task. It looks at the first and second words of the sentence that was entered. The first word is placed into a variable called KEY1 and the second word into the variable KEY2.

If KEY1 is "is," then the program knows it is faced with a logical question. If it is "inquire," it knows that it is going to be asked to handle a general inquiry. Any other input could be designed to add a factual statement to the program's property lists. But we check explicitly for the word "is" as the second word in our stripped-down sentence, just to be sure. Micro-Logician will be able to handle a sentence like:

The programmers are always broke!

quite nicely without doing something unexpected or unwanted.

If the sentence turns out to be a statement of fact, the procedure called ADD.A.FACT is invoked. If a logical question is being posed, the LOGIC.FINDER procedure is called into action. General inquiries are handled by the procedure called INQUIRY.

**ADD.A.FACT**

The ADD.A.FACT procedure is actually divided into two procedures. Besides ADD.A.FACT, itself, there is a subordinate, Level 3, procedure called NEW.SUBJECT.

ADD.A.FACT first sets up the subject and predicate. It now begins to refer to the predicate by its more proper name "trait." The procedure takes the first word of the sentence as the subject and everything after the word IS as the trait. Using the Logo primitives FIRST and BUTFIRST, this process is quite simple.

The procedure determines if the subject is a new one or if new data is being provided concerning a topic it already knows about. It does this by checking to see if an attempt to read information about the subject results in an answer of "nil." Nil in Logo means either that nothing is found or that an answer is false. When nil is returned, ADD.A.FACT calls on the procedure NEW.SUBJECT. Otherwise, it adds the value "1" to the number of facts it knows to be available about the subject and tacks the new information onto the end of the property list.

***NEW.SUBJECT*** The first time Micro-Logician encounters a particular subject, the NEW.SUBJECT procedure adds it to the variable list TOPICS, which keeps track of the subjects it knows about. It then puts the number "1" in place as the number of pieces of information—or traits—stored on the subject and places the trait in the property list.

**INQUIRY**

Micro-Logician permits two types of inquiries. The simpler is a request for all information available on a given subject. This assignment is handled by the Level 2 procedure INQUIRY. It in turn has a Level 3 procedure called SHOW.KNOWLEDGE.

The sentence has to begin with the word "INQUIRE" in order to be classified as a general inquiry sentence. INQUIRY then begins by defining the subject of the inquiry as everything following INQUIRE. It checks to see if that subject is on the list of subjects contained in the list called TOPICS it has been building. If not, it prints a polite message and asks for the next input.

However, if INQUIRY finds that it *does* have information about the subject, it looks up how many pieces of data have been given to it about that subject and then calls the Level 3 procedure SHOW.KNOWLEDGE. This procedure, in turn, loops through the property list belonging to the subject, displaying each value until all have been displayed.

**LOGIC.FINDER**

The meat of Micro-Logician is in this procedure and its associated Level 3 procedures, particularly concerning search techniques. This procedure group responds to questions that may require it to draw conclusions from information it has been given, but not explicitly.

For example, if we gave Micro-Logician the following two pieces of information:

**SOCRATES IS A MAN.**
**MAN IS INTELLIGENT.**

Micro-Logician should be able to handle the question:

**IS SOCRATES INTELLIGENT?**

even though it has not been specifically told that he is.

To accomplish this task, the LOGIC.FINDER procedure uses a series of searches aimed at backward-chaining to the solution to the question posed.

## What Is Backward-Chaining?

Backward-chaining searches are extremely important in AI research. Many expert systems use backward-chaining exclusively as a means of solving problems posed by their human designers and users.

Simply stated, "backward-chaining" embodies the idea of beginning with the *goal* and following a logical chain back to the proof or solution. This is opposed to "forward-chaining" techniques, which begin with the problem and attempt to move forward through a knowledge base to a solution. (See Figure 3-3.) Neither of these techniques is inherently superior to the other; each has its use and its application. Many systems combine the methods.

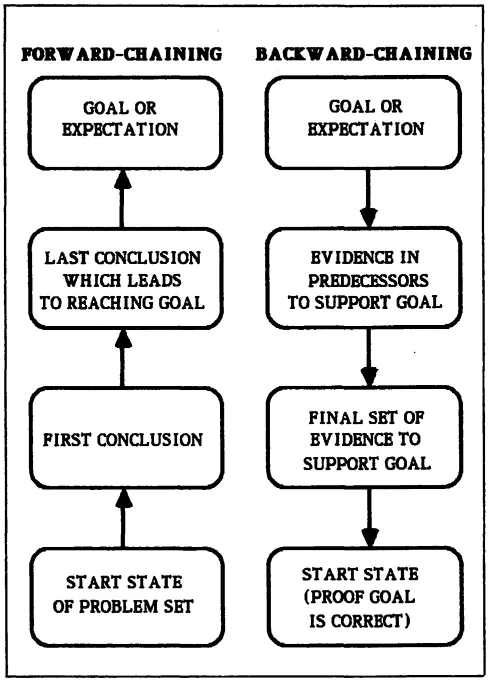

FORWARD-CHAINING
BACKWARD-CHAINING
GOAL OR EXPECTATION
LAST CONCLUSION WHICH LEADS TO REACHING GOAL
FIRST CONCLUSION
START STATE OF PROBLEM SET
GOAL OR EXPECTATION
EVIDENCE IN PREDECESSORS TO SUPPORT GOAL
FINAL SET OF EVIDENCE TO SUPPORT GOAL
START STATE (PROOF GOAL IS CORRECT)

**Figure 3-3. Backward- vs. forward-chaining in AI programs**

In a typical inquiry of Micro-Logician, the program begins with the goal—that is, the status about which the question has been posed—and attempts to work backwards to the point of linking the question to the answer.

An example may help to clarify this.

***Backward-Chaining to Socrates*** Staying with our Socratic example, assume that we have stored the following information in our very small knowledge base.

**SOCRATES IS A MAN.**
**SOCRATES IS A PHILOSOPHER.**
**A MAN IS TWO-LEGGED.**
**A PHILOSOPHER IS EXASPERATING.**

Suppose we wish to know whether Socrates is exasperating. Humans can, of course, quickly determine that to be the case by looking at the information provided. But the computer is not nearly as capable of drawing such inferences. (At least, not yet!)

To get the computer to draw inferences requires a program that will certainly meet our criterion for an artificially intelligent program: the program will do something which, if a human did it, would require intelligence.

Our program will first attempt to solve the question by a form of forward-chaining. It will look through its TOPICS list to see if Socrates is there and, finding it present, will examine the Socrates property list item-by-item to see if "exasperating" is present. For this approach to work, we would have had to tell the program that Socrates is exasperating. We know from the data listed above that we did not do that.

(Please don't conclude from this single example that forward-chaining is less elegant or intelligent than backward-chaining. The example is not at all exemplary of how such an approach might be used effectively and intelligently in a computer program.)

Having failed to find the information *explicitly* present, Micro-Logician moves to a backward-chaining approach to the problem. It starts at the trait "exasperating" and examines each property list in its memory to see if the term appears there. In this case, it finds the word "exasperating" in the property list associated with the word "philosopher."

It now returns to the Socrates property list to see if it has the trait "philosopher" in it. It does, so the program dutifully reports that Socrates is, in fact, exasperating. If the program failed either to find the word "exasperating" in its collection of traits or failed to find the associated term in the Socrates property list, it would inform us that it didn't have enough information to respond to the query.

We could, in theory, expand this backward-chaining search technique *ad infinitum* in our example. For instance, we could have had the sentence:

**AN INTELLECTUAL IS EXASPERATING.**

in place of the sentence about the philosopher and added a new sentence informing the system that:

**A PHILOSOPHER IS AN INTELLECTUAL.**

To achieve the correct response, Micro-Logician would take the following steps:

1. Forward-chain through the Socrates property list. Failing to find "exasperating," it would then take the next step.
2. Find "exasperating" in the property list labeled "intellectual."
3. Look in the Socrates property list for the word "intellectual."

4. Not finding the word "intellectual" in the Socrates property list, look for the word "intellectual" in other property lists.
5. Find the word "intellectual" in the property list labeled "philosopher."
6. Examine the Socrates property list and find the word "philosopher" there.
7. Report its conclusion that Socrates is exasperating.

You can see that the extent of the search—and the resources of memory and time required to carry it out—will expand greatly as each additional level of search is encountered. The technical term for these levels is "ply"; scientists speak of a two-ply or three-ply search. We have confined ourselves to the simplest approach.

**The LOGIC.FINDER Procedure Set**

Let's examine the LOGIC.FINDER procedure itself, now that we understand how the chaining process it implements actually works. We find that the procedure first separates the subject of the inquiry—in our example, "Socrates"—from the predicate, which is the question being asked. The procedure checks if the subject is in the list of TOPICS. If it doesn't find it there, it prints a message to that effect and goes back for another input. If it does find it, the program carries out the simplest check first, finding whether the trait is part of the property list of the subject. It reads the number of traits it has stored and examines each in turn to see if it matches. In our program, CHECK.SUBJECT handles this process.

Failing to find the predicate in the subject's property list, the program then calls on the CHECK.TOPIC.AREA procedure. This sets up a copy of the list. It uses a MEMBERP primitive to check if the predicate for which we are searching is a member of any property list associated with any of the topics.

When we ask about Socrates's two-leggedness, a match is found in the property list for "man". The program then makes the name of the new property list (in this case, "man") the new trait for a search through the subject's property list. In other words, having found "two-legged" as a trait in the property list associated with man, the program tries to find "man" in the property list of Socrates. If it finds it, it will have successfully backward-chained to the answer and will report that Socrates is indeed two-legged. Any other result will produce the answer "Not enough data" and a request for the next input.

**Special Logo Primitive Used**

This program uses only one nonstandard Logo primitive—PRINC, used in the INQUIRY procedure. This would be replaced by the more conventional TYPE in most versions of Logo. Other than that, the rest of the program should run as well in Microsoft Logo® or any other "standard" Logo available on the Macintosh®.

## Exploring AI with Micro-Logician

A number of improvements or enhancements may be made to our basic Micro-Logician program.

For example, consider storing the property lists created by the program in a disk file and then storing and retrieving them with the disk access primitives LOAD and SAVE. The entire process could even be menu-driven in ExperLogo®.

Another refinement would permit use of other verbs besides IS (ARE, for example). If you undertake this, consider whether you should be concerned about singular and plural nouns during output of information as well as during its entry.

Or add the word "about" to the list of words that could be extracted by the program. This would enable you to type

**INQUIRE ABOUT SOCRATES.**

which is a far more natural way of expressing the request than the somewhat arbitrary and rigid way we've designed.

Incidentally, consider making punctuation optional by checking the last character in the SENT1 variable to see if it is punctuation and removing it only if it is. Again, this would make the program a bit easier to use.

A more complicated refinement would permit multiple-ply backward-chaining. This would require considerable program modification, though basic procedures to accomplish the task are generally provided in the listing.

## Summary: What We've Learned about AI Programming

Search techniques are an important aspect of AI programming. In this chapter, we've looked at the following:

• The backward-chaining method of moving from a stated goal or objective to the cause or proof required;

• the forward-chaining approach, though not in an optimum implementation;

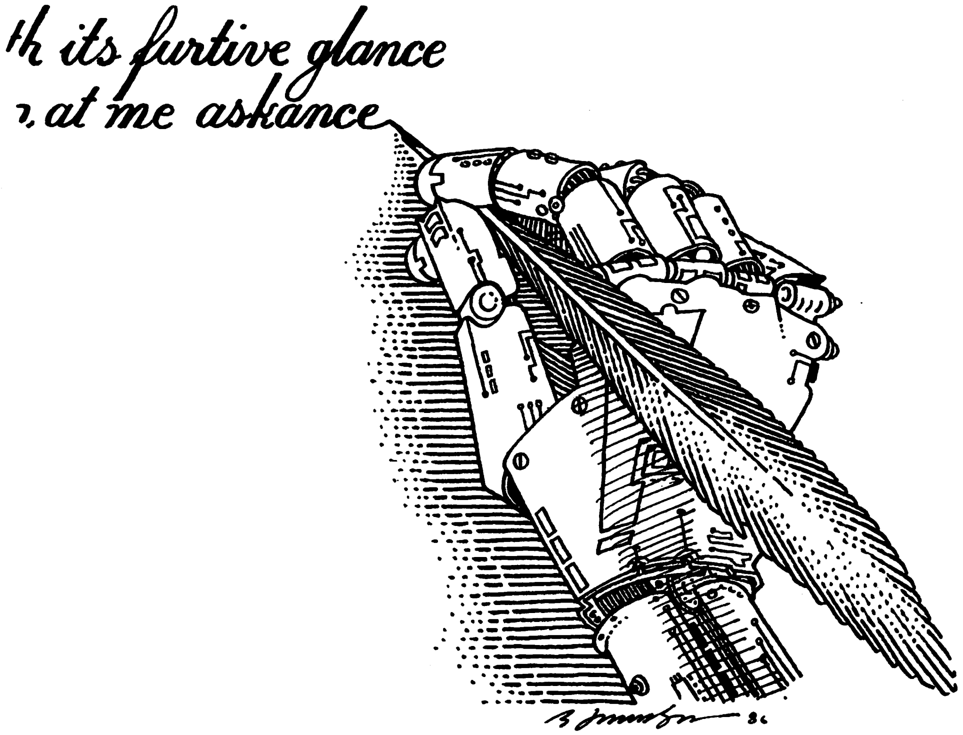
• natural language usage to see how relatively "normal" English can be accommodated as a program input when the right intelligence is built into the program itself.

```
                {Micro-Logician ©1985, The Waite Group}
                      {Logo program by Dan Shafer}

TO MICRO.LOGIC
  MAKE SENT2[]
  PRINT [Let's have a sentence. . .]
  MAKE SENT1 READLIST
   IF :SENT1 = [QUIT] [STOP]
  CLEAN.UP
  SCANNER
  MICRO.LOGIC
END

TO SET.UP
  MAKE ARTICLES [A AN THE]
  MAKE TOPICS []
  MICRO.LOGIC
END

TO CLEAN.UP
  MAKE SENT1 SENTENCE BUTLAST :SENT1 BUTLAST LAST :SENT1
  DELETE.ARTICLES
  MAKE SENT1 BUTLAST :SENT2
END

TO DELETE.ARTICLES
  MAKE WORD1 FIRST :SENT1
  IF EQUALP MEMBERP :WORD1 :ARTICLES NIL [MAKE SENT2 LPUT :WORD1
:SENT2]
  IF EMPTYP :SENT1 [STOP]
  MAKE :SENT1 BUTFIRST :SENT1
  DELETE.ARTICLES
END

TO SCANNER
  MAKE KEY1 FIRST :SENT1
  MAKE KEY2 FIRST BUTFIRST :SENT1
  IF :KEY2 = 'IS [ADD.A.FACT]
  IF :KEY1 = 'IS [LOGIC.FINDER]
  IF :KEY1 = 'INQUIRE [INQUIRY]
END

TO ADD.A.FACT
  SEPARATE.SENTENCE
  IF EQUALP GPROP :SUBJECT "NO_VALUES NIL
[NEW.SUBJECT]
  MAKE N GPROP :SUBJECT "NO_VALUES
  MAKE N :N + 1
  PPROP :SUBJECT "NO_VALUES :N
  PPROP :SUBJECT WORD "P :N :TRAIT
END

TO SEPARATE.SENTENCE
  MAKE SUBJECT FIRST :SENT1
  MAKE TRAIT BUTFIRST BUTFIRST :SENT1
END

TO NEW.SUBJCT
  MAKE TOPICS LPUT :SUBJECT :TOPICS
  PPROP :SUBJECT "NO_VALUES 0
END

TO INQUIRY
  MAKE SUBJECT FIRST BUTFIRST :SENT1
  IF EQUALP MEMBERP :SUBJECT :TOPICS NIL [PRINT<<I have no data on that
subject.>> STOP]
  MAKE N GPROP :SUBJECT "NO_VALUES
  PRINC :SUBJECT PRINT<< IS. . .>>
  SHOW.KNOWLEDGE
END

TO SHOW.KNOWLEDGE
  PRINT GPROP :SUBJECT WORD "P :N
  MAKE N :N-1
  IF :N≠0 [SHOW.KNOWLEDGE]
END

TO LOGIC.FINDER
  MAKE ANSWER [NOT ENOUGH DATA]
  MAKE SUBJECT FIRST BUTFIRST :SENT1
  MAKE TRAIT BUTFIRST BUTFIRST :SENT1
  IF EQUALP MEMBERP :SUBJECT :TOPICS NIL [<<I have no data on that
subject.>>STOP]
  MAKE N GPROP :SUBJECT "NO_VALUES
  CHECK.SUBJECT
  IF :ANSWER=[NOT ENOUGH DATA]
[CHECK.KNOWLEDGE]
  PRINT :ANSWER
END

TO CHECK.SUBJECT
  IF EQUALP GPROP :SUBJECT WORD "P :N :TRAIT [MAKE ANSWER [YES] STOP]
  MAKE N :N-1
  IF :N≠0 [CHECK.SUBJECT]
END

TO CHECK.KNOWLEDGE
  MAKE TEMPLIST COPYLIST :TOPICS
  CHECK.TOPIC.AREA
END

TO CHECK.TOPIC.AREA
  MAKE SEARCH. SUBJECT FIRST :TEMPLIST
  MAKE NEW.LIST PLIST :SEARCH.SUBJECT
  MAKE SEARCH.SUBJECT (LIST :SEARCH.SUBJECT)
  IF NOT EQUALP MEMBERP :TRAIT :NEW.LIST NIL
    [IF NOT EQUALP MEMBERP  :SEARCH.SUBJECT PLIST :SUBJECT NIL
[MAKE ANSWER [YES] [RETURN :ANSWER]]
  IF EMPTYP :TEMPLIST [STOP]
  MAKE TEMPLIST BUTFIRST :TEMPLIST
  CHECK.TOPIC.AREA
END
```

CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI

4

# The Digital Poet



- Text Generation Techniques and Uses
- The Connection to Natural Language Processing
- Haiku Poetry and the Computer: A Neat Connection
- Importance of Vocabulary as Text Generation Driver

The Digital Poet exemplifies a basic form of natural language processing programs known as "text generation" software. Before we can create computer programs that can comprehend human speech patterns, we need to analyze those speech patterns and find sensible, manageable rules. One efficient (and fun) way of doing that is to design programs that create stories, poems, and other human speech patterns.

This program is a variation on themes that have been used in college classrooms, AI laboratories, and homes for many years. I have written such programs in several computer languages on mainframes, minis, and micros. But as an amateur poet, I have never been satisfied with the programs' poetry. Nonetheless, this program illustrates the ideas and techniques involved in text-generation programs.

In some ways, poetry is the easiest form to use for generating text. Poetry can be short, and I have chosen here a form which is quite brief. Poetry may also be structured. These traits make poetry such as that produced by the program in this chapter easy to manage by a moderately sized computer program.

Designing a program that can "make up" entire stories is a far more complex task. A real story contains a beginning, middle, and end; a plot of sorts; and perhaps even characters. The process of generating story text is much more difficult than that of generating poetry.

In this chapter, then, we take our first tentative steps on the path of natural language processing. We have already examined some peripheral areas of interest in Chapter 3 as we examined the need to limit the type of input that would permit a program to "understand" and respond. Now we begin to take human writing patterns apart and see what goes into the computer creating meaningful written products.

## Two Types of Text Generation in AI

Investigation into the area of automatic text generation has been going on for some 20 years without agreement on the part of AI researchers on principles and approaches. In part, this is because text generation has arguably less value to a commercial product implementing AI principles than does natural language processing, which permits the *understanding* of the natural language by the computer.

Researchers and students of AI conducting experimental work in text generation have divided the field into two main types: random and meaningful. The line between the two types of text generation is not clear-cut. Randomly generated text can sometimes be meaningful, particularly in the area of poetry where so much of the meaning is subjective. Similarly, even the best-designed "meaningful" text generation programs produce occasional or frequent gibberish.

### Random Generation of Text

All text generation operates in the context of some rules of grammar and so is not truly random. However, when text is created randomly within the constraints of a grammar, it is described as "random text generation." We will examine this relatively straightforward type of text generation in the "Poetry Maker" program in this chapter.

Programs that generate text pseudo-randomly, like Poetry Maker, select the *words* randomly from a vocabulary available to the program, but the *sequence* of their use—and, generally, of their selection as well—is determined by rules, formats, or patterns supplied in the program. Poetry Maker has four predefined poetry formats. Random selection of nouns, verbs, adjectives, adverbs, prepositions, and articles takes place in conformance with these formats. Thus, generation of the text may *seem* random, but it is only somewhat so.

Randomly generated text, as you will see when you type in and run the Poetry Maker program in this chapter, can be perfectly correct from a grammatical perspective and yet nonsensical, even confusing, to people trying to understand what the computer has created. For example, one of the first poems Poetry Maker turned out after we had programmed it read:

**SEA TO A WATERFALL**
**A YELLOW RIVER NEAR A MOUNTAIN**
**RED EVENING**

Even for blank verse this is pretty difficult to understand! But it makes sense grammatically; it follows a pattern of parts of speech that will often produce meaningful sentences. The same *grammatical rules* that produced this nonsense are capable of producing this more comprehensible poetry:

**END OF A WATERFALL**
**A MUDDY RIVER AT THE MOUNTAIN**
**ETERNAL CYCLES**

We may not *appreciate* or even *understand* this bit of poetry but at least we know that waterfalls have ends, rivers get muddy and find themselves at mountains, and some cycles are eternal. That's clearly more than we can say about "sea to a waterfall"!

## Meaningful Text Generation

The other form of text generation used by AI researchers is far more complex. Programs designed to handle this meaningful text generation are generally large. They are not beyond the capability of the Mac but are beyond the scope of this book.

Generally speaking, the reason for creating meaningful text generators is to convert some internal representation of information into an appropriate string of words that may be understood by the user. In other words, it emphasizes the *meaning* rather than the syntactical *form* of natural language.

As it turns out, the computer can never be said to "understand" the information it contains. For example, it might have stored somewhere in its memory the information that someone named "Rex Sole" has a string of convictions for stealing information from computers. The information may be stored in a data base so that everything the computer "knows" is represented by fields of data (see Figure 4-1). The computer, if programmed to produce natural language responses to inquiries, might even be able to produce the result:

**REX SOLE STEALS INFORMATION FROM COMPUTERS.**

| Last_Name | First_Name | Age | Sex | Occupation Code |
|---|---|---|---|---|
| Jones | Alan | 39 | M | 27.023 |
| Meltz | Deborah | 26 | F | 19.113 |
| Sole | Rex | 29 | M | 09.119 |
| Wilson | Todd | 19 | M | 0 |

**Figure 4-1. Data stored about Rex Sole, computer thief**

But if Rex Sole comes to the system and types in his name, unless the computer is programmed to search its data base for such an individual, it will blithely let Mr. Sole extract whatever data he wants. The computer doesn't understand what "steals information from computers" means in the same sense that we do.

Nonetheless, a computer programmed to generate text in a way that is concerned with meaning and not just form will appear to be more intelligent than one which, like most modern systems, is capable only of telegraph English communication. ("Telegraph English" refers to sentences like "Rex Sole computer thief," which employ the fewest possible words to convey an idea, with no regard to the correctness of the sentence or to the aesthetic of the use of articles and verbs, adjectives and adverbs.) More important, we can learn a great deal from designing, using, and analyzing the output of such text generation programs.

***A Learning Example*** AI researchers learned early in their work in text generation that there are hundreds of possible sentence forms or structures in the English language. They also learned that the ambiguity of words is not necessarily a function of where they fall in a sentence. Dr. Roger C. Schank of Yale University's AI Laboratory provides the following example in his popular book *The Cognitive Computer*.

We start with a sample sentence that says:

**JOHN GAVE MARY A BOOK.**

Now we program the computer to understand that when we say "gave" we mean that, when the action depicted in the sentence is complete, the person named as having been given something is now in possession of it. The computer is now able to understand if we type in the sentence and then ask it:

**IS MARY IN POSSESSION OF A BOOK?**

that the answer is "yes."

But what will the computer do with this understanding of the word "gave" in the following sentences?

**JOHN GAVE MARY A HARD TIME.**
**JOHN GAVE MARY A NIGHT ON THE TOWN.**
**JOHN GAVE UP.**
**JOHN GAVE A PARTY.**
**JOHN GAVE HIS LIFE FOR FREEDOM.**

If we now ask the computer:

**IS MARY IN POSSESSION OF A HARD TIME?**

the answer will still be affirmative, even though you and I know that both the question and the answer are nonsense.

This excursion into the ambiguity of the word "gave" does not carry the solution to such ambiguity. We'll discuss this more in later chapters on NLP concepts, but an actual solution to the problem still awaits AI researchers. Perhaps you will find the answer! Our intent in the example is to show something that AI researchers learned about language by using text generation programs that produced sentences having logical understanding errors in them. By analyzing such sentences in terms of their grammar and content, researchers learned more about human language and how the computer would have to be programmed to deal with its intricacies.

## The Program

Poetry Maker has two main subprograms which do not interact directly with one another. The first, ADD.VOCABULARY, permits us to put new words into the vocabulary from which the program will create poetry. The second, MAKE.UP.POEM, generates a poem in one of four predefined formats. The two modules are loosely stuck together at the beginning of the program with a routine called POETRY.MAKER whose job is to ask the user which of the two main functions is desired to perform and then to call the appropriate subprogram.

Figure 4-2 is a box diagram of the Poetry Maker program. Referring to it may help you understand the following discussion about the program's contents.

### What the Program Does

Figure 4-3 shows the opening menu of the program. The menu is displayed and managed by the main routine, POETRY.MAKER.

The main program uses READCHAR to get the user's response to the menu request, so no [RETURN] key is needed. The user types in one of the three letters and the program follows instructions. The menu is repeated if a letter other than A, M, or Q is pressed.

***Adding Vocabulary: An Overview*** If users indicate that they want to add new words to the files of Poetry Words stored on the disk called "Logo Files," the program calls the ADD.VOCABULARY routine. This routine reads the existing file of words from the disk, if any are present, and informs the user that it is creating a new file if no previous file exists on the disk.

The rest of the ADD.VOCABULARY routines allow the user to enter new word(s) and identify their parts of speech and number of syllables. It adds these new words to the file in the appropriate property list for later retrieval and use by the MAKE.UP.POEM routines. If any articles or particles ("a," "an," and "the") are present, they are stripped from the input. The routine which handles this task is an old friend, the DELETE.ARTICLES routine found in Chapter 3 in the Micro Logician program.

When the ADD.VOCABULARY routines finish their task, they return control to the main POETRY.MAKER routine, where the menu is repeated.

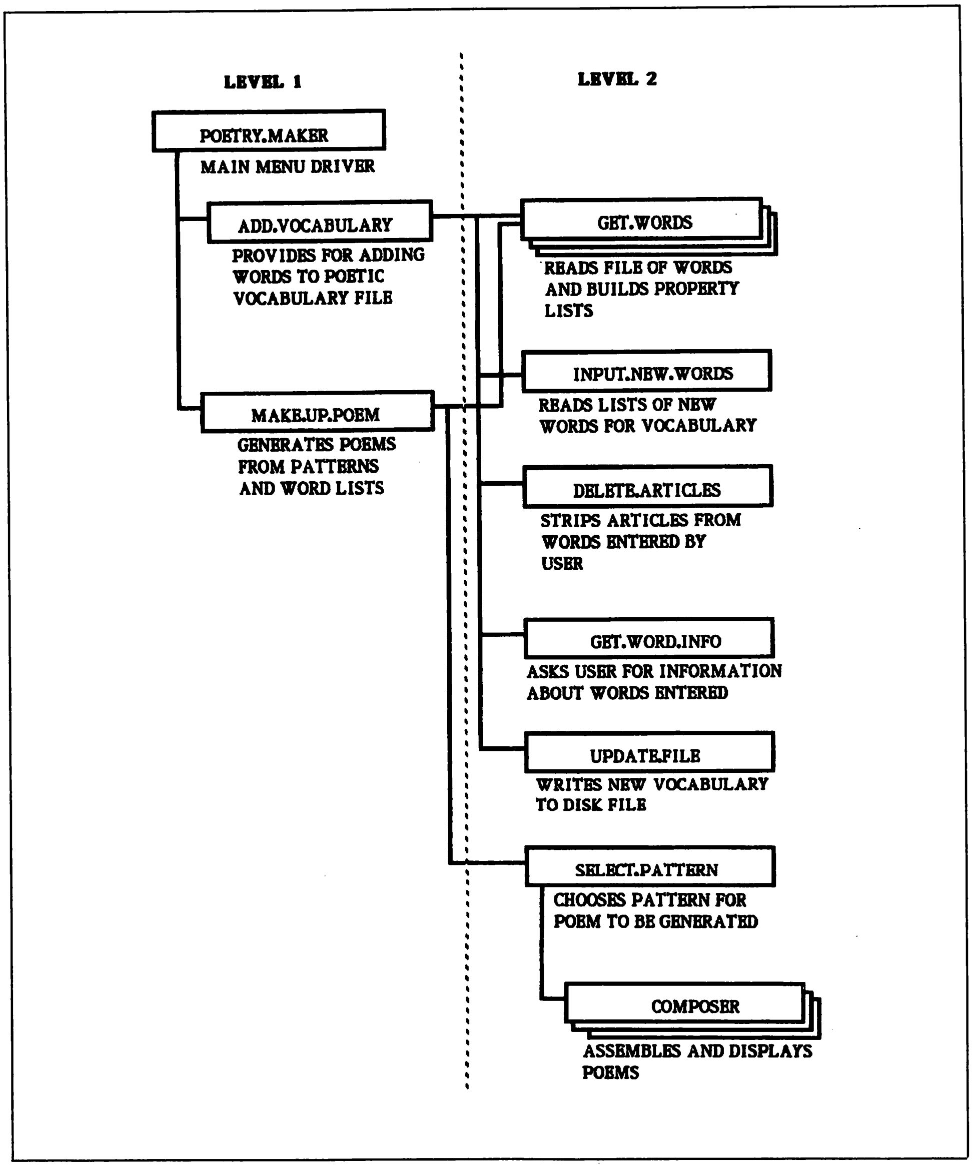
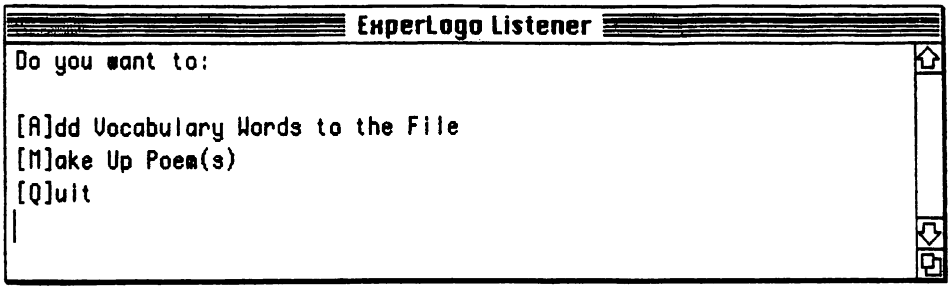

LEVEL 1
LEVEL 2
POETRY.MAKER
MAIN MENU DRIVER
ADD.VOCABULARY
PROVIDES FOR ADDING WORDS TO POETIC VOCABULARY FILE
MAKE.UP.POEM
GENERATES POEMS FROM PATTERNS AND WORD LISTS
GET.WORDS
READS FILE OF WORDS AND BUILDS PROPERTY LISTS
INPUT.NEW.WORDS
READS LISTS OF NEW WORDS FOR VOCABULARY
DELETE.ARTICLES
STRIPS ARTICLES FROM WORDS ENTERED BY USER
GET.WORD.INFO
ASKS USER FOR INFORMATION ABOUT WORDS ENTERED
UPDATE.FILE
WRITES NEW VOCABULARY TO DISK FILE
SELECT.PATTERN
CHOOSES PATTERN FOR POEM TO BE GENERATED
COMPOSER
ASSEMBLES AND DISPLAYS POEMS

**Figure 4-2. Partial box diagram of Poetry Maker**



```
Do you want to:

[A]dd Vocabulary Words to the File
[M]ake Up Poem(s)
[Q]uit
```

**Figure 4-3. Opening menu of Poetry Maker**

***Creating Poems: A Quick Look*** If, at the main menu, users indicate that they want the program to make up a new poem, the routines associated with the MAKE.UP.POEM subprogram take over. This set of routines asks users for the poem pattern they want the program to use, leaving the choice up to the program if desired. It then sets up the patterns of parts of speech needed to create the lines of poetry to match the chosen form and composes the poem.

Composing the poem consists of going through each line of the poem's format and generating words randomly from property lists for each part of speech. Each line is printed as it is composed. This process continues until a three-line poem has been composed and displayed. Control then returns to the main POETRY.MAKER menu.

## Sample Runs of the Program

Figure 4-4 shows a sample run of the POETRY.MAKER program's ADD.VOCABULARY routines. Note that users are asked to enter all the words to be added to the vocabulary, separating them with spaces. This method permits users to add entire poems to the vocabulary if they desire to do so. You could choose your favorite short poems and put them into the vocabulary (but punctuation must be omitted in the program's present form).

```
                         Poetry.Maker

Do you want to:
[A]dd Vocabulary Words to the File
[M]ake Up Poem(s)
[Q]uit
Please enter new words, separated by spaces.
You can even enter a whole new poem if you like!

When you're done, just enter a RETURN at the start of a line. Then I'll ask you about the
new words you've given me before I add them to the vocabulary file.
POETS WORK INSPIRATIONALLY

HOW MANY SYLLABLES DOES POETS HAVE?
WHAT PART OF SPEECH IS POETS?
N = NOUN V = VERB A = ADJECTIVE D = ADVERB P = PREPOSITION
HOW MANY SYLLABLES DOES WORK HAVE?
WHAT PART OF SPEECH IS WORK?
N = NOUN V = VERB A = ADJECTIVE D = ADVERB P = PREPOSITION
HOW MANY SYLLABLES DOES INSPIRATIONALLY HAVE?
WHAT PART OF SPEECH IS INSPIRATIONALLY?
N = NOUN V = VERB A = ADJECTIVE D = ADVERB P = PREPOSITION
Do you want to:

[A]dd Vocabulary Words to the File
[M]ake Up Poem(s)
[Q]uit
```

**Figure 4-4. Sample run of ADD.VOCABULARY routines**

After the words have been entered, the program goes through the word list and asks about each word (except articles, which it skips), the part of speech it is, and the number of syllables it contains. (The program as presented here doesn't use this information about number of syllables, but we ask for it and store it to enable an interesting modification discussed later.)

The program writes the new vocabulary on the "Poetry Words" file on the "Logo Files" disk and returns to the main menu.

Figure 4-5 shows a sample run of the MAKE.UP.POEM routines. There is, as you can see, very little for users to do. After choosing the "M" option from the main menu, they need only type a single-digit number in response to the computer's request for the format of the poem to be used. Then, in a few seconds, a poem appears on the screen and users are asked if they'd like to see another.

```
Do you want to:

[A]dd Vocabulary Words to the File
[M]ake Up Poem(s)
[Q]uit
I know four Haiku poetry patterns.

Enter the number of the pattern you want me to use (1-4) or use any other number to
tell me to pick one at random.

SOARING EARLY AFTERNOON
THE POOR SHIMMERS UNDER GREEN PEOPLE
LITTLE QUIET GRASS
Do you want me to compose another poem?
I know four Haiku poetry patterns.

Enter the number of the pattern you want me to use (1-4) or use any other number to
tell me to pick one at random.

AN DARK HAZY RIVER
AT THE RED LEAVES
THE LEAVES DIES
Do you want me to compose another poem?
nil
```

**Figure 4-5. Sample Run of MAKE.UP.POEM Routines**

## A Few Words about Poetry

Before analyzing the program listing to see how Poetry Maker works, it will be useful to discuss poetry in general and the Haiku form specifically. This will enable you to understand better why the poems produced by Poetry Maker *sound* the way they do.

The poems created by Poetry Maker don't rhyme. At least they don't *automatically* rhyme. They are of the form called "blank verse" and, though they don't often rhyme, they *are* poems.

Not all poetry is structured in a definable way, but much poetry *is* structured in terms of rhyme and *meter*. Meter refers to the way syllables are emphasized and the kind of "singing" effect that results from particular patterns of such syllables. For example, the famous Joyce Kilmer poem, "Trees," uses the simplest meter—emphasizing alternate syllables. The dark syllables are emphasized:

I **think** that **I** shall **never see**
a **poem** as **love**ly as a **tree.**

Other forms of meter emphasize varying patterns of syllables; and some meter patterns can be quite complex and sophisticated.

**What's Haiku Poetry?**

While the poems created by this program are not structured as to their meter, they are, never the less, highly structured compositions. They are English variations on the Japanese theme of poetry known as Haiku. They are not true Haiku because they do not pay attention to syllable count, but they are attempts at emulating the Haiku form.

Haiku is an ancient form of verse-making that lends itself better to Japanese than to English. In Japanese, Haiku has two important characteristics. Most important, it attempts to distill deep and eternal truths and ideas into a very small number of words. Each composition is supposed to contain power and depth. Some Haiku poems, even in Japanese, fail to do this, but that doesn't change the fact that the poet's intent was to focus a great idea into a small poem. Second, each Haiku poem has the same kind of syllable structure, with five syllables in the first line, seven in the second, and five in the third.

Because of their brevity, intense imagery is essential in Haiku. Most of the poems focus on events and images in nature; it is with this in mind that we chose the vocabulary for the program. You may wish your Poetry Maker program to produce poems with themes of sports, or love and romance, or religion. In that case, you choose appropriate nouns, verbs, adjectives, and adverbs—prepositions are somewhat limited and of more general use—and watch as your program cranks out masterpiece after masterpiece. Well, poem after poem.

People who have the time to pursue such studies have determined that the vast majority of published Haiku poetry can be seen as falling into one of four different *patterns* or formats. We have chosen these four formats for Poetry Maker. You can alter them in almost any way you wish, as I will explain when we discuss the routines that contain representations of the formats. The formats themselves are described as sequences and combinations of articles, nouns, verbs, adjectives, adverbs, and prepositions.

Do you need a review of the parts of speech? If not, skip to "How the Program Works."

- *Articles* point to the thing about which we are talking. *Particles* allow the thing to remain indefinite. The common article is "the." Particles are "a," and "an." *The* book is different from a book. *An* alligator is far different from *the* alligator which just bit the end off your stick!

- *Nouns* are names of people, places, things, or ideas. Nouns include "book" and "alligator" from the two previous examples.
- *Verbs* are of two types: action and state-of-being. The former express what a particular noun is doing or having done to it. In the sentence, "The alligator just bit the end off my stick!" the word "bit" is an action verb. Verbs that describe the state of being of something tend to be variations on the verb "to be." In the sentence, "The alligator is very large," the verb "is" indicates the state of being or condition of the noun "alligator."
- *Adjectives* describe or tell something about nouns or pronouns. The word "large" in the preceding example is an adjective that describes, or *modifies*, the noun "alligator."
- *Adverbs* describe (modify) verbs, adjectives, or other adverbs. In the sentence, "The alligator *quickly* bit the end off the stick," the word "quickly" is an adverb which describes *how* the alligator bit the stick.
- *Prepositions* are connecting words such as "to," "toward," "at," and "in," which indicate relationships between objects. They often indicate place or position: The book is *in* the car. The alligator came *toward* the dog.

## How the Program Works

We have already seen what the program does. We have gained an appreciation for the program's overall operation. Now let's look at the Logo procedures that carry out the functions. The three major routines in the program are POETRY.MAKER, ADD.VOCABULARY, and MAKE.UP.POEM. We will examine each in turn, describing subprocedures as we encounter them.

### POETRY.MAKER Procedure

The POETRY.MAKER procedure is a straightforward menu handler.

Figure 4-2 provides a partial box diagram of the Poetry Maker program. It shows the major functional blocks that make up the program. Figure 4-6 provides a more detailed box diagram of ADD.VOCABULARY and its associated routines, showing their relationships and briefly defining their purposes.

The first statement, CLEARTEXT, erases the contents of the Listener Window, which is the only text window permitted in ExperLogo®. In Microsoft Logo®, the command CT will have the same effect on the current text window.

A series of PRINT statements follows. A peculiarity in these statements is the use of guillemets (<< and >>) instead of the more traditional quotation marks. These marks are made by holding down the [OPTION] and [bs] keys simultaneously, along with the [SHIFT] key for the closing guillemets. In Microsoft Logo®, ordinary double quotation marks will work nicely.

After using READCHAR to get the user's response, the next three lines call the appropriate procedures or the primitive STOP, as appropriate. If the user's input is not A, M, or Q, the program calls POETRY.MAKER again and redisplays the menu.

### ADD.VOCABULARY and Related Procedures

When users select A, indicating they wish to add to the file, the POETRY.MAKER routine calls the ADD.VOCABULARY procedure.

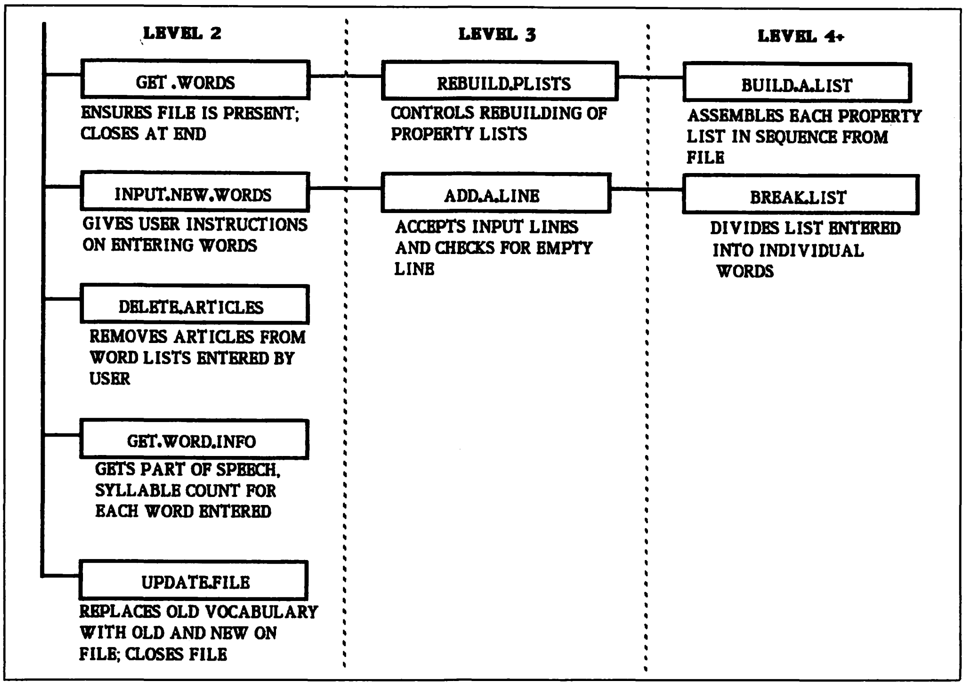

LEVEL 2
LEVEL 3
LEVEL 4+
GET.WORDS
ENSURES FILE IS PRESENT; CLOSES AT END
REBUILD.PLISTS
CONTROLS REBUILDING OF PROPERTY LISTS
BUILD.A.LIST
ASSEMBLES EACH PROPERTY LIST IN SEQUENCE FROM FILE
INPUT.NEW.WORDS
GIVES USER INSTRUCTIONS ON ENTERING WORDS
ADD.A.LINE
ACCEPTS INPUT LINES AND CHECKS FOR EMPTY LINE
BREAK.LIST
DIVIDES LIST ENTERED INTO INDIVIDUAL WORDS
DELETE.ARTICLES
REMOVES ARTICLES FROM WORD LISTS ENTERED BY USER
GET.WORD.INFO
GETS PART OF SPEECH, SYLLABLE COUNT FOR EACH WORD ENTERED
UPDATE.FILE
REPLACES OLD VOCABULARY WITH OLD AND NEW ON FILE; CLOSES FILE

**Figure 4-6. Detailed box diagram of ADD.VOCABULARY routines**

The ADD.VOCABULARY routine provides our first example of how a computer might "learn" something new—in this case, new words for use in poetry. I'll have more to say about the subject later in the book, but you should know that the learning involved here is called "explicit instruction" learning. The computer is given information and retains it in some form for later use. The process parallels rote memorization for humans. The computer learns new vocabulary words in the same way most of us "learned" our multiplication tables—by committing them to memory. Its "memory" is a disk file.

The procedure first calls the GET.WORDS procedure (discussed in the next section in greater detail), which reads in the existing file of poetry words and sets up the property lists for adding words to them. ADD.VOCABULARY then initializes three variables that will be needed by the procedures that follow. These variables must not be reinitialized during vocabulary input.

The main work of the procedure is then handled by the next four statements. INPUT.NEW.WORDS gets new words from the user. DELETE.ARTICLES is the same procedure of the same name in the Micro-Logician program in Chapter 3. The next statement converts the output from the DELETE.ARTICLES routine back to the WORD.LIST with which the program works. GET.WORD.INFO goes through WORD.LIST and asks the user to enter the part of speech and number of syllables for each word and adds this information to the property lists. Finally, UPDATE.FILE writes the property lists to the file, closes the file, and returns control to the POETRY.MAKER menu.

***GET.WORDS*** This procedure attempts to open a poetry words file on the Logo Files disk. If the result of this is "nil," there is no such file. In that case, the procedure prints a message to that effect and STOPs, returning control to the ADD.VOCABULARY procedure.

**Note**

If this is the first time you are creating a file of words, be certain to put at least one word of each type—noun, verb, adjective, adverb, and preposition—into the file. Failure to do so may result in the file becoming garbled later when you attempt to add more words. The poetry produced will contain "nil," perhaps with some frequency.

Assuming GET.WORDS finds the file it expected to find, it then sets up a list called WORDTYPES to contain the letters "N" (noun), "V" (verb), "A" (adjective), "D" (adverb—we couldn't use "A" again, so we punted!), and "P" (preposition). Then it calls the REBUILD.PLISTS procedure.

(Note that if you are using Microsoft Logo® rather than ExperLogo®, the disk file handling routine must be substantially rewritten. See Appendix B for details.)

***REBUILD.PLISTS*** Information about words in the vocabulary is contained in five property lists, each of which is named after one part of speech. After storing information in a disk file, retrieve it and print out the property list showing, for example, all nouns in the file by typing:

```
PRINT PLIST 'N
```

Substitute the other letters—V, A, D, or P—for the N in order to examine the property lists for the other parts of speech.

The REBUILD.PLISTS procedure reassembles the information stored in each of the five lists into a PLIST—or property list. It must do this since Logo stores the information in each property list on the file as a simple list and doesn't remember it as a property list. REBUILD.PLISTS is designed to be used in all parts of the program.

The procedure reads the first list from the file and puts this list into the variable named TEMP1. Then it calls the BUILD.A.LIST procedure, which assembles this information into the PLIST for nouns.

When BUILD.A.LIST has completed its task, the REBUILD.PLISTS procedure eliminates the first element in the list of WORDTYPES using the BF (ButFirst) primitive. It then checks to see if the end of the word file has been reached and, if so, STOPs processing. Otherwise, it recalls itself and the WORDTYPES list now contains the label for the next property list to be read. The first statement reads the subsequent list in the file and processing continues until the end of the file has been reached.

***BUILD.A.LIST*** This procedure (see Figure 4-7) constructs the property list for each of the five parts of speech using the data in the variable TEMP1 as read by the REBUILD.PLISTS procedure.

This line takes the first element of the list WORDTYPES and gives the property list that name. Then it appends the first and second elements of the TEMP1 list to the property list for this part of speech. The next line removes the two elements of the TEMP1 list that have just been appended to the property list and

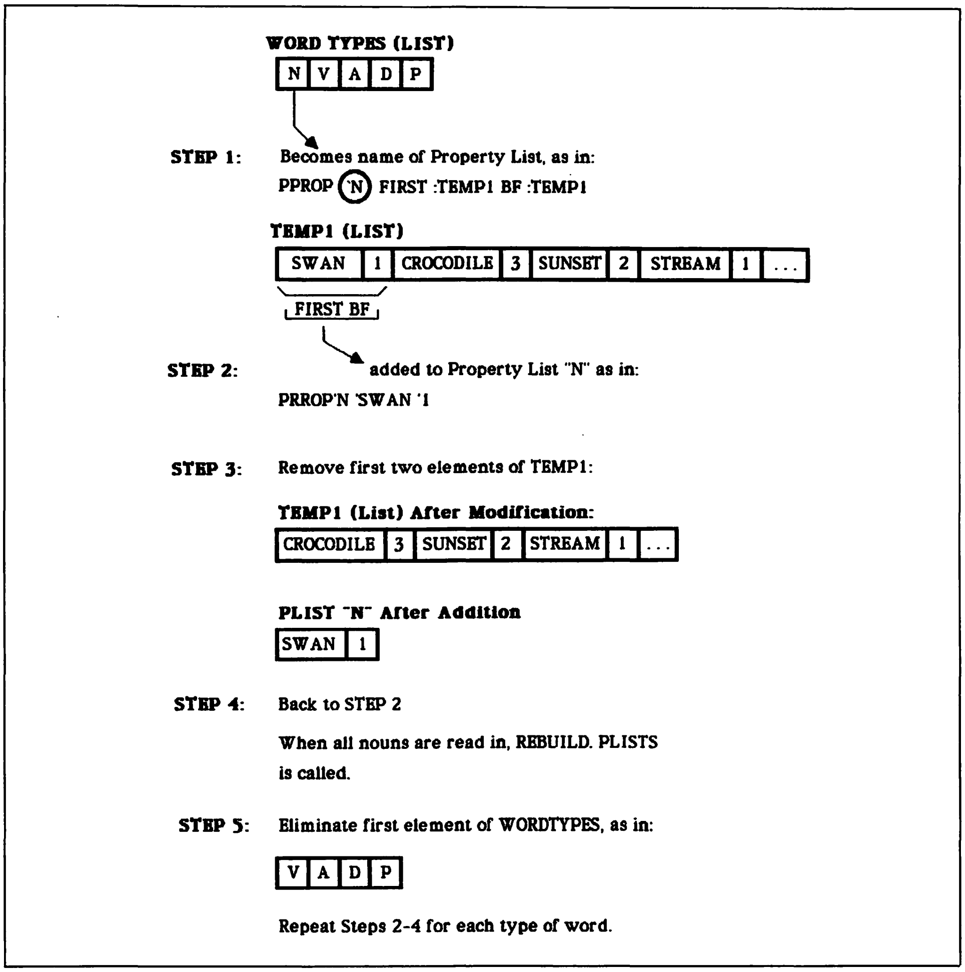

WORD TYPES (LIST)
N V A D P
STEP 1:
Becomes name of Property List, as in:
PPROP 'N FIRST :TEMP1 BF :TEMP1
TEMP1 (LIST)
SWAN 1 CROCODILE 3 SUNSET 2 STREAM 1 ...
FIRST BF
STEP 2:
added to Property List "N" as in:
PRROP'N 'SWAN '1
STEP 3:
Remove first two elements of TEMP1:
TEMP1 (List) After Modification:
CROCODILE 3 SUNSET 2 STREAM 1 ...
PLIST "N" After Addition
SWAN 1
STEP 4:
Back to STEP 2
When all nouns are read in, REBUILD. PLISTS is called.
STEP 5:
Eliminate first element of WORDTYPES, as in:
V A D P
Repeat Steps 2-4 for each type of word.

**Figure 4-7. Constructing property lists in BUILD.A.LIST**

the following line checks to see if the end of that list has been reached. If so, it STOPs processing. Otherwise, BUILD.A.LIST recalls itself with the list now two elements shorter than it was on the last pass through the procedure.

***INPUT.NEW.WORDS*** This procedure is simply a set of instructions that tell the user what is going to happen and then calls the ADD.A.LINE subprocedure.

***ADD.A.LINE*** This procedure reads a line of input from the user, puts the input into the variable LINE, checks to see if it is an empty list (i.e., that the user has finished and has simply pressed the [RETURN] key), and then calls the BREAK.LIST procedure.

***BREAK.LIST*** The BREAK.LIST procedure breaks each line of input into its component words and puts each into the variable WORD.LIST. This is necessary so that we don't end up with WORD.LIST as a list of lists rather than a list of words. This avoids some strange and undesirable consequences. For example, if the user entered the following two lines of words to add to the vocabulary:

```
SLIMY SLITHERING SNAKES
SOAKED SUDSY STUFF
```

[RETURN] ← to indicate the input is ended. If we hadn't set up the BREAK.LIST procedure correctly, the variable WORD.LIST would eventually contain two lists:

```
[[SLIMY SLITHERING SNAKES][SOAKED SUDSY STUFF]]
```

The problem with this is that when the program asks the user to enter information about each "word," it would be asking the number of syllables and which part of speech the "word" SLIMY SLITHERING SNAKES is. Further, the first element of a list of lists would be a list, not a word. Thus,

```
MAKE WORD1 FIRST :WORDLIST
```

would result in WORD1 containing the list [SLIMY SLITHERING SNAKES] rather than the word, SLIMY, that we obviously wanted. Since BREAK.LIST essentially goes through the list entered one element at a time, adding each element to the list and redefining the input to be reduced by the element with which it has just dealt, we build one long list instead of a list of lists.

After removing the word just placed into the variable WORD.LIST, the BREAK.LIST procedure calls itself again. When the list LINE is finally empty (as shown by the result of "T," for "true" when the EMPTYP test is run on it), the procedure STOPs execution and returns control to the ADD.A.LINE procedure.

***DELETE.ARTICLES*** After all of the user's words have been assembled into the variable called WORD.LIST, the procedure DELETE.ARTICLES goes through the list and removes all occurrences of "a," "an," and "the." This eliminates the necessity of asking for their number of syllables and parts of speech over and over when actual poetry is used to update the vocabulary. Since this procedure is identical to the one we studied in Chapter 3, it is not described in detail here.

Note that DELETE.ARTICLES routine leaves the original WORD.LIST in a temporary holding variable called SENT2 (for SENTence 2). The ADD.VOCABULARY procedure removes the last element of this list and puts the rest into the variable WORD.LIST again. The last element of the list always is a "nil" because we initialized SENT2 to start out as the empty list which in ExperLogo® is represented as "nil."

***GET.WORD.INFO*** GET.WORD.INFO is a straightforward procedure. It goes through the contents of WORD.LIST, which contains all words entered by the user with articles removed. It asks about each word—how many syllables it has and what part of speech it is. Each input uses READCHAR so the user never has to press the [RETURN] key. This means a word cannot contain more than nine syllables, a limitation we won't find troublesome!

Information about each word is placed into the appropriate property list and removed from the WORD.LIST. When the list of words entered by the user is exhausted, the program returns control to the ADD.VOCABULARY procedure for the final step in the processing.

***UPDATE.FILE*** To update the file of vocabulary words, we set up the file for writing and then use one of the output functions to print each property list on the file. Normally we store data on a disk file with the PRIN1 function in ExperLogo® because that function preserves the data type and special punctuation (brackets, etc.) associated with the data as it is written to the file. But handling a property list that way has no real advantage. Since the PRINT function is more common in Logo, we chose to use that approach here.

At the end of this processing, the file is closed and control passes through ADD.VOCABULARY back to POETRY.MAKER.

## MAKE.UP.POEM and Related Procedures

Figure 4-8 is a detailed box diagram of the MAKE.UP.POEM procedures. The diagram will help you follow the discussion about how creating a new poem is handled by Poetry Maker.

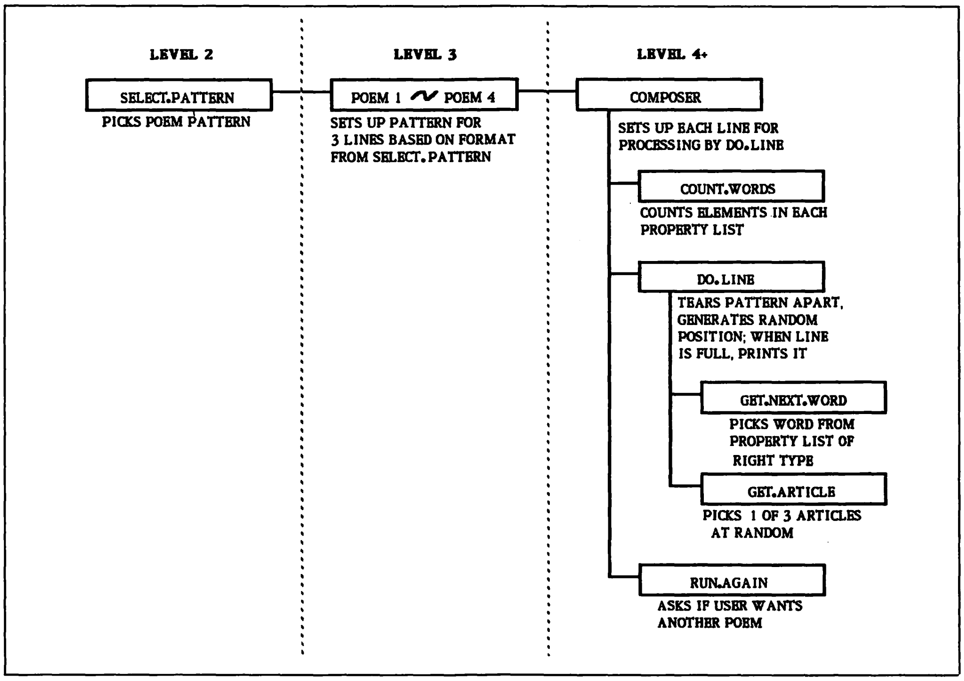

LEVEL 2
LEVEL 3
LEVEL 4+
SELECT.PATTERN
PICKS POEM PATTERN
POEM 1 ~ POEM 4
SETS UP PATTERN FOR 3 LINES BASED ON FORMAT FROM SELECT.PATTERN
COMPOSER
SETS UP EACH LINE FOR PROCESSING BY DO.LINE
COUNT.WORDS
COUNTS ELEMENTS IN EACH PROPERTY LIST
DO.LINE
TEARS PATTERN APART, GENERATES RANDOM POSITION; WHEN LINE IS FULL, PRINTS IT
GET.NEXT.WORD
PICKS WORD FROM PROPERTY LIST OF RIGHT TYPE
GET.ARTICLE
PICKS 1 OF 3 ARTICLES AT RANDOM
RUN.AGAIN
ASKS IF USER WANTS ANOTHER POEM

**Figure 4-8. Detailed box diagram of MAKE.UP.POEM routines**

Compare Figure 4-8 with Figure 4-6. The structure of this set of routines differs markedly from that of the ADD.VOCABULARY procedures. Here, the main procedure, MAKE.UP.POEM, is little more than a driver routine that gives the user access to the rest of the procedures. The procedures are arranged less hierarchically and more sequentially because the processing in MAKE.UP.POEM is less repetitive and more serial in nature.

MAKE.UP.POEM uses the same GET.WORDS procedure we discussed in ADD.VOCABULARY procedures. One difference, though, is that if you try to run MAKE.UP.POEM without a file, the program informs you that, "I can't compose poetry with no vocabulary!" It STOPs processing and returns control to the POETRY.MAKER menu handler. Figure 4-9 shows the vocabulary used by POETRY.MAKER.

| | |
|---|---|
| N | POOR 1 PEOPLE 2 LEAVES 1 MOUNTAIN 2 GRASS 1 BUTTERFLY 3 SHADOW 2 AFTERNOON 3 PUDDLE 2 SUNSET 2 EVENING 2 MORNING 2 WAVES 1 SEA 1 RIVER 2 WATERFALL 3 LAKE 1 |
| V | GIVE 1 ROLLS 1 STATTERS 2 COURSES 2 FLIES 1 ROARS 1 BABBLES 2 WHISPERS 2 SLEEPS 1 STRUGGLES 2 ENDS 1 RISES 2 FLOATS 1 DIES 1 SHIMMERS 2 |
| A | LITTLE 2 COMFORTABLE 4 CRAWLING 2 SOARING 2 BLAZING 2 HOT 1 COOL 1 DARK 1 YELLOW 2 GREEN 1 RED 1 EARLY 2 CRASHING 2 RUSHING 2 CLEAR 1 BLUE 1 HAZY 2 SOFT 1 QUIET 2 |
| D | VERY 2 OFTEN 2 SLOWLY 2 LIGHTLY 2 GENTLY 2 SWIFTLY 2 SOMBERLY 3 MOURNFULLY 3 ENDLESSLY 3 |
| P | TOWARD 2 INTO 2 FOR 1 AT 1 TO 1 IN 1 UNDER 2 OVER 2 NEAR 1 |

**Figure 4-9. Vocabulary chosen for Poetry Maker**

***SELECT.PATTERN*** This procedure provides instructions on selecting the type of poem to be created and then obtains the user's selection. The line:

```
IF NOT NUMBERP :POEM.TYPE [SELECT.PATTERN]
```

checks to be sure the POEM.TYPE entered by the user is a number. If not, the routine calls itself again, reprompting the user for a number.

After ensuring the input is numeric, the program calls SET.POEM.

***SET.POEM*** This calls one of four procedures—named, appropriately enough, POEM1, POEM2, POEM3, and POEM4—depending on whether the user entered a 1, 2, 3, or 4.

But the instructions produced by SELECT.PATTERN give users the option of permitting the program to create one of the four formats at random. The final line of the SET.POEM procedure says, "If the user didn't enter any of the expected values, 1–4, then generate a random number between 1 and 4 and run the procedure with that type as the input."

(As with many microcomputer random number generators, the RANDOM instruction in both ExperLogo® and Microsoft Logo® produces a number between 0 and one less than the number provided. Therefore, we added 1 to the value produced to ensure that we didn't end up with a value of 0. Getting a 0 would not cause the program to malfunction, but it would slow down execution while the program repeated the selection until it generated a random number that was between 1 and 4. Also, failure to use the form shown would result in format 4 never being selected except by explicit instructions from the user.)

***POEM1, POEM2, POEM3, and POEM4*** The four procedures create patterns for each of three lines of poetry. Each calls the COMPOSER procedure when it has established the patterns.

Pattern-matching is an important AI concept that we will consider when we examine programs dealing with natural language processing. Here, there is no real *matching* of patterns going on; instead, the program simply generates patterns for the computer to fill with words.

Let's look at POEM1 to see how each pattern is constructed; you can look at the other three formats for yourself.

Each of the first three lines of the procedure creates a list associated with a pattern. Each list consists of a number of letters. You will recognize all of them except one as identical to those used in the variable WORDTYPES in the ADD.VOCABULARY routine. The one new letter added here is "R," which is used to mean "article."

When POEM1 completes its run, PATTERN1 will be defined as a line of poetry consisting of an adjective, followed by an adjective, followed by a noun. A line like:

**COOL GREEN SEA**

would fill this pattern. The second line would consist of an article or a particle, a noun, a verb, a preposition, an adjective, and a noun. This is one of the most common forms of the English *declarative sentence* (a sentence that makes a statement). Look at this sentence for the pattern just described:

The boy listened to loud noises.

In poems of Type 1, then, the second line will often sound like a "normal" sentence conveying some information (though the information may be unusual given the vocabulary we are using).

The third line of this type of poem has two adjectives followed by a noun, just like the first line. This brings up a good point about Haiku which we saved until now so you could see how it related to the patterns we define.

The "point" of a Haiku poem is often conveyed in the last line. The absence of a verb in its format—as in this pattern and two of the others as well—evokes imagination, but not action, from the poet's perspective. Thus we might have a poem which ends:

**QUIET DARK MORNING**

Such a line doesn't tell us anything about the quiet, dark morning other than that it *is* or *was*. The poet leaves the image for our imagination to deal with, a technique the great human poets use to great effect.

***COMPOSER*** COMPOSER is the "heart" of the MAKE.UP.POEM procedure group. It first initializes the variable LINE to be an empty list. It then counts the number of elements in each of the property lists (we'll soon see why we need this information) and it defines the number of articles in the variable NO._R to be 3.

Then it places the three patterns (PATTERN1, PATTERN2, and PATTERN3) consecutively into a variable called PAT and passes this variable to the procedure called DO.LINE. When it has done this three times, it calls the RUN.AGAIN procedure, which asks users if they want to see another poem created.

***COUNT.WORDS*** This procedure uses the Logo primitive COUNT to determine the number of elements in each property list. Keep in mind that each property list contains the word and the number of syllables as two units, so the result of COUNT will always be an even number. Later, we have to adjust for that fact.

***DO.LINE*** Taking each pattern as it receives it an element at a time, this procedure looks at the type of word (part of speech) in each element of the pattern. It then generates a random number between 1 and the number of elements that particular property list contains (as determined by COUNT.WORDS). Control is then passed to the GET.NEXT.WORD procedure.

After each word is handled this way, the DO.LINE procedure removes it from the PAT variable with the command:

```
MAKE PAT BF :PAT
```

where "BF" is a shorthand way of programming the BUTFIRST primitive.

It then checks to see if the pattern is empty. If so, it prints the line of poetry it has been composing and then STOPs. If there are more word types to find in the pattern, it calls itself with PAT reduced by the word just located.

The exception to the use of GET.NEXT.WORD is when the word to be found is an article or a particle (type "R"). In that case, the DO.LINE procedure calls GET.ARTICLE.

***GET.NEXT.WORD and GET.ARTICLE*** These two procedures do essentially the same thing, though GET.NEXT.WORD is more complex for obvious reasons.

GET.NEXT.WORD first ensures that the random number generated by DO.LINE is *odd*. This is because the information is stored in each property list so that the word is followed by its number of syllables, followed by the next word and its number of syllables, and so forth. The words themselves, therefore, occupy the first, third, fifth, seventh, and remaining odd positions. To find a word, then, we select an odd-numbered position in the list. The procedure ensures this by checking to determine what remainder we have when dividing by 2 the number generated by the DO.LINE procedure. If the answer is 0 (i.e., EQUALP. . .0), then we subtract 1 from the answer. Note that this can never leave us with a zero, since the number had to be even before we subtracted.

GET.NEXT.WORD then defines the variable NEXT.WORD to be the :LOCth element of the appropriate property list. This line is worth a brief examination. It reads as follows:

```
MAKE NEXT.WORD ELEMS :LOC 1 PLIST :WORD1.TYPE
```

The primitive ELEMS is unique to ExperLogo®. (Appendix B describes how to carry out the same function in Microsoft Logo®.) ELEMS takes three arguments. The first, in this case :LOC, is the beginning position in the list from which extraction is to begin. The second, in this case 1, tells the program how many elements to extract beginning at that location. The final argument, in this case PLIST :WORD1.TYPE, is the name of a list from which the information is to be extracted. This is a very efficient way of finding a specific element in a list. ExperLogo® simply goes to the property list for the type of word involved, moves to the position indicated by the variable :LOC and takes that one element as the value of NEXT.WORD.

After picking up NEXT.WORD from the property list, it puts this word into the variable LINE. Note that it takes the first element of it before putting it into LINE. This is because ELEMS returns a list and we want the variable LINE to contain *words* not single-word *lists*.

The same procedure is followed in GET.NEXT.ARTICLE except we don't have to check for odd numbers—there are only three values and no syllable lengths stored in our arbitrarily defined list of articles.

***RUN.AGAIN*** When COMPOSER has run once for each line of a poem's pattern, it calls the RUN.AGAIN procedure, which asks if the user wishes to see another poem. If the answer is "Y," the program reruns SELECT.PATTERN; otherwise, the program will STOP and return to the POETRY.MAKER menu. (It calls SELECT.PATTERN rather than MAKE.UP.POEM to avoid rereading the data file and unnecessarily reinitializing variables.)

## Exploring AI with Poetry Maker

Poetry Maker provides many opportunities for enhancements, refinements, and explorations. For the sake of organization and discussion, we'll examine these possible changes in three categories: improving the program's poetic product, making the program more flexible, and adding greater intelligence.

### Better Poetry Through Programming

There are at least four interesting changes to Poetry Maker that would improve the quality of the poetry it produces. Your love of poetry and your imagination can provide dozens of ideas for such modifications.

***Handling Punctuation Correctly*** Poetry Maker will not deal with punctuation. If we enter a line of poetry like this:

**UNDER A SOARING RIVER,**
**A MORNING DIES.**

the computer will pick up the punctuation as part of the words and define RIVER, and DIES. as two new words. When words are being added to the vocabulary, we must strip them of punctuation just as we strip them of articles As we did with the DELETE.ARTICLE procedure, we now define a variable list of the punctuation marks, scan for them, and eliminate them.

Punctuating poems as they are displayed is trickier, but not too difficult with the four patterns given in the program. We can, for example, add a dash (a double-hyphen) to the end of the first line of a poem of the first type, a period to the end of the second line (remember, the pattern merely produces a declarative sentence) and a second period to the end of the last line. Examine the three other formats given in the program and you will find that more than one kind of punctuation may be used at the ends of some lines.

Punctuation possibilities also include inserting commas between adjectives any place they appear next to each other, as they do in the first line of type 1, for example. This is the preferred method of punctuating such sentences as:

**THE BIG, BLUE BALL.**

Punctuation isn't always handled that way, but it is correct in this case to do so.

***Proper Use of "A" and "An"*** It would be relatively straightforward to examine the first syllable of each word preceded by a particle, find out if the item in question requires "a" or "an" and make the appropriate substitution. As it stands, the program frequently misuses those two articles, producing phrases like "an red afternoon" and "a awesome sunset." Cleaning up such usage would take the program a step closer to producing human-sounding poetry.

***Eliminating Duplicate Word Use*** We indicated in our discussion of ADD.VOCABULARY routines that it would be possible, though perhaps unnecessary, to eliminate duplicate words from the vocabulary of the program. The only "harm" caused by duplicate words in the file is the slight increase in the probability that the word will be used more than once in a poem—which occasionally might even be desirable. But if Poetry Maker produces something like this:

```
SOFT SOFT SEA
A MOUNTAIN ROARS NEAR SOFT WAVES
QUIET SOFT MORNING
```

one's mind could turn soft reading it. You can use a number of methods to avoid the duplicate use of words in a poem. Perhaps the easiest is to build a variable that lists all words used and checks each word as it is found by GET.NEXT.WORD against the contents of this list. If the word is found, the program could generate another random number (or, alternatively, the routine could move up or down the list two positions to the next word) and pick a new word for comparison and possible inclusion.

***True Haiku Patterns*** The final change is the reason a syllable count of each word was included in property lists and files of words learned by the program. True Haiku has five syllables in the first line, seven in the second, and five in the third. Using the syllable count in the property lists, it is possible to construct Haiku poetry that fits this traditional set of rules.

A word of caution is in order. It is not just a matter of keeping track of how many syllables are used and merely subtracting. For example, suppose you generate a poem of type 1 and you are on the first line where the pattern is adjective-adjective-noun. If you've already picked two adjectives and they're each one syllable long, but you don't have a three-syllable noun in your vocabulary, you're in trouble. Also, if you use the current random selection of words and if there are a limited number of three-syllable words and each word has to be checked before it can be used, you may find yourself with a program that takes longer to run than you have patience for. Several iterations through the noun property list might be required before the correct word is found.

You might solve this problem by altering the patterns so that in place of the present PATTERN1 in POEM1 you would have a pattern like this:

```
MAKE PATTERN1 [A 2 A 1 N 2]
```

The program would be revised to look for a two-syllable adjective, followed by a one-syllable adjective, followed by a two-syllable noun. This is more complex and restricts the poetic output a bit more than our original design, but it's an approach that would work. You can undoubtedly think of others.

## Adding Flexibility

Two relatively easy changes would make Poetry Maker more flexible from the user's perspective. One involves permitting the user to enter a poetic pattern and the other permits use of more than one word file.

***Give Me a Pattern Where the Poets Roam . . .*** The first modification would be to ask the user, in the SET.PATTERN routine, to enter either a number from 1 to 4 or another number to indicate a desire to enter a new pattern. Then the user enters three lists consisting of the six letters representing the parts of speech and generates a poem using that pattern.

The poem does not have to be limited to three lines, but if you change this, you must modify the COMPOSER routine accordingly.

As an adjunct to this idea, you *could* design the program to store poetic formats in a disk file, just as it stores vocabulary. Then large numbers of forms could match large, perhaps multiple, vocabularies.

***Lots of Kinds of Words*** Another modification would permit users to tell the name of the file containing the words they want to work with. This would permit files of different kinds of words for different kinds of poetic moods. For example, one might be called "Romance Words," another "Nature Words," and still another "Humor Words." Then, depending on mood, the user could create any of several kinds of poems.

This approach, if combined with permitting users to enter a poetry format, would result in a highly flexible and recreational program. It might even have some limited commercial potential!

## Adding Intelligence to Poetry Maker

If you're ready for a *real* challenge, try adding *semantic content* considerations (i.e., meaning) to the Poetry Maker. A complete discussion of how this could be done is beyond the scope of this book, but a few ideas might get you started in the right direction.

As the program is now designed, it can create pretty nonsensical sentences like:

```
MOUNTAIN FOR THE LEAVES
SEA TO A WATERFALL
```

and others. The problem is that words picked at random don't necessarily bear any semantic relationship to one another. "Sea *near* a waterfall" and "mountain *over* the leaves" make sense in a poetic way, but our sentences do not.

It would be possible to add some information to the property list about each word. As one minor example, if the user enters a noun, we could ask if it's the name of a place or a thing, and add a property list entry accordingly. Or we could ask if the thing was movable or not and add a property for that condition. Then if we added information about verbs that indicated whether they are applicable to objects that cannot move, we'd avoid nonsense like:

```
THE MOUNTAIN DROVE HOME.
```

You get the idea. By giving the program more information about each word—in the form of additional data fields in the property lists—we could give our poems a better chance of making sense.

## Summary: What We've Learned about AI Programming

In this chapter we've discussed some key concepts involved in natural language processing. We've laid the groundwork for programs coming up which are part of the NLP thread. We've examined key issues in text generation and reviewed the degree to which programs that produce text are useful tools for helping us learn about how language is put together. We have looked at how patterns can be used as ways of formatting results in addition to their more traditional AI role of being matched for identification of unknown items.

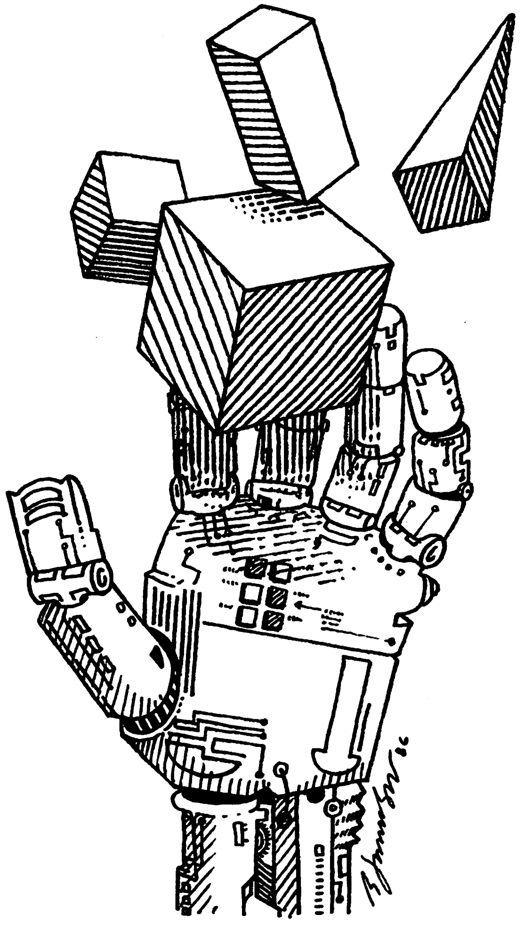
```
                    {Poetry Maker ©1985, The Waite Group}
                        {Logo Program by Dan Shafer}

TO POETRY.MAKER
  CLEARTEXT
  PRINT <<Do you want to:>>
  PRINT <<>>
  PRINT <<[A]dd Vocabulary Words to the File>>
  PRINT <<[M]ake Up Poems(s)>>
  PRINT <<[Q]uit>>
  MAKE MENU.CHOICE READCHAR
  IF :MENU.CHOICE = 'A [ADD.VOCABULARY]
  IF :MENU.CHOICE = 'M [MAKE.UP.POEM]
  IF :MENU.CHOICE = 'Q [STOP]
  POETRY.MAKER
END

TO ADD.VOCABULARY
  GET.WORDS
  MAKE WORD.LIST []
  MAKE SENT2[]
  MAKE ARTICLES [A AN THE]
  INPUT.NEW.WORDS
  DELETE.ARTICLES
  MAKE WORD.LIST BUTLAST :SENT2
  GET.WORD.INFO
  UPDATE.FILE
END

TO INPUT.NEW.WORDS
  PRINT <<Please enter new words, separated by spaces.>>
  PRINT <<You can even enter a whole new poem if you like!>>
  PRINT <<>>
  PRINT <<When you're done, just enter a RETURN at the start>>
  PRINT <<of a line. Then I'll ask you about the new words>>
  PRINT <<you've given me before I add them to the vocabulary file.>>
  ADD.A.LINE
END

TO ADD.A.LINE
  MAKE LINE READLIST
  IF EQUALP :LINE [NIL] [STOP]
  BREAK.LIST
  ADD.A.LINE
END

TO BREAK.LIST
  IF EMPTYP :LINE [STOP]
  MAKE WORD.LIST LPUT FIRST :LINE :WORD.LIST
  MAKE LINE BUTFIRST :LINE
  BREAK.LIST
END

TO DELETE.ARTICLES
  MAKE WORD 1 FIRST :WORD.LIST
  IF EQUALP MEMBERP :WORD 1 :ARTICLES NIL [MAKE SENT 2 LPUT :WORD 1
  :SENT 2]
  IF EMPTYP :WORD.LIST [STOP]
  MAKE :WORD.LIST BUTFIRST :WORD.LIST
  DELETE.ARTICLES
END

TO GET.WORD.INFO
  MAKE WORD 1 FIRST :WORD.LIST
  PRINC [HOW MANY SYLLABLES DOES<< >>] PRINC :WORD 1 PRINT
  [<< >>HAVE?]
  MAKE SYLLABLES READCHAR
  PRINC [WHAT PART OF SPEECH IS<< >>] PRINC :WORD 1 PRINT <<?>>
  PRINT [N = NOUN V = VERB A = ADJECTIVE D = ADVERB P = PREPOSITION]
  MAKE WORD.TYPE READCHAR
  PPROP :WORD.TYPE :WORD 1 :SYLLABLES
  MAKE WORD.LIST BUTFIRST :WORD.LIST
  IF EMPTYP :WORD.LIST [STOP]
  GET.WORD.INFO
END

TO UPDATE.FILE
  MAKE WORDFILE OPEN_WRITE <<Logo Files:Poetry Words>>
  (PRINT PLIST "N :WORDFILE)
  (PRINT PLIST "V :WORDFILE)
  (PRINT PLIST "A :WORDFILE)
  (PRINT PLIST "D :WORDFILE)
  (PRINT PLIST "P :WORDFILE)
  CLOSE_FILE :WORDFILE
END

TO MAKE.UP.POEM
  GET.WORDS
  IF EQUALP :WORDFILE NIL [PRINT <<I can't compose poetry with no
  vocabulary!>> STOP]
  SELECT.PATTERN
END

TO SELECT.PATTERN
  PRINT <<I know four Haiku poetry patterns.>>
  PRINT <<>>
  PRINT <<Enter the number of the pattern you want me to use (1-4)>>
  PRINT <<or use any other number to tell me to pick one at random.>>
  PRINT <<>>
  MAKE POEM.TYPE READCHAR
  IF NOT NUMBERP :POEM.TYPE [SELECT.PATTERN]
  SET.POEM :POEM.TYPE
END

TO SET.POEM :POEM.TYPE
  IF :POEM.TYPE = 1 [POEM1]
  IF :POEM.TYPE = 2 [POEM2]
  IF :POEM.TYPE = 3 [POEM3]
  IF :POEM.TYPE = 4 [POEM4]
  IF NOT MEMBERP :POEM.TYPE [1 2 3 4] [SET.POEM RANDOM 4 + 1]
END

TO POEM1
  MAKE PATTERN1 [A A N]
  MAKE PATTERN2 [R N V P A N]
  MAKE PATTERN3 [A A N]
  COMPOSER
END

TO POEM2
  MAKE PATTERN1 [N P R N]
  MAKE PATTERN2 [R A N P R N]
  MAKE PATTERN3 [A N]
  COMPOSER
END

TO POEM3
  MAKE PATTERN1 [R A A N]
  MAKE PATTERN2 [P R A N]
  MAKE PATTERN3 [R N V]
  COMPOSER
END
```

```
TO POEM4
  MAKE PATTERN1 [R A N V]
  MAKE PATTERN2 [R A A N]
  MAKE PATTERN3 [P R A N]
  COMPOSER
END

TO COMPOSER
  MAKE LINE []
  COUNT.WORDS
  MAKE NO.__R 3
  MAKE PAT :PATTERN1
  DO.LINE
  MAKE PAT :PATTERN2
  DO.LINE
  MAKE PAT :PATTERN3
  DO.LINE
  RUN.AGAIN
END

TO COUNT.WORDS
  MAKE NO.__N COUNT PLIST 'N
  MAKE NO.__V COUNT PLIST 'V
  MAKE NO.__A COUNT PLIST 'A
  MAKE NO.__D COUNT PLIST 'D
  MAKE NO.__P COUNT PLIST 'P
END

TO DO.LINE
  MAKE WORD1.TYPE FIRST :PAT
    IF :WORD1.TYPE = 'N [MAKE LOC (RANDOM :NO.__N) + 1 GET NEXT.WORD]
    IF :WORD1.TYPE = 'V [MAKE LOC (RANDOM :NO.__V) + 1 GET NEXT.WORD]
    IF :WORD1.TYPE = 'A [MAKE LOC (RANDOM :NO.__A) + 1 GET.NEXT.WORD]
    IF :WORD1.TYPE = 'D [MAKE LOC (RANDOM :NO.__D) + 1 GET.NEXT.WORD]
    IF :WORD1.TYPE = 'P [MAKE LOC (RANDOM :NO.__P) + 1 GET.NEXT.WORD]
    IF :WORD1.TYPE = 'R [MAKE LOC (RANDOM :NO.__R) + 1 GET.ARTICLE]
  MAKE PAT BF :PAT
  IF EMPTYP :PAT [PRINT :LINE MAKE LINE [] STOP]
  DO.LINE
END

TO GET.NEXT WORD
  IF EQUALP REMAINDER :LOC 2 0 [MAKE LOC :LOC - 1]
  MAKE NEXT.WORD ELEMS :LOC 1 PLIST :WORD 1.TYPE
  MAKE LINE LPUT FIRST :NEXT.WORD :LINE
END

TO GET.ARTICLE
  MAKE NEXT.WORD ELEMS :LOC 1 [A AN THE]
  MAKE LINE LPUT FIRST :NEXT.WORD :LINE
END

TO RUN.AGAIN
  PRINT <<Do you want me to compose another poem?>>
  MAKE ANSWER READCHAR
  IF :ANSWER = 'Y [SELECT.PATTERN]
END

TO GET.WORDS
  MAKE WORDFILE OPEN_READ <<Logo Files:Poetry Words>>
  IF EQUALP :WORDFILE NIL [PRINT <<New File Being Created This Time>> STOP]
  MAKE WORDTYPES [N V A D P]
  REBUILD.PLISTS
  CLOSE_FILE :WORDFILE
END

TO REBUILD.PLISTS
  MAKE TEMP1 (READLIST :WORDFILE)
  BUILD.A.LIST
  MAKE WORDTYPES BF :WORDTYPES
  IF EQUALP (END_OF_FILE :WORDFILE) T [STOP]
  REBUILD.PLISTS
END

TO BUILD.A.LIST
  PPROP FIRST :WORDTYPES FIRST :TEMP1 FIRST BF :TEMP1
  MAKE TEMP1 BF BF :TEMP1
  IF EMPTYP :TEMP1 [STOP]
  BUILD.A.LIST
END
```

CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELL
ELLIGENCE ARTIFICIAL IN
L INTELLIGENCE ARTIFICIA
ICIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELL
ELLIGENCE ARTIFICIAL IN
L INTELLIGENCE ARTIFICIA
ICIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELL
ELLIGENCE ARTIFICIAL IN
L INTELLIGENCE ARTIFICIA
ICIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELL

# 5

# *Micro Blocks World*



- What Parsing Is
- Bottom-Up vs. Top-Down Parsing Strategies
- Grammar and Its Importance in NLP
- Dividing Input into Manageable Groups for Analysis
- The Complexities of NLP Emerge

In the early 1970s, an AI researcher named Terry Winograd wrote a program called, unpronounceably, SHRDLU. (Unlike the names of other programs, Professor Winograd picked this name because it had *no* meaning!)

SHRDLU displayed an amount of apparent intelligence previously unknown to computers. Within its knowledge domain, it could respond to complex commands written in ordinary English (for example, "Find a block taller than the one you are holding and put it into the box."). More significant for AI research, SHRDLU could also explain *why* it did what it did. You could ask it, "Why did you pick up the green pyramid?" and it would respond with an explanation like, "The green pyramid was on top of the blue block. You asked me to put the red block on top of the blue block. I had to clear the top of the blue block."

Any program that can understand reasonably complex English commands and then explain its behavior apparently evidences a fascinating degree of intelligence. People who saw and ran SHRDLU certainly felt that way. In point of fact, SHRDLU is neither complicated nor revolutionary in light of some of the research that has followed it in the past decade or so. But it remains an interesting program to contemplate and to use.

We will examine a scaled-down version of the famous SHRDLU program, called "Micro Blocks World." We will prepare to understand the program and the AI implications it presents by first studying parsing—one of the key new ideas the program presents. Parsing is crucial to natural language processing design and to AI research in general.

## Parsing Natural Language

The idea of parsing, and the term itself, originated with linguists. Linguists use parsing to refer to the process of analyzing language and breaking it into its component parts. There is a strong resemblance between that definition of parsing and what computer scientists call "parsing input." There are some differences, however.

When computer scientists speak of parsing, they may not be referring to language in the traditional sense. Not only inputs, responses, and natural language entries are parsed in the world of computers. Some data base management programs store data in a compact stream of characters. When such data is retrieved, it must be broken into various fields and groups of characters. This process is sometimes referred to as parsing.

An interesting symbiosis has been going on in the study of language. Prior to the emergence of natural language processing (NLP) and the related field of study known as *computational linguistics,* there was little need for scientists to think precisely about meaning, syntax, and structure in language. Humans intuitively understood what was intended. But the introduction of computers and their insistence on precision of communication forced linguists to take a closer, more precise look at languages.

The study of language benefited from the inspection forced by the computer—and from the theoretical ideas growing out of that inspection. Computer science benefited as linguists learned more about how humans communicate with language. The linguists provided AI researchers and other computer people with accurate and useful data about the processes involved. This, in turn, led to greater insight into the role computers and programs might play in the process.

## What Parsing Is

For our purposes, parsing is the process in which a computer program takes apart natural language to gain insight into the meaning of the words in a particular sentence or phrase. However, this definition of parsing is at once too simple and too complex.

The definition is too simple from the perspective of the serious AI researcher who wants to be sure that a definition of parsing incorporates such ideas as syntax, sources of knowledge, and data structures for knowledge representation. On the other hand, the definition is too complex in that, even stated simply, the idea is difficult to grasp and to deal with in a computer program. Nonetheless, the definition is sufficient for our needs in this chapter.

It is interesting to investigate the process of parsing more rigorously than the Micro Blocks World program requires. So in the next few pages we will examine parsing on a broader scale.

## Parsing Techniques and Strategies

In designing a full-fledged parser for a complete AI system, a programmer would be forced to make some important design trade-off decisions. A discussion of all of these decisions is beyond the scope of this book, but one aspect is of particular interest to us.

***Bottom-Up vs. Top-Down Design*** A parser can either start with the goals (i.e., the set of possible sentence structures) or it can begin with the words actually in the sentence being parsed. The former technique is referred to as "top-down" and the latter as "bottom-up."

For example, a top-down parser would begin parsing a sentence by looking at the rules for a sentence, then looking for the constituent parts of a sentence (clauses, phrases, etc.), and then looking to *their* constituent parts until it has composed a complete sentence structure from the rules. If this sentence matches the pattern of the sentence being parsed, the program is successful; otherwise, it starts back at the top again, generating a different sentence structure to compare.

The bottom-up parser would attempt to combine the words and word groups in the input sentence into larger and larger structures, trying to recombine them to prove that all of the input words together form a legal sentence in the grammar. In theory, both parsers reach the same conclusion about the same input, given the same grammar. Many large-scale parsers combine both techniques and use them interactively with one another.

One frequently recurring discussion in AI is whether top-down or bottom-up parsing techniques will lead to more efficient or accurate results. The details of that discussion are esoteric, but one thing is safe to say: there is no way to conclude yet that one approach is superior to the other. They are simply different ways of pursuing similar goals.

Micro Blocks World performs bottom-up parsing. It begins with the words entered by the user as a command and attempts to find the specific parts of the entry with which the program must concern itself. The bottom-up approach works best in this instance because of the program's limited knowledge base, the predictable nature of the input, and the narrow set of rules within which parsing must take place.

***Combining Words in Bottom-Up Parsing*** A related design issue in parsing concerns the method by which portions of input—technically called "constituents"—are to be combined in a bottom-up approach. Words are generally dealt with in groups. The rules by which words are combined into these groups may affect the ability of the parser to "understand" the meaning of the input.

Our Micro Blocks World program requires three elements: identification of the subject and the object and the operation to be performed by one on the other. That makes it easy for us to combine words into groups by their position relative to the basic command word or verb. The program finds this verb, splits the sentence into two pieces and then searches for the other necessary components within these pieces.

The more complex are the rules associated with the language being implemented and understood by the program, the more difficult and complicated the issue of design becomes.

## Examples of Parsing

Parsing is easier to understand if we apply some of the basic ideas of the technique to a simple but potentially ambiguous English sentence.

***Vocabulary: A Starting Point*** Parsers generally begin with a basic vocabulary of words they understand as being certain parts of speech or playing specific roles in communication. For example, a parser might be given the information that the word "give," along with its variants "gives," "gave" and "given," is transitive, requiring a direct object (the thing that is given) and an indirect object (the person to whom the thing is given).

The parser might also know that any word beginning with a capital letter is the name of a person. To deal with the possibility that the first word in a sentence will be capitalized but not necessarily a proper name, we provide a second rule that says that if the program can make sense of the first word in a sentence by some other rule, it should use that rule. If not, it should assume that name is a proper noun, too.

Let's also have our program understand some nouns that serve as direct objects. We could choose "bell," "book," and "candle," for example. When the program encounters these words, it will know that they are potential direct objects of the sentence.

We'll add one other rule about vocabulary to our knowledge base: the words "a," "an," and "the"—the two particles and the article—are "throwaway words," which can be simply ignored.

We should note here that not all of these rules would be valid in all parsers; in fact, this last rule about throwaway words would almost certainly *not* be acceptable in a truly useful language parsing program. Whether or not a word has an article can affect the use of the word in the sentence, but in complex ways.

***Rules to Be Applied*** A parser will also have a set of rules, called a "grammar," to apply to data it receives from the user. We'll discuss grammars later, but for now assume our parser has one rule. It knows that a valid sentence will include the word "give" or one of its variations, that the next word group it encounters will define the indirect object, and that the final word group will describe the direct object.

The parser also knows that a word group can consist of throwaway words, nouns it understands, and any number of other words (adjectives, adverbs and the like) that must be dealt with by some other part of the parser.

Finally, it knows that the direct object portion of the input sentence will always begin with a throwaway word. (This is almost always true in real-life English as well.)

***A Sample Sentence*** Now let's give our "program" a sentence to work with:

**John gave Mary a book.**

Our program could easily handle this group of words. Parsing from left to right and using a bottom-up approach, the program first looks at the word "John," notes that it is capitalized but that it is the first word in the sentence. It then looks in its vocabulary to see if "John" is a word it knows in some other context. Not finding it, the program concludes that John is the name of a specific person.

Now the parser can move to the word "gave." This word is a variant on the main verb it understands, "give." Therefore, the sentence is potentially a sentence that it can parse. It moves to the next word.

"Mary" is capitalized and does not appear at the beginning of a sentence; therefore, it is a proper noun. The program knows that a verb, "gave," can be followed by an indirect object. No great amount of thinking or analysis is needed here.

The word "a" is a throwaway word. The program knows that the direct object portion of a valid sentence for our arbitrary set of rules could begin with such a throwaway word, so it notes that it may now be dealing with the direct object and throws the "a" away.

"Book" is a noun the parser knows can serve as a direct object. Since it has already determined from previous analysis that it may be dealing with the direct object part of the input sentence, it now concludes that "book" is the direct object of the sentence it is parsing.

The period signals the end of the sentence, so parsing stops and the program can now be said to have understood and assimilated this sentence.

***A Bit More Complexity*** Just so we can see what our hypothetical parser will do when faced with more difficult problems, let's create a slightly more complex sentence for it to parse.

**The big man gave little Mary a purple book.**

Parsing from left to right, the program first analyzes the word "The." Noting that it is capitalized, but the first word in the sentence, the parser checks to see if it knows the word from some other context. It finds that it is a throwaway word and therefore ignores it and moves to the next word.

The two words "big" and "man" receive identical treatment. Neither is capitalized, neither is known to the program. The parser does what it has been told to do with any words it doesn't know. (Possibly keeping them in reserve for later use.)

Next, the familiar word "gave" appears. The program now knows that this is a sentence it can possibly handle. It expects to confront a direct object at this point. (If the parser has been designed to do so, it will also now conclude that "big man" is a word group that serves as the subject of the sentence.)

The word "little" gets the same handling as the words "big" and "man"; it is set aside for the moment as not conforming to any known rules.

"Mary," on the other hand, is a known quantity—a proper noun, and, therefore, the indirect object of the sentence. (Again, depending on design, the parser might even conclude that "little Mary" is an indirect object word group.)

You can probably figure out that the next word group will be handled by treating "a" as a throwaway word, setting aside "purple" as unrecognizable, and noting "book" as a legal direct object. The period flags the end of the sentence and the parser now "knows" that the sentence is valid, that it is likely that the subject is "big man," that the indirect object is "Mary" or perhaps "little Mary" and that the direct object is "book," or perhaps "purple book."

***A Final Test*** What will a parser such as this do when it encounters a sentence outside its realm of knowledge? Let's try it on one:

**The 1985 San Francisco Giants played baseball poorly.**

Our parser first determines that "The," is a throwaway word. It then looks at "San" and, since it is capitalized and not the first word in the sentence, decides that it is the name of a person. It does the same thing with "Francisco" and "Giants."

None of the last three words mean anything to the parser, which has never been told about the verb "play" or the noun "baseball" or the adverb "poorly." But all of a sudden, there's a period, so the parsing assignment is over.

The parser cannot interpret this sentence. How it deals with that fact depends upon how the program has been designed to deal with the unknown. In any case, the parser would not acknowledge that this sentence is valid since it doesn't comply with the parser's knowledge of what a valid sentence is.

### Parsing and Other NLP Techniques

Real-life NLP programs in AI labs also include such things as generators, inference engines, and memories in addition to parsers. Generators create parsable sentences from information passed to the program. Inference engines draw conclusions from information. Memories build up information over time and permit parsers to adapt to changing information.

We examined one crude text generator in the Poetry Maker program. In Chapter 7 we present an inference engine of sorts that implements a subset of a full-fledged AI programming language written in ExperLogo®. Memories are too complex a subject to be dealt with in this book.

In any case, all NLP programs include a parser of some kind as part of their design. So understanding parsing is an important part of understanding NLP generally.

## The Role of Grammars

We mentioned the use of grammars several times in the course of describing the process of parsing. We now turn our attention to a brief examination of grammars as they relate to computers.

## What a Grammar Is

We all studied grammar in school. A few people love studying grammar—including professional linguists and writers (some of whom view themselves as saviors of a rapidly vanishing "pure" English language). However, the subject is not generally relished by students (or, for that matter, by anyone else).

But when it comes to computers, grammar is given a very specific definition. A "grammar" is that portion of an NLP program that consists of the rules of what constitutes a grammatical (i.e., correct or understandable) sentence and what makes up its component parts.

***Sentence Grammar*** We could have a grammar with only one rule: anything that begins with a capital letter and ends with a period, question mark, or exclamation point is a correct input. This grammar would be easy for a computer to implement, but it has two primary drawbacks: first, it doesn't tell us what the sentence is or means; second, it fails to match what we think of as "correct" grammar. It would pass all of the following as complete or correct inputs:

The cow jumped over the moon.

We are one.

Cats, dogs, chickens.

It.

Somewhere over the rainbow!

Taking a walk?

Grzwilly frbl sksb!

On the other hand, it would reject, as incorrect sentences, all of the following inputs:

ready? ←no capital letter at the beginning
"Help me, please." ←doesn't end with known punctuation mark
e e cummings ←no capital, no punctuation

Also, it would never tell us anything about input except that it was a valid or an incorrect sentence—not particularly helpful information. It is certainly not worth the power of a computer program.

***Subsentence Grammar*** The next logical level of sentence parsing differentiates between subjects and predicates. We can define a grammar that, along with associated vocabulary, contains the following rules:

1. If the sentence ends with a period, look for the verb in your vocabulary. Everything coming before the verb is the subject; the verb and the rest of the sentence are the predicate. A subject always acts on a predicate.
2. If the sentence ends with a question mark, the first word is a helper that belongs with the verb. Find the verb. Everything that appears before the verb and is not a helping verb, is the subject. The helper verb and the verb along with the rest of the sentence, is the predicate. Such a sentence always asks if the predicate is true of the subject.

Such a grammar will correctly recognize and analyze the following sentences:

The subject of this sentence acts on the predicate.

We entered the room calmly.

Does she look healthy to you?

Are you going to the dance?

It will not, however, deal correctly with these sentences:

How many are in your party? ←no helper verb before verb
Quaking in his boots, the man turned away. ←introductory clause
"Merry Christmas," he said to me. ←no rules about quotes

***The Role of Context*** AI researchers are coming to the conclusion that grammar rule sets, in and of themselves, do not provide a sufficiently rich framework within which to attempt the machine interpretation of natural language. The most complex parser-generator combined with the most capable and rich grammar would still have trouble with the common sense in the following sentences:

We saw Detroit flying to New York.

Jane was hurt.

Both sentences are valid and correct, from a grammatical perspective. The problem is that they are ambiguous on a *contextual* level. To make sense of them, we need to know something about the context within which they are said.

The first example, "We saw Detroit flying to New York," would not, in context, give human listeners any trouble. But for a computer to understand it correctly, the program would have to know something about flight, about which objects fly and which don't, about what New York and Detroit are, and perhaps a number of other facts. It is not difficult to present such information to a computer program or to store it, but such rules are neither grammars nor vocabularies. They are, rather, contexts or concepts.

The second sentence is ambiguous even to a human listener. We do not know whether Jane was hurt physically or emotionally. We need to know the context, the action that took place just prior to the statement.

The importance of context or concept has led a number of researchers—notably those at Yale University's AI Laboratory under the direction of AI guru Dr. Roger Schank—to focus on issues other than grammars in attempting to deal with NLP problems and designs.

***And So On . . .*** We could go on, adding more and more "knowledge" about grammar, syntax, semantics (whatever *they* are!), vocabulary, and context. But the conclusion is probably clear already; parsing is no simple matter. Without it, however, NLP would remain an unattainable idea.

Let's turn our attention now to the program we will be working with in this chapter. Although the program uses a simple form of parsing, it will clearly demonstrate the role parsing plays and one way parsing is undertaken by NLP programs.

## The Program

SHRDLU has a tradition in AI computing circles. Since you will encounter our program's "big brother" if you do any serious reading in AI literature, we will take a few moments to explore what the original program did—and how it did it—before jumping into our microcomputer version.

**Original Program**

When Winograd introduced SHRDLU to the world of AI, he offered his colleagues a radical departure from earlier work in NLP. The program represented a robot (after whom the program is named) that responded verbally and graphically to human commands.

This program "knew" about a small universe of a tabletop holding various sizes of blocks and boxes of various shapes and colors. (Thus the name "Blocks World"; it is a world of blocks about which the program has knowledge.) When told to do something with a particular block or box, SHRDLU carries out the instruction and either informs the user of its success or asks for clarification.

Within the confines of this small universe, SHRDLU is a real expert. For example, it knows that things put on top of pyramids will fall off. It knows that to move an object which is supporting another object, it must first move the top object.

But SHRDLU's real strength lies in its ability to deal with English language input. In this sense, it is a truly remarkable program, even by today's more advanced standards. Let's look at a sample of the kind of dialog in which SHRDLU could engage.

**Sample Dialog**

Figure 5-1 shows the universe of the SHRDLU robot at the beginning of the dialog. (SHRDLU is capable of dealing with a far larger number of objects than this, but we will reduce the universe for the sake of clarity and brevity.)

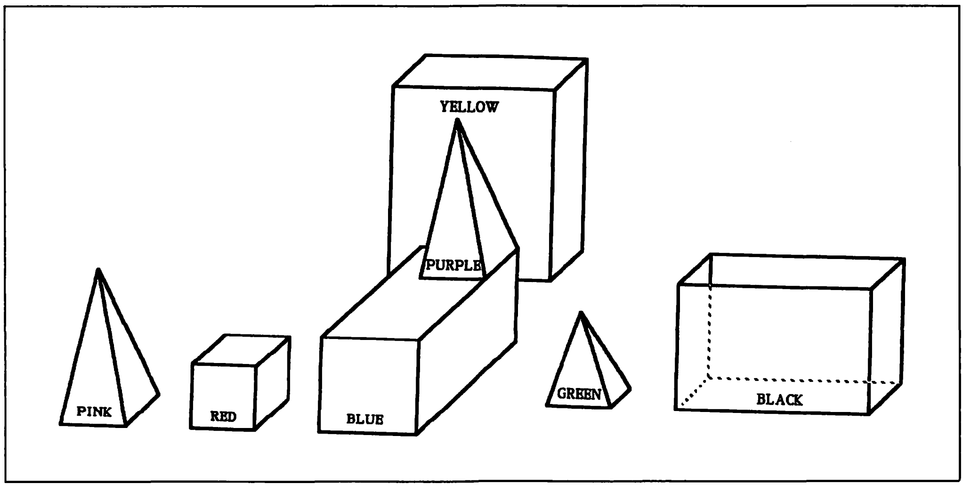


**Figure 5-1. Starting point for SHRDLU dialog**

Now let's follow a few lines of dialog between the user of this program (in capital letters) and the computer (in lower case letters and indented from the left margin):

**PICK UP THE BIG YELLOW BLOCK.**
  **ok.**
**PUT IT IN THE BOX.**
  **ok.**

(Note: figuring out what "it" means is not always so easy, as we shall soon see; it is here that SHRDLU shines!)

**PICK UP THE BIG PYRAMID.**
  **i don't understand which pyramid you mean.**

(There are two large pyramids.)

**PUT THE BLUE BLOCK IN THE BOX.**
  **ok.**
**WHERE IS THE PURPLE PYRAMID?**
  **the purple pyramid is supported by the table.**
**WHEN DID YOU MOVE IT THERE?**
  **when i was putting the blue block in the box.**
**WHY?**
  **to clean off the blue block.**
**WHY?**
  **because you told me to.**

(The machine's ultimate justification!)

**PUT THE YELLOW BLOCK BEHIND THE PURPLE PYRAMID.**
  **ok.**
**PUT THE PINK PYRAMID ON TOP OF THE GREEN PYRAMID.**
  **i can't.**
**WHY?**
  **a pyramid cannot support an object.**
**PUT THE PINK PYRAMID ON TOP OF THE YELLOW BLOCK.**
  **ok.**
**PUT THE RED BLOCK ON TOP OF THE YELLOW BLOCK.**
  **ok.**
**WHAT IS THE TABLE SUPPORTING?**
  **the purple pyramid, the yellow block, the green pyramid, and the black box are supported by the table.**
**PUT IT DOWN.**
  **i can't.**

(Note that it doesn't need to know what "it" we are referring to; any "it" won't work since it is "empty-handed.")

**WHY?**
  **i am not in possession of anything now.**

Pretty smart program, eh? Yet, it knows about only a very limited knowledge domain and only responds mechanically to information it is given and requests it is handed.

**What Micro Blocks World Does**
The Macintosh® ExperLogo® implementation in this program is not nearly as elaborate as Dr. Winograd's original product. Nonetheless, it is fairly sophisticated and we think it is instructive and enjoyable. The program contains several of the fascinating features that made SHRDLU such a widely studied and copied program.

***What It Knows*** The universe of Micro Blocks World consists of two blocks and two pyramids. One of each of these pairs is larger than the other so that there are four easily differentiated objects: a large block, a small block, a large pyramid, and a small pyramid.

The *vocabulary* contains recognizable nouns (block and pyramid), adjectives (small and large) and "operators" (top, right, and left).

Rules of the *grammar* by which Micro Blocks World interprets input and responds to "commands" are:

1. A grammatically correct sentence contains one operator preceded by a subject and followed by an object.
2. A subject and an object each consist of one adjective and one noun.
3. Everything that isn't an operator, a known noun, or a known adjective is a throwaway word and may be ignored.
4. Punctuation is *not* permitted.

The laws that govern the program's world include:

1. An object can't be placed next to or on top of itself.
2. Objects can't be stacked on top of pyramids.
3. Objects can't be positioned outside the range of the tabletop.
4. Objects can't be placed next to objects that are not themselves positioned directly on the tabletop.
5. Subjects that are under other objects can't be moved.
6. Two objects can't occupy the same position at the same time.

This is the sum total of the knowledge of the program. Obviously, it is far more limited than SHRDLU. But the principles upon which it operates and the means by which it knows things are quite similar.

***What It Understands*** Given the rules and knowledge base, we can see that Micro Blocks World "understands" commands that tell it to do things with objects. More precisely, the program understands instructions that tell it to move one of the four objects into a new spatial relationship with another of the four objects. Because the program focuses attention upon subject-object combinations placed on either side of a known operator, it can deal intelligently with a wide range of commands. All the following will be understood and responded to properly by our program.

**Note**

Do not end sentences with punctuation. Doing so will cause the program to behave unpredictably.

**Put the large pyramid to the right of the small block**
**Large pyramid right small block**
**Pick up the large pyramid and move it just to the right of the small block please**

The following sentences would not be understood but in the program attempts to elicit missing information from the user.

**Put it to the right of the large block.**
**Put the block to the left of the small pyramid.**
**Put the large pyramid on top of the block.**

How the computer specifically reacts to such situations is covered in the next section of this discussion.

***How It Reacts*** When you enter a command into the Micro Blocks World's Listener Window, one of three things can happen.

First, if the command is complete and your instruction can be followed, the program will do what you tell it and await another command. (You will *see* your instruction carried out in ExperLogo's® Graphics Window.) A complete command has a subject containing an adjective and a noun, and an object containing an adjective and a noun, with a known operator between the two.

Second, if the command you enter contains a known operator but is missing some information, the program will request clarification until it is sure it understands your command. Then, if possible, it will carry out your instructions.

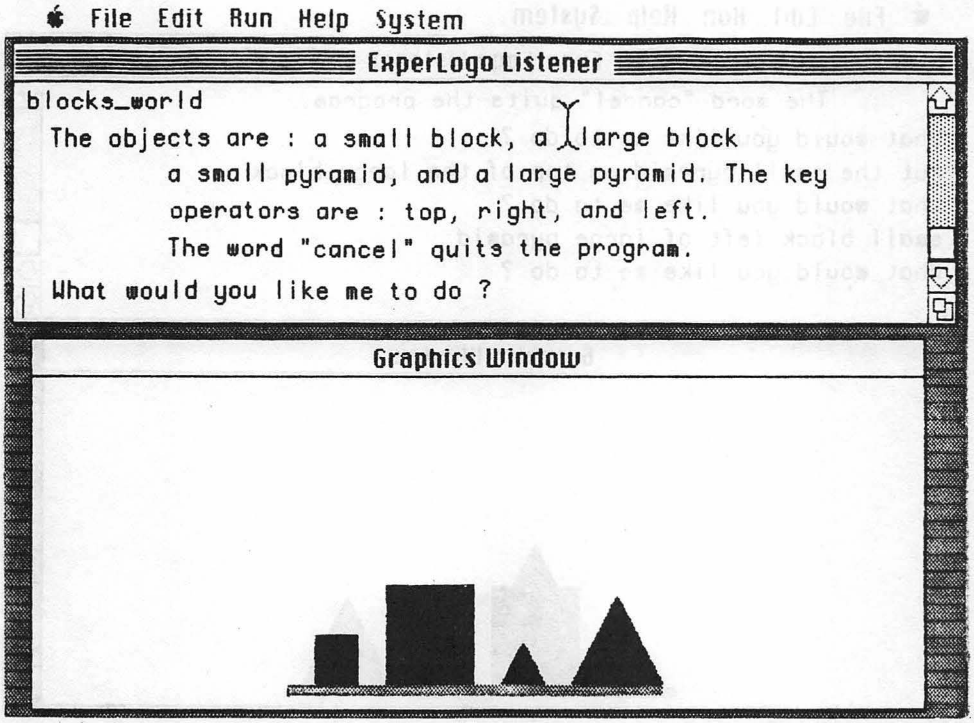
Finally, if you enter a command which has no operator—and therefore makes no sense to the program—it will tell you it doesn't know how to carry out that instruction and ask for another.

There is one other command the program understands; the word "cancel" stops the program from running and returns you to ExperLogo®.

***A Small Sample Session*** When you start the program by typing in the command BLOCKS_WORLD, the screen will look like Figure 5-2. Figures 5-3 through 5-8 depict a typical session running Micro Blocks World. The interaction with the program takes place in the ExperLogo® Listener Window and results are displayed in the Graphics Window.

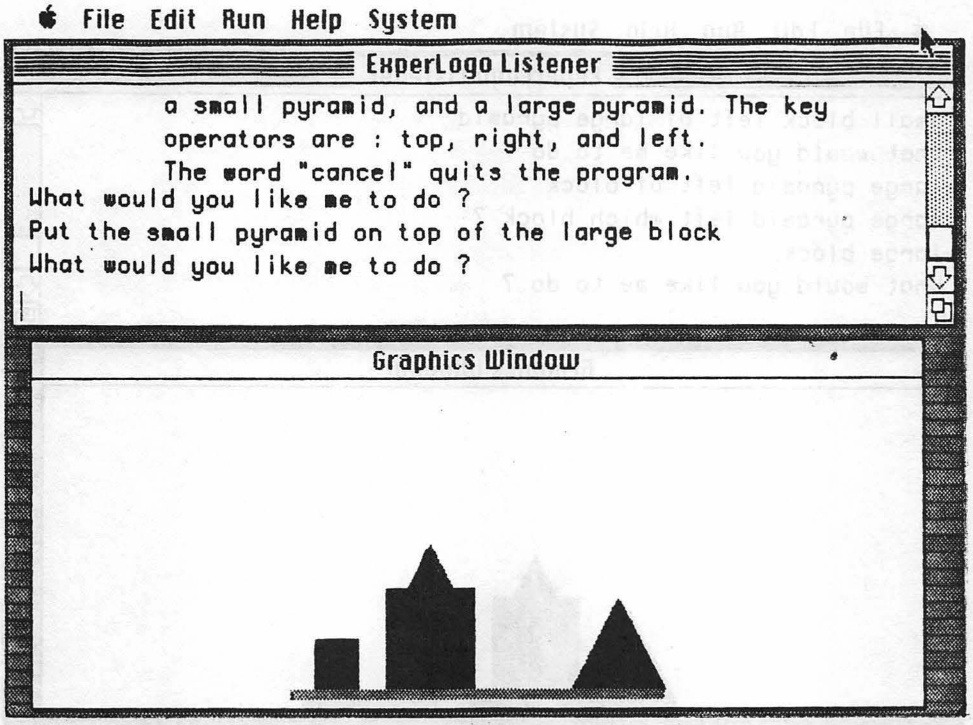
If we ask the program to carry out an instruction with more than one piece of required information missing, it will doggedly pursue the matter until we have told it everything it needs to know, as you can see from Figure 5-9.

If you let your friends work with the program for a while, they will probably conclude that the program is intelligent. It seems to understand human input and knows how to ask for information needed to use that information wisely. In short, it's a "smart" program!



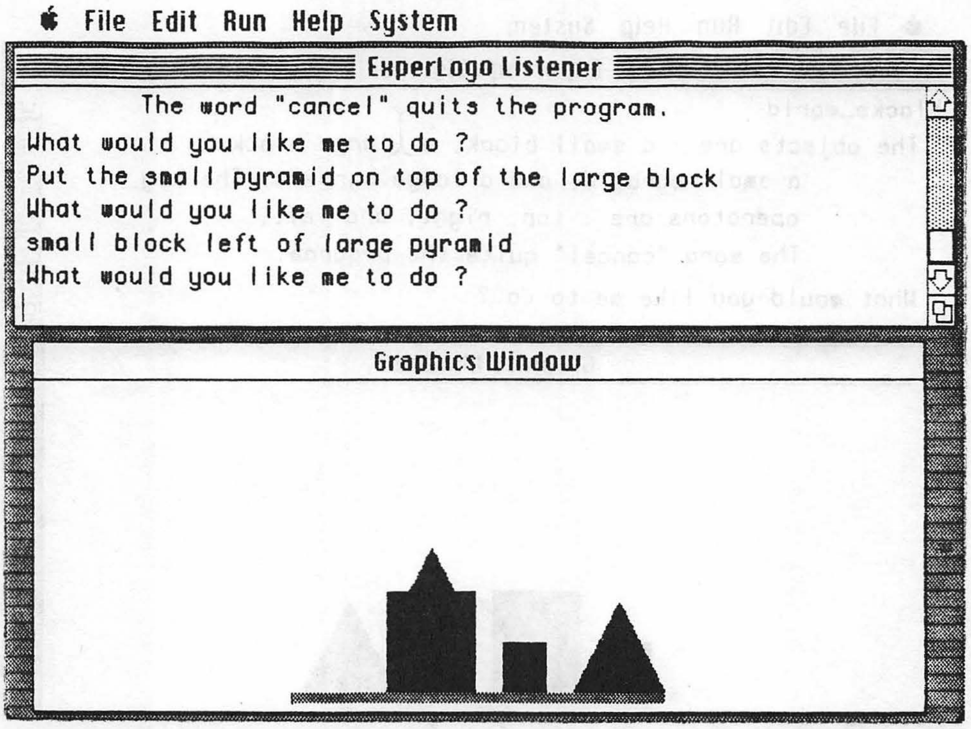
```
blocks_world
 The objects are : a small block, a large block
        a small pyramid, and a large pyramid. The key
        operators are : top, right, and left.
        The word "cancel" quits the program.
 What would you like me to do ?
```

**Figure 5-2. Starting position of Micro Blocks World**



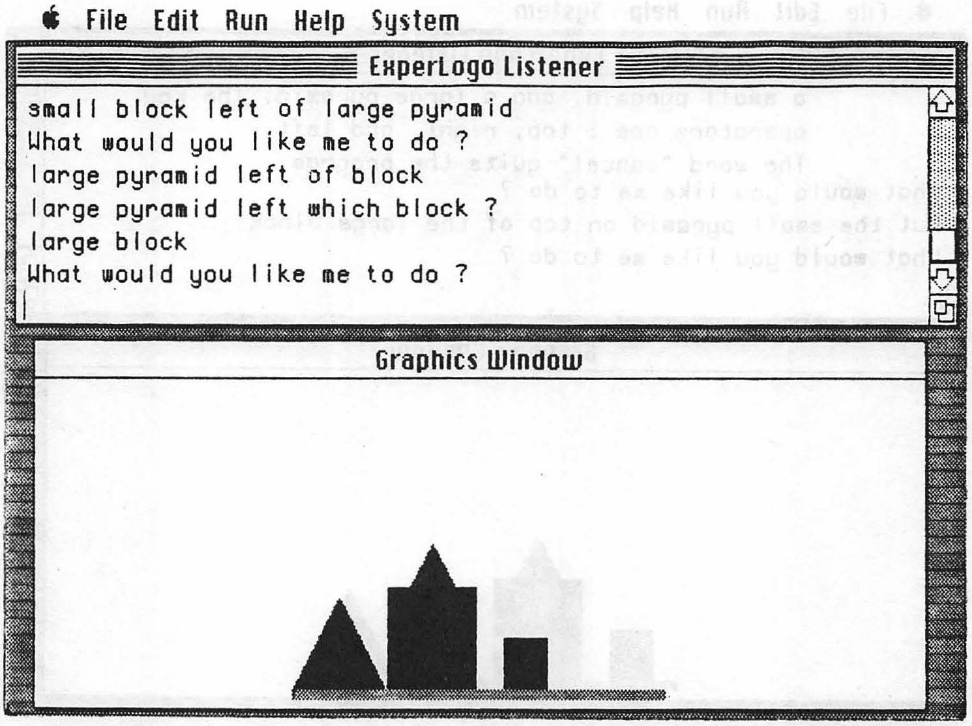
```
        a small pyramid, and a large pyramid. The key
        operators are : top, right, and left.
        The word "cancel" quits the program.
What would you like me to do ?
Put the small pyramid on top of the large block
What would you like me to do ?
```

**Figure 5-3. First command executed**



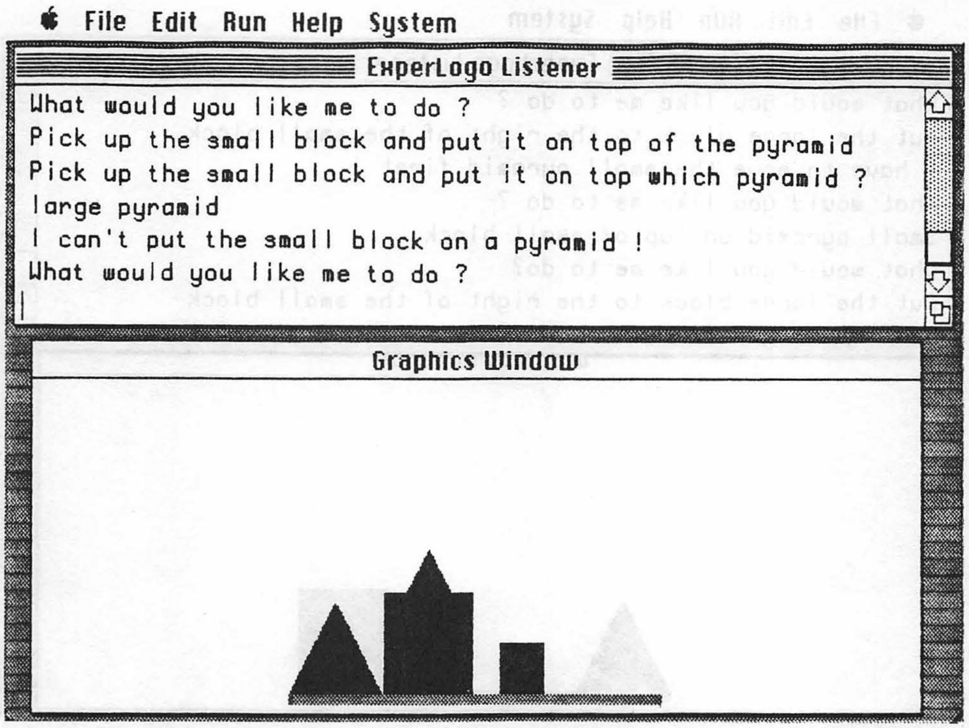
```
        The word "cancel" quits the program.
What would you like me to do ?
Put the small pyramid on top of the large block
What would you like me to do ?
small block left of large pyramid
What would you like me to do ?
```

Figure 5-4. Abbreviated command understood and executed



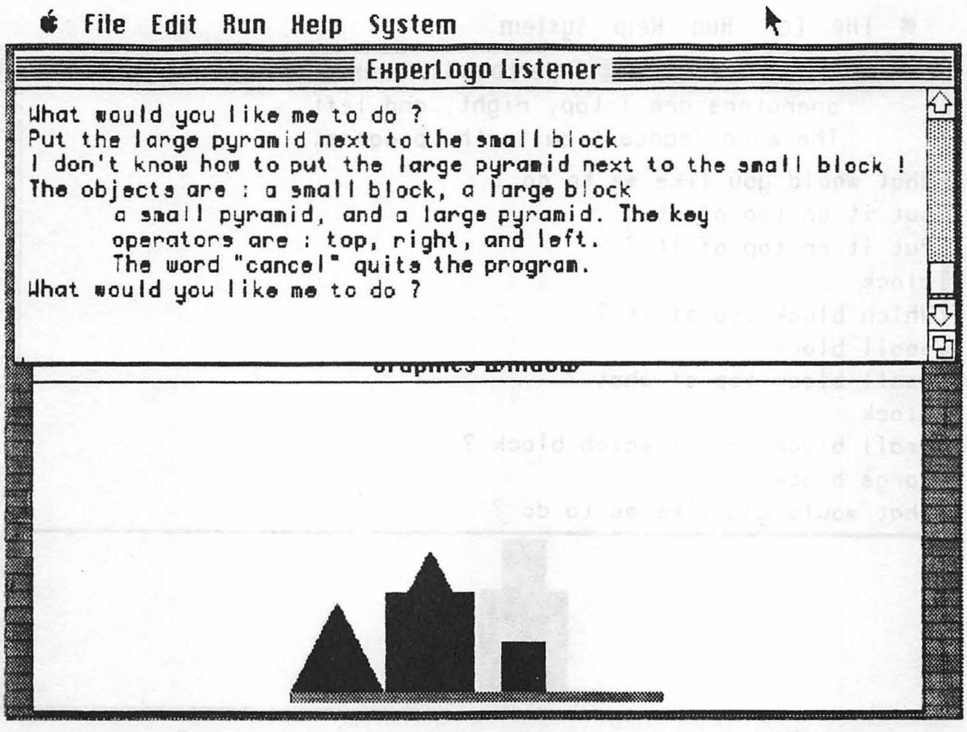
```
small block left of large pyramid
What would you like me to do ?
large pyramid left of block
large pyramid left which block ?
large block
What would you like me to do ?
```

Figure 5-5. Missing information obtained



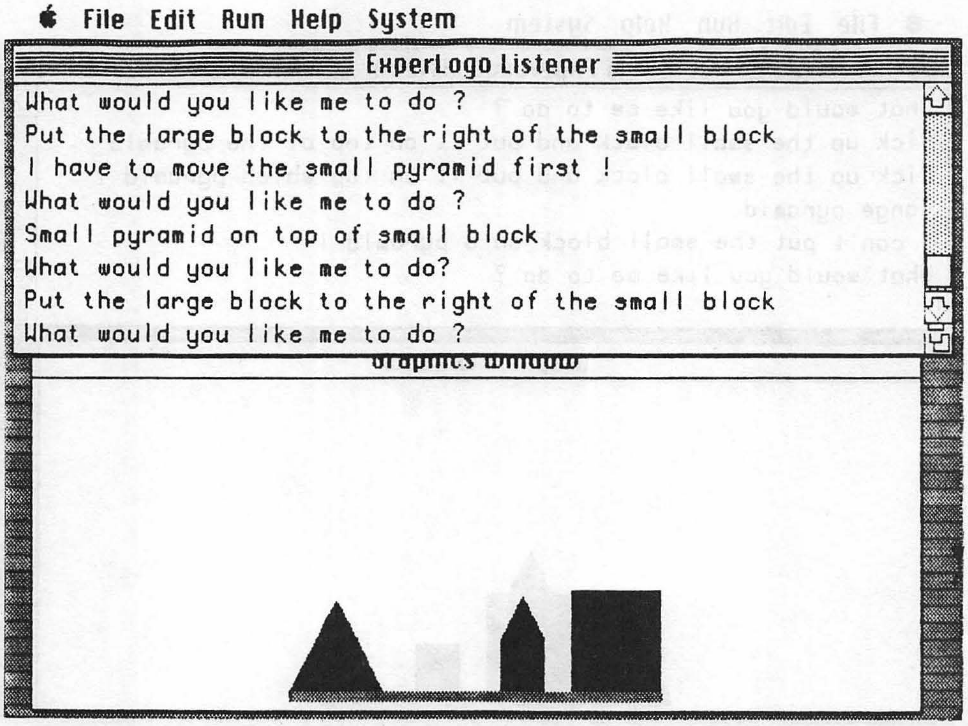
```
What would you like me to do ?
Pick up the small block and put it on top of the pyramid
Pick up the small block and put it on top which pyramid ?
large pyramid
I can't put the small block on a pyramid !
What would you like me to do ?
```

**Figure 5-6. Can't stack things on pyramids!**



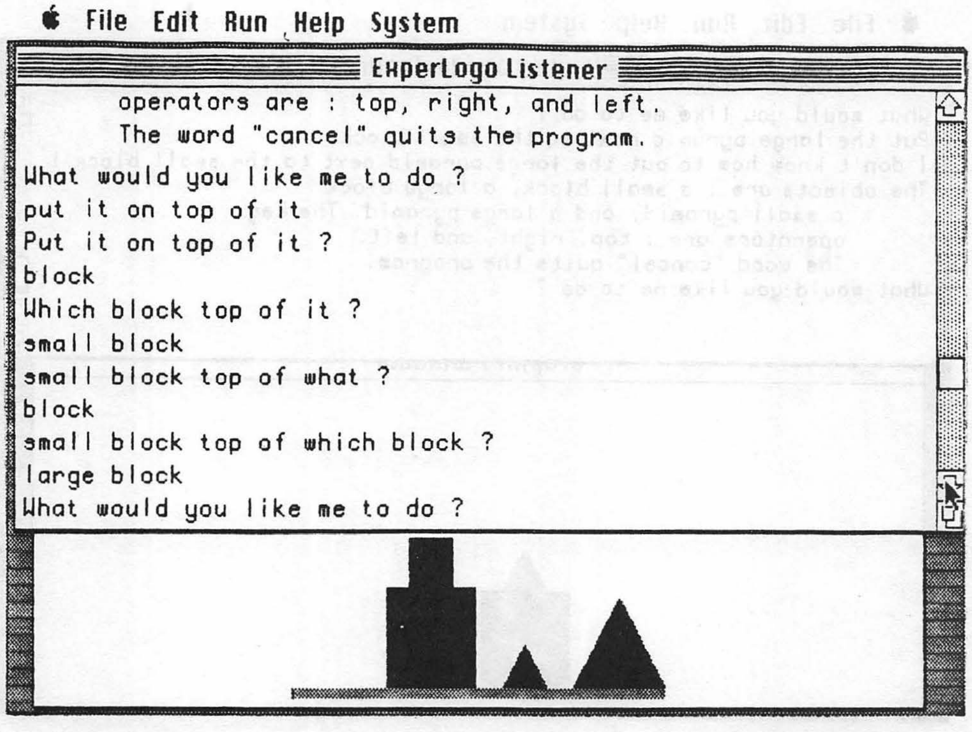
```
What would you like me to do ?
Put the large pyramid next to the small block
I don't know how to put the large pyramid next to the small block !
The objects are : a small block, a large block
        a small pyramid, and a large pyramid. The key
        operators are : top, right, and left.
        The word "cancel" quits the program.
What would you like me to do ?
```

**Figure 5-7. Illegal sentence entered and rejected**



```
What would you like me to do ?
Put the large block to the right of the small block
I have to move the small pyramid first !
What would you like me to do ?
Small pyramid on top of small block
What would you like me to do?
Put the large block to the right of the small block
What would you like me to do ?
```

Figure 5-8. Tried to move an object under another object



```
     operators are : top, right, and left.
     The word "cancel" quits the program.
What would you like me to do ?
put it on top of it
Put it on top of it ?
block
Which block top of it ?
small block
small block top of what ?
block
small block top of which block ?
large block
What would you like me to do ?
```

Figure 5-9. Program plays detective, gives us third degree!

## How Micro Blocks World Works

Figure 5-10 is a box diagram of MICRO_BLOCKS_WORLD. It is divided into two major blocks: INITIALIZE and GET_COMMAND block. The latter, in turn, is broken into two blocks: a single procedure called INITIALIZE_COMMAND and the "workhorse" of the program, PARSE. We will spend most of our time in this section examining the PARSE procedure and some of its key subprocedures.

***The INITIALIZE Procedure*** INITIALIZE has two major functions: it sets up the initial position of the tabletop and objects, and it prints the rules of the blocks world.

The SET_TABLE procedure uses QuickDraw Graphic commands, which are key extensions to the Macintosh® implementations of both ExperLogo® and Microsoft Logo®. (These commands vary from one Logo to another; see Appendix B for more information.)

***The GET_COMMAND Procedure Group*** The GET_COMMAND procedure asks the user, "What would you like me to do?" It then waits for an answer, checks to see that it is not CANCEL, and sends the input to the PARSE procedure.

Before it does this, though, GET_COMMAND calls the INITIALIZE_COMMAND procedure. The procedure clears out information that may have been contained previously in the four key variables, PHASE1, PHASE2, SUBJECT, and OBJECT. PARSE procedures use these variables to analyze the input. Leaving information in them from one command to the next would result in the program not recognizing when information was missing from a command.

***The PARSE Procedure Group*** The process of parsing an input sentence in the program consists of five basic steps. Figure 5-11 depicts these steps and provides a map of the rest of this discussion.

***Dividing the Input into Clauses*** The PARSE procedure parses from left to right the sentence it receives from the GET_COMMAND procedure, seeking an operator: "top," "right," or "left." If it doesn't find one, it takes three steps:

1. It prints, "I don't know how to" followed by a repeat of the input.
2. It displays the rules as a gentle reminder to the user of what it *does* understand.
3. It returns control to GET_COMMAND.

When it does find an operator, it splits the input on both sides of the operator, putting the first part into the variable PHASE1 and the second part into PHASE2.

***Locating and Verifying the Subject*** According to the program's rules, the subject of the sentence always precedes the operator. Therefore, the program searches through the list called PHASE1 using the LOCATE_SUBJECT procedure and, if it can find both, picks out the adjective and noun which make up the subject.

After parsing through PHASE1, the PARSE procedure passes control to the VERIFY_SUBJECT procedure, which checks to be sure that what LOCATE_SUBJECT has found is complete—a valid noun and adjective. If one or both are missing, the procedure asks for the missing information needed until it *is* complete. Control is then returned to the PARSE procedure.

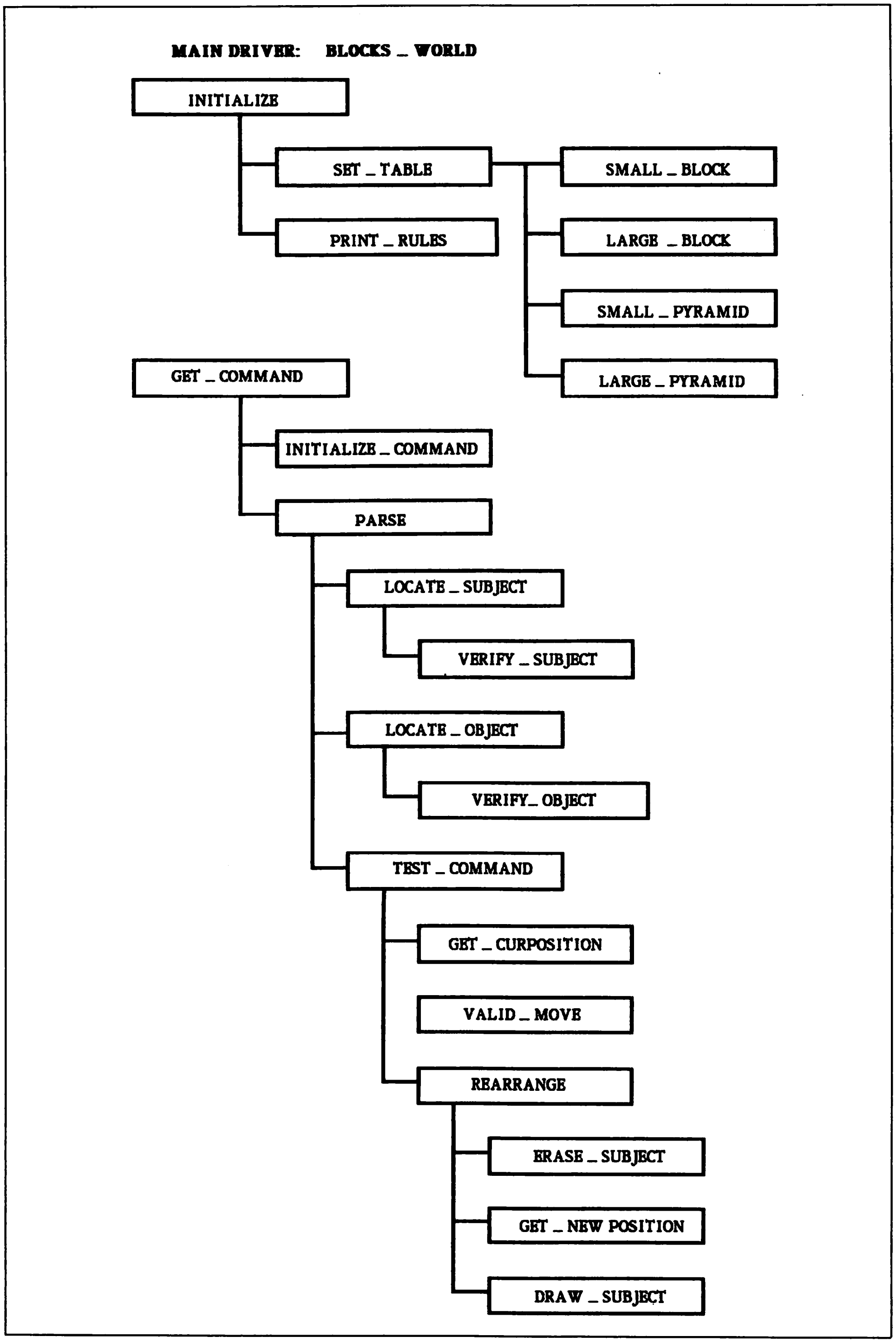


**Figure 5-10. Main driver MICRO__BLOCKS__WORLD**

**Figure 5-11. The parsing process in five steps**

```
            Put the large block on top of the small block

STEP 1:  Operator = top
         PHASE1 = Put the large block on
         PHASE2 = of the small block

STEP 2:  SUBJECT = large block

STEP 3:  OBJECT =  small block

STEP 4:  Is the move valid? (Depends on setup of tabletop
         and on input compared to rules.)

STEP 5:  Rearrange the tabletop to reflect the new
         situation.
```

***Locating and Verifying the Object*** PARSE takes the same steps with PHASE2 that it did with PHASE1 in an effort to find and complete any missing elements in the object of the sentence.

It is interesting to note that the process results in one anomalous situation. If we ask the program to:

**Put the block on the large pyramid**

the program will dutifully ask us for the missing description of the block and *then* inform us that it can't put that block—or any other block, for that matter!—on top of a pyramid. This is because the program deals with the subject first and then the object.

***Testing the Validity of the Move*** Once the program has a complete subject and object and knows what the operator is, it is ready to see if it can legally make the move. The TEST__COMMAND procedure is the main driver routine for this processing.

First, the GET__CURPOSITION procedure examines the subject and object and determines where on the tabletop—or where in relation to one another—they are. It puts the current row and column of the subject in the variables R1 and C1, respectively. Then it puts the current row and column of the *object* in R2 and C2.

The program uses this information in the VALID__MOVE procedure, which informs the program if the requested move is valid or invalid. If invalid, it informs the user of the reason and returns "nil" to the TEST__COMMAND procedure. In that event, the procedure invokes a STOP command and returns control to PARSE.

When the move *is* valid, TEST__COMMAND calls the REARRANGE procedure, which erases the subject from its current position, uses GET__NEW-POSITION to calculate the new position for the item, and stores this information for later use. TEST__COMMAND then calls the procedure DRAW__SUBJECT, which in turn calls on the appropriate graphic command routine to redraw the moved item.

**The Use of Arrays in the Program**

While most of the ExperLogo programs in this book use *lists* as their primary knowledge representation vehicle, Micro Blocks World uses an *array*. Chapter

10 provides a basic introduction to the idea and use of arrays in Logo. If the idea is unfamiliar to you, it might be useful to review that material in trying to understand Micro Blocks World.

The array used here is called POSITION. It keeps track of the graphic coordinate positions of each of the four objects. Four variables, called SB_POS, LB_POS, SP_POS, and LP_POS, point to the locations in this array where information about the small block, large block, small pyramid, and large pyramid, respectively, is stored.

## Exploring AI with Micro Blocks World

We can make a number of improvements to Micro Blocks World without trying to duplicate Winograd's original SHRDLU program. Here are a few suggested changes; you will undoubtedly see others as you work with Micro Blocks World.

### Dealing with Objects under Objects

One change we could make would be to have the program react differently when we ask it to move an object that has an object sitting on it. We could take one of two approaches.

First, the program could move the object along with anything stacked on it. This results in a straightforward programming situation. It will be a little tricky getting all the coordinates of the three stacked objects right, but that should not involve a great deal of trouble.

Second, and more sophisticated, the program could ask the user where to place the object that is on top of the object to be moved. Then it could go through the same process of subject-object checking, rearranging, and so on. Then it could *automatically* carry out the user's first command. This alteration gives the program a stronger appearance of intelligence in its analysis and understanding of our input.

### Coping with Punctuation

As the program is now designed, punctuation at the end of the input sentence causes problems. It would be relatively easy to look for punctuation and strip off any. Define a list called PUNCTUATION to contain all the punctuation marks, scan from *right to left* on the input, and strip them off until you come to a character other than a punctuation mark. Then proceed as the program now runs.

### Adding Objects

A more complex alteration would add graphic elements and/or objects to the universe of the program. We could, for example, add small blocks or large pyramids or both. In fact, we could add more of each type of object. To do so, we'd need a way of differentiating objects from one another. Colors were used in SHRDLU. Our program could use pen patterns (black, gray, light gray, dark gray, and white).

This would necessitate a fairly major change. We would first have to add these new descriptions to the vocabulary. Then we'd have to define new objects to incorporate their patterns. We'd also have to modify PARSE to look for two adjectives and a noun and to be prepared for these adjectives to appear in either color-size or size-color sequence (i.e., "large gray block" is the same thing as "gray large block").

**Expanding its Comprehension**
It would be interesting to fiddle with the program's vocabulary to expand its ability to deal with certain kinds of input. For example, we could add the word "next" as an operator and design the program's response so that it would ask, "next to the large pyramid on which side?" Many such opportunities may occur to you as you run this program and watch your friends use it.

**Adding an Explanation Facility**
Perhaps the most difficult improvement we could consider would enable the program to explain its actions. This would involve giving the program a "memory." This could be done by a series of lists or an array in which we store the last step the program took at the top of a stack of information and let older information drop off the bottom. Then when we ask the program why it did something, it would simply go through the list or array and print the information stored there.

## Summary: What We've Learned about AI

We have taken an in-depth look at one of the most important ideas in NLP programming: parsing input. We have seen what parsing is, strategies available for its implementation, its importance in NLP, and its relationship to other aspects of such programs. We have also taken a look at the concept of "grammars" in the context of a computer program whose goal is to understand human language.

We also examined the capabilities and performance of the famous AI program SHRDLU as preparation for understanding our own new microcomputer version of a subset of that program. We have seen what parsing, grammars, and rules in knowledge bases do when trying to design a program to understanding natural language input.

```
            {Micro Blocks World ©1985, The Waite Group}
                 {Logo program by Ken Schieser}

TO BLOCKS_WORLD
  INITIALIZE
  GET_COMMAND
END

TO INITIALIZE
  MAKE NOUNS [block pyramid]
  MAKE ADJECTIVES [small large]
  MAKE OPERATORS [top right left]
  {Arrays hold an object & graphic position}
  MAKE POSITION MAKE_ARRAY [4 5]
  MAKE (POSITION 3 1) [[small block][-75 70]]
  MAKE (POSITION 3 2) [[large block][ - 25 70]]
  MAKE (POSITION 3 3) [[small pyramid][25 70]]
  MAKE (POSITION 3 4) [[large pyramid][75 70]]
  {SB short for small block etc. Variables hold array positions [column row]}
  MAKE SB_POS [3 1]
  MAKE LB_POS [3 2]
  MAKE SP_POS [3 3]
  MAKE LP_POS [3 4]
  SET_TABLE
  PRINT_RULES
END

TO SET_TABLE
  PENPAT :GRAY
  PAINTRECT [70 - 100 75 100]
  PENPAT :BLACK
  SMALL_BLOCK - 75 70
  LARGE_BLOCK - 25 70
  SMALL_PYRAMID 25 70
  LARGE_PYRAMID 75 70
  (LISTENER 'SELECTWINDOW)
END

TO PRINT_RULES
  PR<<The objects are :a small block, a large block, a small pyramid, and a large
pyramid. The key operators are: top, right, and left. The word "cancel" quits the
program.>>
END

TO GET_COMMAND
  INITIALIZE_COMMAND
  PR<<What would you like me to do?>>
  MAKE COMMAND READLIST
  IF :COMMAND = [CANCEL] [STOP]
  PARSE
  GET_COMMAND
END

TO INITIALIZE_COMMAND
  MAKE PHRASE1[]
  MAKE PHRASE2[]
  MAKE SUBJECT[]
  MAKE OBJECT[]
END

{Procedure Parse: Find an operator (top right or left) splits the sentence into 2 phrases
(phrase1- first part before the operator, phrase2- second part after operator}
TO PARSE
  IF EMPTYP :COMMAND [PR (SE<< I don't know how to>> :PHRASE1 <<!>>)
```

```
    PRINT_RULES STOP]
  MAKE WORD FIRST :COMMAND
  IF MEMBERP :WORD :OPERATORS
    [MAKE OPERATOR :WORD
    MAKE PHRASE2 BF :COMMAND
{subject is located in phrase1}
    LOCATE_SUBJECT :PHRASE1
{object is located in phrase2}
    LOCATE_OBJECT :PHRASE2
{test_command is the call to the graphics}
    TEST_COMMAND STOP]
  MAKE PHRASE1 LPUT :WORD :PHRASE1
  MAKE COMMAND BF :COMMAND
  PARSE
END

  TO LOCATE_SUBJECT :PHRASE
  IF EMPTYP :PHRASE [VERIFY_SUBJECT STOP]
  MAKE WORD FIRST :PHRASE
  IF OR MEMBERP :WORD :NOUNS MEMBERP :WORD :ADJECTIVES
    [MAKE SUBJECT LPUT :WORD :SUBJECT]
  LOCATE_SUBJECT BF :PHRASE
END
{Procedure verify_subject verifies that the variable subject contains one adjective & one
noun.
The user is given the option to enter the correct data. Same for verify_object}
TO VERIFY_SUBJECT
  IF OR EMPTYP :SUBJECT (COUNT :SUBJECT) >2
    [PR (SE <<Put what>> :OPERATOR :PHRASE 2 <<?>>)
    MAKE SUBJECT[]
    MAKE PHRASE1 READLIST
    LOCATE_SUBJECT :PHRASE1 STOP]
IF NOT MEMBERP FIRST :SUBJECT :ADJECTIVES
    [PR (SE <<Which>> :SUBJECT :OPERATOR :PHRASE2: <<?>>
    MAKE SUBJECT[]
    MAKE PHRASE1 READLIST
    LOCATE_SUBJECT :PHRASE1 STOP]
IF NOT MEMBERP LAST :SUBJECT :NOUNS
  [PR (SE <<The>> :SUBJECT <<what>> :OPERATOR :PHRASE2 <<?>>
    MAKE SUBJECT[]
    MAKE PHRASE1 READLIST
    LOCATE_SUBJECT :PHRASE1 STOP]
END

TO LOCATE_OBJECT :PHRASE
  IF EMPTYP :PHRASE [VERIFY_OBJECT STOP]
  MAKE WORD FIRST :PHRASE
  IF OR MEMBERP :WORD :NOUNS MEMBERP :WORD :ADJECTIVES
    [MAKE OBJECT LPUT :WORD :OBJECT]
  LOCATE_OBJECT BF :PHRASE
END

TO VERIFY_OBJECT
  IF OR EMPTYP :OBJECT (COUNT :OBJECT) >2
    [PR (SE :PHRASE1 :OPERATOR <<of what?>>)
    MAKE OBJECT []
    MAKE PHRASE2 READLIST
    LOCATE_OBJECT :PHRASE2 STOP]
  IF NOT MEMBERP FIRST :OBJECT :ADJECTIVES
    [PR (SE :PHRASE1 :OPERATOR <<which>> :OBJECT <<?>>)
  MAKE OBJECT[]
  MAKE PHRASE2 READLIST
  LOCATE_OBJECT :PHRASE2 STOP]
IF NOT MEMBERP LAST :OBJECT :NOUNS
  [PR (SE :PHRASE1 :OPERATOR <<the>> :OBJECT <<what?>>)
  MAKE OBJECT[]
  MAKE PHRASE2 READLIST
  LOCATE_OBJECT :PHRASE2 STOP]
END

TO TEST_COMMAND
  GET_CURPOSITION
  IF NOT VALID_MOVE [STOP]
  REARRANGE
END

{Procedure get_curposition obtains the current array position of the subject [C1 R1] &
the object [C2 R2]}
TO GET_CURPOSITION
  IF :SUBJECT = [small block]
    [MAKE C1 FIRST :SB_POS MAKE R1 LAST :SB_POS]
  IF :SUBJECT = [large block]
    [MAKE C1 FIRST :LB_POS MAKE R1 LAST :LB_POS]
  IF :SUBJECT = [small pyramid]
    [MAKE C1 FIRST :SP_POS MAKE R1 LAST :SP_POS]
  IF :SUBJECT = [large pyramid]
    [MAKE C1 FIRST :LP_POS MAKE R1 LAST :LP_POS]
  IF :OBJECT = [small block]
    [MAKE C2 FIRST :SB_POS MAKE R2 LAST :SB_POS]
  IF :OBJECT = [large block]
    [MAKE C2 FIRST :LB_POS MAKE R2 LAST :LB_POS]
  IF :OBJECT = [small pyramid]
    [MAKE C2 FIRST :SP_POS MAKE R2 LAST :SP_POS]
  IF :OBJECT = [large pyramid]
    [MAKE C2 FIRST :LP_POS MAKE R2 LAST :LP_POS]
END

TO VALID_MOVE
{If subject & object are the same}
```

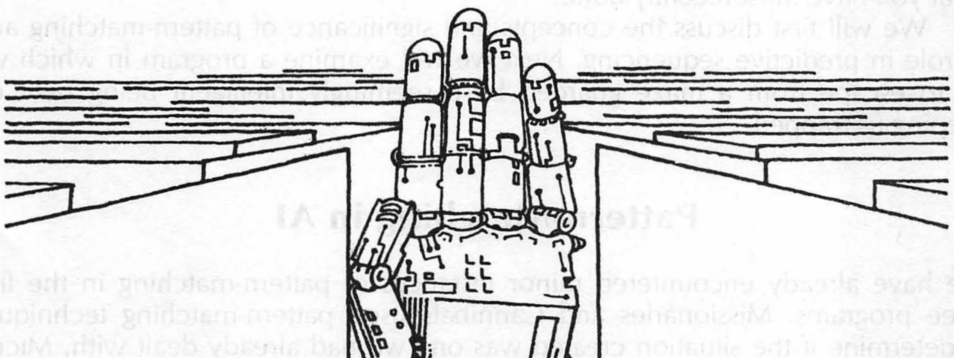
```
  IF AND :C1 = :C2 :R1 = :R2
    [PR (SE <<I can't>> :PHRASE1 :OPERATOR :PHRASE2 <<!>>)
  OP NIL]
{if trying to place on a pyramid}
  IF AND :OPERATOR = 'top (LAST :OBJECT) = 'pyramid
    [PR (SE <<I can't put the>> :SUBJECT << on a pyramid!>>) OP NIL]
{if trying to place out of array bounds (off the table)}
  IF OR (AND :OPERATOR = 'right :R2 = 4)
    (AND :OPERATOR = 'left :R2 = 1)
  [PR (SE <<I can't put the>> :SUBJECT <<off the table!>>)
  OP NIL]
{if trying to place next to an object that is on top of another}
  IF AND (OR :OPERATOR = 'right :OPERATOR = 'left) :C2<3
    [PR (SE <<I can't place the>> :SUBJECT <<to the right or the left since the>>
    :OBJECT <<is not sitting on the table.>>) OP NIL]
{if the subject is pinned down by another object}
  IF NOT EMPTYP (POSITION :C1 - 1 :R1)
    [PR (SE <<I have to move the>> FIRST (POSITION :C1 - 1 :R1) <<first!>>) OP
    NIL]
{NEXT 3 - if another object currently holds the position}
  IF AND :OPERATOR = 'top NOT EMPTYP (POSITION :C2 - 1 :R2)
    [PR (SE <<The>> FIRST (POSITION :C2 - 1 :R2) <<is on top of the >>
    :OBJECT <<!>>) OP NIL]
  IF AND :OPERATOR = 'right NOT EMPTYP (POSITION :C2 :R2 + 1)
  [PR (SE <<The>> FIRST (POSITION :C2 :R2 + 1) <<is to the right of the >>
    :OBJECT <<!>> OP NIL]
  IF AND :OPERATOR = 'left NOT EMPTYP (POSITION :C2 :R2 - 1)
  [PR (SE <<The>>FIRST (POSITION :C2 :R2 - 1) <<is to the left of the>>
    :OBJECT <<!>> OP NIL]
  OP 'TRUE
  END

TO REARRANGE
  ERASE_SUBJECT
  GET_NEWPOSITION
  DRAW_SUBJECT
END

{Procedure get_newposition: calculates coordinates & array position, redefines array &
array position variables}
TO GET_NEWPOSITION
  MAKE X FIRST LAST (POSITION :C2 :R2)
  MAKE Y LAST LAST (POSITION :C2 :R2)
  IF :OPERATOR = 'TOP
    [IF :OBJECT = [large block]
      [MAKE HEIGHT 48][MAKE HEIGHT 24 ]
    MAKE :C1 :C2 - 1
    MAKE :R1 :R2
    MAKE Y :Y - :HEIGHT]
  IF :OPERATOR = 'right
    [MAKE C1 :C2
    MAKE R1 :R2 + 1
    MAKE X :X + 50]
  IF :OPERATOR  = 'left
    [MAKE C1 :C2
    MAKE R1 :R2 - 1
    MAKE X :X - 50]
    MAKE (POSITION :C1 :R1) LIST :SUBJECT SE :X :Y
  IF :SUBJECT  = [small block][MAKE SB_POS SE :C1 :R1]
  IF :SUBJECT  = [large block][MAKE LB_POS SE :C1 :R1]
  IF :SUBJECT  = [small pyramid][MAKE SP_POS SE :C1 :R1]
  IF :SUBJECT  = [large pyramid][MAKE LP_POS SE :C1 :R1]
END

TO DRAW_SUBJECT
  MAKE X FIRST LAST (POSITION :C1 :R1)
  MAKE Y LAST LAST (POSITION :C1 :R1)
  IF :SUBJECT  = [small block][SMALL_BLOCK :X :Y]
  IF :SUBJECT  = [large block][LARGE_BLOCK :X :Y]
  IF :SUBJECT  = [small pyramid][SMALL_PYRAMID :X :Y]
  IF :SUBJECT  = [large pyramid][LARGE_PYRAMID :X :Y]
END

TO ERASE_SUBJECT
  PENPAT :WHITE
  DRAW_SUBJECT
  PENPAT :BLACK
  MAKE (POSITION :C1 :R1) NIL
END

TO SMALL_PYRAMID :X :Y
  PU SETPOS SE :X - 12 :Y
  MAKE TRI OPENPOLY
  SHOWPEN
  LEFT 90 REPEAT 3[RT 120 FD 24] RT 90
  CLOSEPOLY
  PAINTPOLY :TRI
  KILLPOLY :TRI
END

TO LARGE_PYRAMID :X :Y
  PU SETPOS SE :X - 24 :Y - 1 PD
  MAKE TRI OPENPOLY
  SHOWPEN
  LT 90 REPEAT 3[RT 120 FD 48] RT 90
  CLOSEPOLY
  PAINTPOLY :TRI
  KILLPOLY :TRI
END
```

```
TO SMALL__BLOCK :X :Y
  PAINTRECT (SE :Y - 24 :X - 12 :Y :X + 12)
END

TO LARGE__BLOCK :X :Y
  PAINTRECT (SE :Y - 48 :X - 24 :Y :X + 24)
END
```

CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL IN
INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
L INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
L INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELL

6

# Intelligent Maze Game



○ **Pattern-Matching Concepts**
○ **Predictive Sequencing**
○ **Uncertainty, Randomness, and Unpredictability**
○ **Three Main Types of Pattern-Matching**

In the early 1980s when video games were the rage, some programs were notorious for the seeming intelligence of the "enemy" against whom one "fought." It seemed that no matter how hard one tried, the enemy was relentless and seemingly unbeatable. The persistent little opponents heatedly and smartly pursued the "victim." They even anticipated one's actions. The ghosts pursuing the little hero always "knew" how to cut the hero off, unless the player was particularly quick-witted.

"How does he know that's where I'm going?" more than one human player might have wondered. How, indeed?

Predicting behavior can be done in a number of ways. The one with which this chapter will concern itself is predicting sequencing through pattern-matching. That means the program anticipates what you are going to do next based on what you have most recently done.

We will first discuss the concepts and significance of pattern-matching and its role in predictive sequencing. Next we will examine a program in which we try to escape from a maze guarded by a seemingly intelligent being bent on keeping us trapped.

## Pattern-Matching in AI

We have already encountered minor instances of pattern-matching in the first three programs. Missionaries and Cannibals used pattern-matching techniques to determine if the situation created was one we had already dealt with. Micro-Logician could compare an input string to a stored pattern or template to determine if the program could deal with the sentence and, if so, how. Poetry Maker used patterns to establish the sequence of parts of speech in order to randomly generate poetry so the program would seem sensible.

### An Important Concept

Pattern-matching is an important concept in AI programming—one which has been applied to a number of fields of investigation. For example, it plays a key role in most NLP programming. Matching sentences and clauses against stored patterns as they are parsed enables programs to differentiate one kind of grammatical input from another.

Robotic vision is another area that has seen much emphasis on pattern recognition and matching. A robot "eye"—a camera or cameralike apparatus—gathers information about images in its field of view. It compares edges, sizes, and even shadows with stored patterns to determine the kind of object it is dealing with.

### Humans and Patterns

It has often been said, with some exaggeration, that "people are creatures of habit." We may not be entirely creatures of habit but we are certainly creatures of pattern. When faced with a frequent trip between home and a store, we will probably follow the same route almost every time. We may eat meals in a certain pattern, we probably always get dressed in a certain sequence. (The difference between habit and pattern is perhaps only one of degree. A "habit" is an unconscious but repetitively performed set of behaviors, while a "pattern" is performed consciously.)

Much of our success coping with seemingly trivial things in daily life can be attributed to patterns and predictability. A traffic light which has just turned amber is certain to turn red; faced with a red light, people will almost always stop. A delicious dinner served by Aunt Mabel will almost surely be followed by something chocolate and fattening.

***Intelligence Is Seeing Patterns*** Analogy problems are perhaps the most frequently posed problems on tests designed to measure either intelligence or ability to learn new concepts. You've almost certainly encountered such problems as that shown in Figure 6-1. The problem is stated algebraically as: "A is to B as C is to what?"

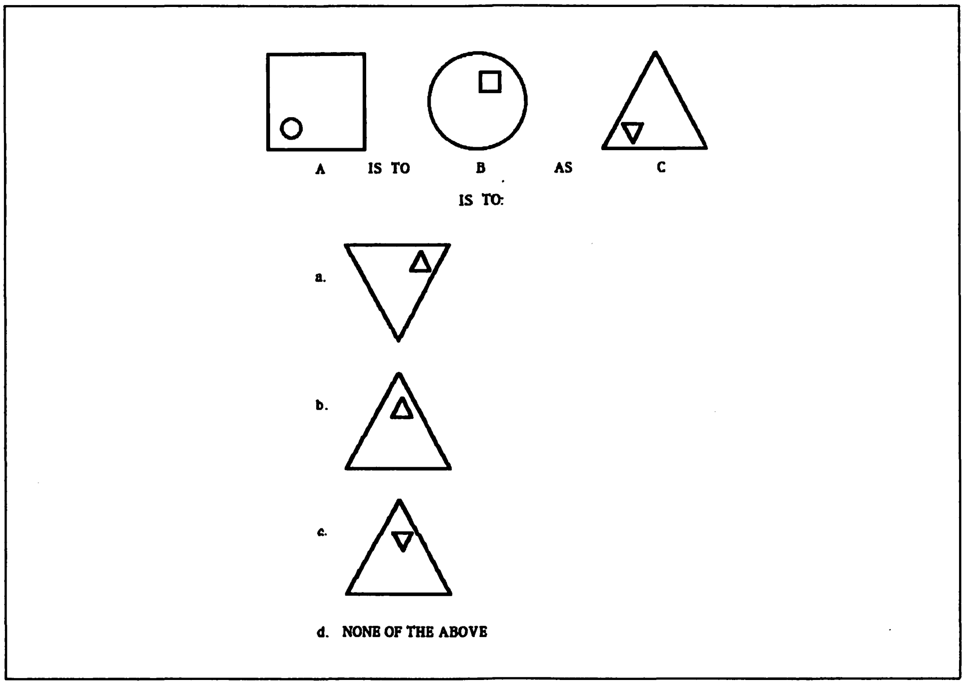


**Figure 6-1. Analogy problems test intelligence**

How we go about solving such problems has fascinated AI researchers almost from the beginning. Problem-solving is an area of AI research which has borne important fruit in the past few years. Without going into psychological or educational theories, it is clear that we all follow roughly the same sequence of steps.

1. We look back and forth between A and B and try to find out what they have in common and what is different between them.
2. We try to formulate a rule about what is different between A and B.
3. Then we look at C and try to see how *it* differs from and is similar to A.
4. We try to formulate a rule about what is different between C and A.
5. We examine the optional answers in turn to see if one predicts the logical outcome of applying the A-B rule to the figure called C.

Quite often, similarities between A and C may be minimal or nonexistent, at least on the surface. The less obvious such similarities are, the more difficult the problem will be to solve.

The understanding of vocabulary also is often tested by analogic reasoning. The test provides three words along with instructions to find the word in the answer list that is related to the third word as the first two are related to one another. For example:

Empty is to full as depressed is to:

a. joyful
b. tired
c. sad
d. content

We observe that the first two words are direct opposites, so we look for the direct opposite to the word "depressed" in the list and find "joyful." Note that "content" comes close but seems to fall short of being a *direct* opposite.

The point is simple—a provable relationship exists between the ability to discern patterns in things and the ability to learn new ideas and incorporate new concepts into our thinking. Analogic reasoning is not in itself intelligence, but it certainly plays a major role in measuring intelligence.

**Predictive Sequencing: Basic Ideas**

So far the analogy problems we've looked at involve very small two-step sequences. From the A-B sequence of two steps we try to analogize a rule which will solve the C-? two-step sequence. In human behavior—and in other aspects of intelligence testing as well—sequences are often much longer than two steps.

To predict how an individual will react to a specific situation, we would increase the probability of accuracy as we recorded more and more observations of his behavior in that situation. We see a man arrive at a restaurant at 7:33 A.M. on a weekday. He goes to the newspaper rack outside, buys a paper, carries it inside the restaurant, and reads it while he has breakfast.

We predict, based on this single observation, that at 7:33 A.M. the following day, the same man will repeat the sequence. If he doesn't, we have no pattern on which to base another prediction. If he does show up and follow the same steps, we have a much higher degree of confidence that the man will do the same thing the next day, and so on.

What if, on the second day we are watching our man, he does indeed show up at 7:33 A.M., but this time has the newspaper under his arm when he arrives. For the next few days, he repeats the same pattern. We now might feel safe in predicting that his behavior pattern, broken only occasionally, is to bring his newspaper to the restaurant, arriving 7:33 A.M., and eat breakfast while reading his paper. We have altered our prediction about the sequence of events based upon our observation.

To predict the outcome of sequences, then, we have to have enough observations, or examples, to work with to think that our prediction will be right. The more instances we have, the more likely we are to be correct in a prediction.

**Number Sequences**

Anyone who has taken an IQ test or an achievement test in the past few years will recognize this type of problem:

> Find the next number in the sequence:
> 32, 24, 18, __
>
> a. 0
> b. 27/2
> c. 6
> d. 14

Faced with this kind of problem, we apply slightly different logic than we did to our earlier analogies problems. Here, we try to find a pattern among the three numbers given to us and then predict, based on that pattern, what the next value in the sequence will be. In other words, we *assume* the sequence will continue and that the pattern, whatever it is, will repeat itself. We don't merely look through the answers to see which one might fit (unless we reach the point where we simply can't see the pattern and are looking for some clues in the answers).

(The answer, by the way, is "b." Each number is 3/4 of its predecessor.)

## When Prediction Is Less Certain

Most prediction, whether involving predictive sequencing or some other technique, is far less precise than prediction involving analogies and number sequences where strict rules apply and one answer *must* be right, even if the answer is the infamous "none of the above." Weather forecasting, particularly by laypeople, provides a good example of the kind of prediction we are likely to engage in in daily life—and which, therefore, provides a better model for AI programming than does an intelligence test.

**One Hundred Heads in a Row. . .**

The traditional logic problem about coin-tossing provides an example of the truth of the difficulty of real-life prediction. The question is usually framed like this: "If I toss a coin in the air 100 times and it comes down heads every time, what are the chances that the 101st toss will also be heads?" The answer, of course, is 50–50, the same as on every toss of a coin. Every toss of the coin is a completely independent event; previous results do not affect the next toss.

When events are either truly random or disconnected from one another, we cannot find a pattern that can predict what will happen in a given event. But we can often bring some sense of order and sequence and consequent predictability to a seemingly random situation by examining more information.

There is an important idea in pattern-matching in AI programs; the more factors available to compose the pattern, the more likely we are to find some pattern fit in a knowledge base and thus be able to use the information for prediction.

## Three Types of Pattern-Matching

Pattern-matching doesn't lend itself well to categorization. It may be helpful, however, to think of pattern-matching as having three varieties: template-matching, macro-matching, and micro-matching (Drs. Cherniak, Riesbeck, and McDermott identified the last two categories).

**Template-Matching**

At its simplest level, "template-matching" compares one thing to another to see if sufficient similarities exist to declare them to be of the same pattern. An example of this kind of simplistic pattern-matching occurs in Micro-Logician (Chapter 3) when we compare input against several stored prototypes of what sentences should look like to the program.

Early NLP programs—ELIZA, PARRY, and others—used relatively straightforward pattern-matching of this variety to achieve some seemingly astonishing results.

Another level of template-matching involves comparing two pieces of information to see if they intend to represent the same thing. For example, the following two sentences:

John gave Mary the ball.
Mary got the ball from John.

should come out as "equal" in any pattern-matching, which would be useful in NLP programming. The level of sophistication is obviously one degree greater than that of the simple template-matching we've seen earlier.

**Micro-Matching**

Quite often AI programming has two "small" data or knowledge structures, each of which contains variable information (i.e., unknowns). Our objective is to "squeeze" the two together to see if they will fill in some or all of each other's missing data. This important part of the AI programming environment is referred to as "micro-matching."

We will view this kind of pattern-matching in some detail in Chapter 7 when we examine a small language built in Logo to combine two such inputs to produce an output containing less unknown information than either original.

**Macro-Matching**

Sometimes AI programs need to take two large structures—entire knowledge bases, for example—and compare them to identify similarities and differences. A classic example of this kind of matching is the General Problem-Solving (GPS) program (written by G. Ernst and A. Newell in 1969 in an effort to devise strategies for problem-solving that would work on a broad range of problem types).

The GPS program essentially compared two sets of knowledge, looked for differences and then tried to eliminate them. As the number of differences was reduced, the data bases become more and more similar and so more closely approached a solution.

Except for some very theoretical exploration, little work has been done in macro-matching because such programs must be large and are very expensive to create and maintain.

The pattern-matching going on in a macro-matching program goes beyond template-matching. It is micro-matching without variables. In the following program, the goal is simply to look for any pattern whatsoever—sequences of moves that seem to repeat—to help predict with some confidence the next move the player is liable to make.

## The Intelligent Maze Program

The program's execution is quite simple, and its structure and syntax are refreshingly straightforward. Best of all, it provides a clear and convincing demonstration of pattern-matching at work.

### What It Does

The "Intelligent Maze" program pits the user against the program's predictive pattern-matching algorithms. The program's objective is to keep the user from successfully moving from the bottom of the maze to the exit at the top. It does this by blocking the pathway in the direction the program thinks the person is likely to try to move next. Figure 6-2 shows what the opening screen looks like as you begin to play the game.

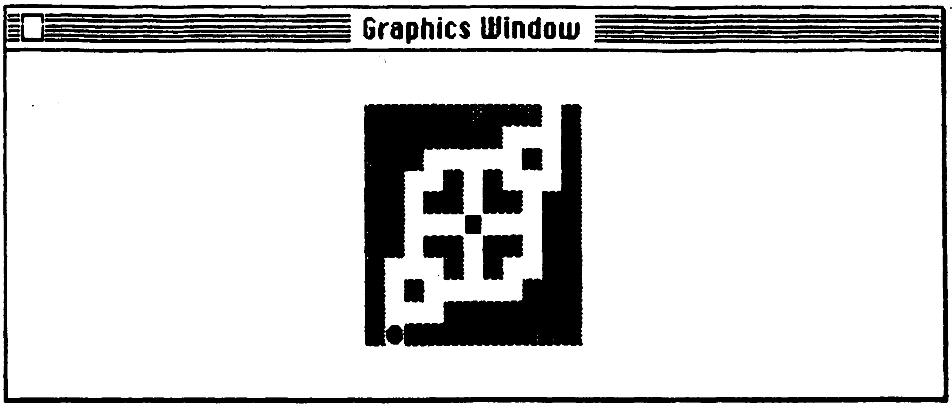


**Figure 6-2. Opening screen of INTELLIGENT_MAZE**

Users, represented by the circle, click the mouse button anywhere in the Graphic Window. To move left, for example, they click anywhere along the row to the left of where the marker is positioned. The program interprets that click as "move to the left one position." By holding the mouse button down, the player can move as far to the left as the maze and the enemy, the square, will allow. Similarly, upward movements, downward movements, and shifts to the right are handled by clicking and/or holding the mouse button down with the pointer positioned on the side of the maze where users wish to move.

We could have designed this program so that the enemy simply moved to the exit and blocked it at the beginning of the game. So much for intelligence! But that would have defeated our purpose, besides creating a very boring program.

We decided instead to make the "enemy" powerful but not invincible. He can move through walls and traverses the maze quite rapidly. But he only moves when he thinks he's found a pattern in the user's movements *and* the user is in a position to move in at least two directions. This means the player will quite often move without any response from the computer.

**Winning the Game**

It is quite difficult for the player to emerge from the maze. Once you reach the upper right corner of the maze area, your opponent moves to block the exit and you are stuck. (Figure 6-3 shows this situation.) But at that point, you must become "sneaky." You have to convince the enemy, based on your pattern of movement, that you are heading in a different direction. But you can't get so far away from the exit that when the enemy finally moves to block your path, you can't exit in a move or two, before he discerns the pattern of movement. You *can* win (for proof, see Figure 6-4).

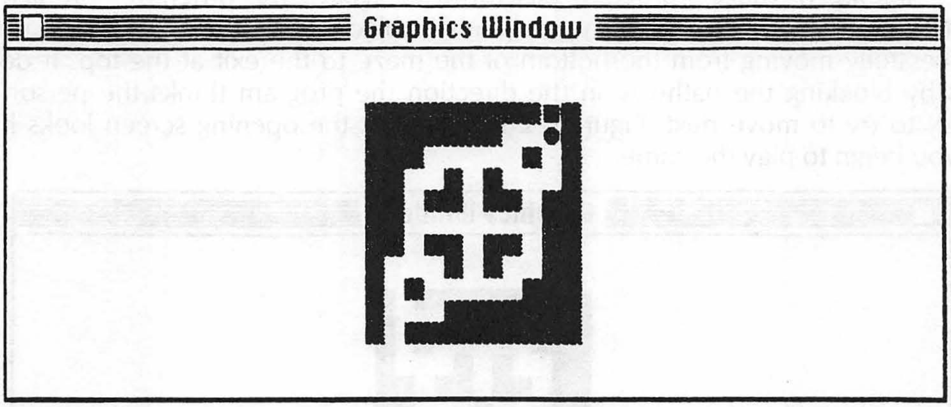


**Figure 6-3. INTELLIGENT__MAZE game at impasse near end**

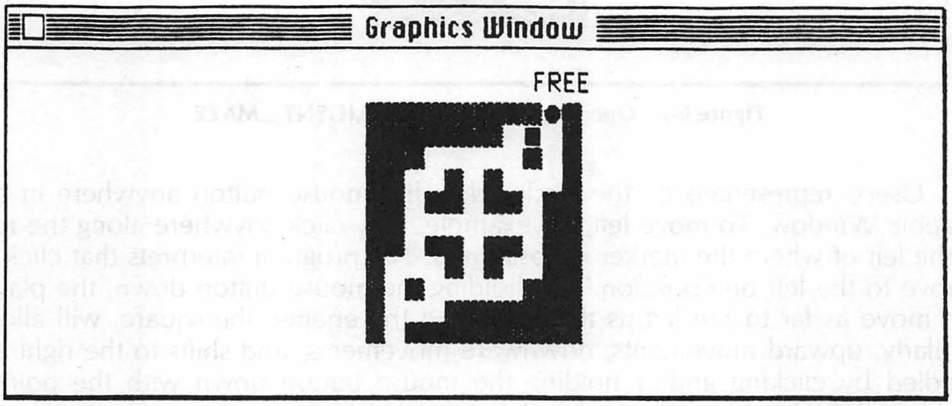


**Figure 6-4. Success in INTELLIGENT__MAZE**

Don't get discouraged if you don't win this game right away. Learn how it works first. Remember, the idea isn't to create a game you can win or to create one you *can't* win. The idea, rather, is to help you learn something about AI.

## How It Works

Figure 6-5 is a box diagram of the INTELLIGENT__MAZE program. The main driver routine, INTELLIGENT__MAZE, calls two other main procedures. SET-UP__GAME initializes the information needed to keep track of what is going on in the maze (more in a moment). It then draws the three graphic objects: the maze itself, the circle representing the player, and the square representing the computer enemy.

We are ready for the PLAY procedure, where all of the action in the program takes place.

Before we look at several of the procedures that make up PLAY, we should understand how the layout of the collection of rows composing the maze is represented.

**INITIALIZE__ROWS Procedure**

You can see from Figure 6-6 that the maze layout is composed of nine rows, not counting the entrance and exit rows. The rows are numbered from top to bottom.

Figure 6-6 shows how Row 1 is represented in the data structure in the program. The variable ROW1 is a list of lists, one list for each space in the row where the circle could land. Each list consists of two more lists. The first list provides coordinates of the position (with the center of the maze at 0,0, or "origin" point) and the second list contains all the moves legal from that position.

Thus, if we look at the leftmost open square in Row 1 of Figure 6-6, we find that it has coordinates [20, −40] and the legal moves that can be made from it are [D, R], or down and right. The second square is located at position [30, −40] and a circle placed there can move left or right. The third and final square on this row is at coordinates [40, −40]. Legal moves are up, down, or left, but not right.

Each row of the maze is set up the same way, so that once we have positioned the circle, we only need to determine its coordinates, match them with the coordinates possible for that row, and we know what possible moves the circle can make.

## PLAY Procedure Group

As you can see from Figure 6-5, the PLAY procedure contains three main procedure groups and a fourth, stand-alone procedure. The procedure groups are called GET__DIRECTION, MOVE__CIRCLE, and BLOCK. The stand-alone FREE procedure checks to see if the circle's vertical coordinate has exceeded −50, in which case it has escaped from the top of the maze and the game is over.

**GET__DIRECTION Procedure Group**

This small collection of procedures determines the direction the player is trying to move and, if it is a legal move, updates the list called LAST__MOVES. Based on the contents of this list, the program predicts the player's next move.

The third line of the GET__DIRECTION procedure is the key to this group. The line checks to see if *either* the direction the player is trying to move is not one of the legal moves for that square *or* the player is blocked in that direction

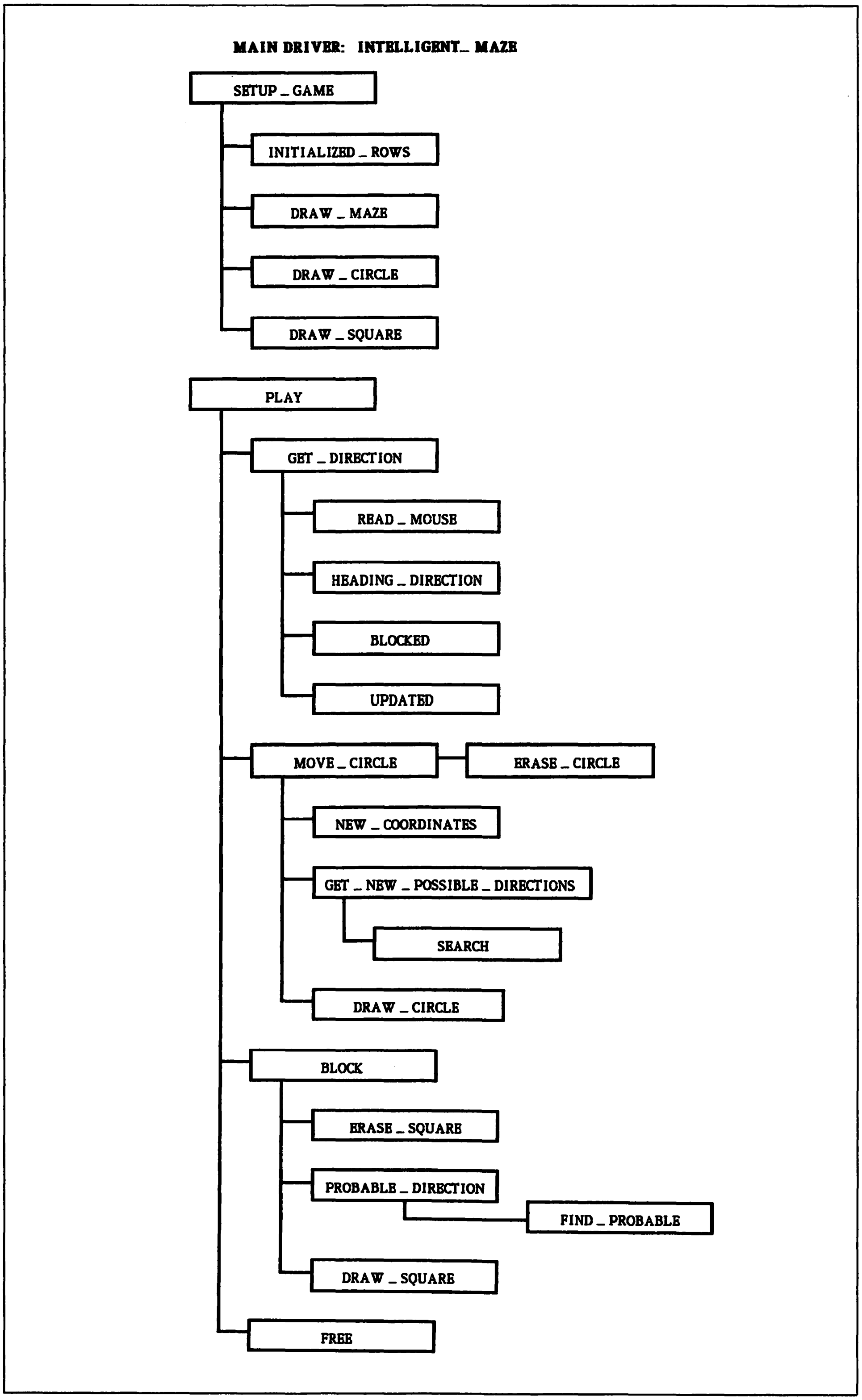
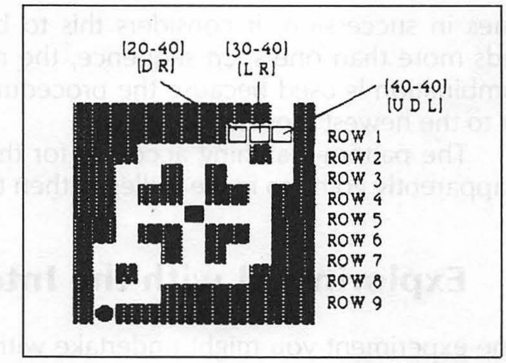


Figure 6-5. Main driver: INTELLIGENT__MAZE

**Figure 6-6. Row organization of INTELLIGENT__MAZE**



by the enemy. If either is true, the procedure simply calls itself and waits for the user to make a *legal* move.

If neither condition is true—in other words, if the move is legal and not blocked—the procedure updates the LAST__MOVES list and returns control to PLAY.

**MOVE__CIRCLE Procedure Group**

This group of procedures performs two major functions. It actually relocates the circle for the player from its present position to the place the player has asked it to move. Second, it updates the list of legal directions the player can now move, based on the contents of the row list representing the row on which the circle is located.

The NEW__COORDINATES procedure looks at the direction the player has moved. This information is output by the HEADING__DIRECTION procedure in the GET__DIRECTION group. Each row of the maze is $-10$ units away from the row below it. If the player has moved up, the NEW__COORDINATES procedure updates only the Y coordinate by subtracting 10 from its former position. Similarly, right, left, and down are handled by adding or subtracting 10 from the former X or Y coordinate, as appropriate.

GET__NEW__POSSIBLE__DIRECTIONS determines which row the player is on by examining the Y coordinate of the position. It then calls the SEARCH procedure and tells it which row to look at. SEARCH, in turn, looks at each pair of lists in the variable associated with that row description (ROW1 for the first row, ROW2 for the second, and so on) until it finds the coordinates that match those of the circle. It then takes the second part of that list and puts its contents into the POSSIBLE__DIRECTIONS variable.

**BLOCK Procedure Group**

This procedure group actually carries out the program's moves. It erases the square and draws it in a new position. The heart of this procedure group, and of the program from an AI perspective, is the procedure called FIND__PROBABLE.

The FIND__PROBABLE procedure operates on a list called LAST__MOVES which is passed to it by the PROBABLE__DIRECTION procedure. LAST__MOVES contains the last six moves made by the player, with the most recent on the right end of the list. FIND__PROBABLE goes through the list from left to right and, if it finds sequences of moves which consist of the same move two or three times in succession, it considers this to be the most probable next move. If it finds more than one such sequence, the rightmost (in other words, most recent) combination is used because the procedure moves through the list from the oldest to the newest move.

The pattern-matching accounts for the fact that if the next move the player is apparently going to make is illegal, then that possibility is excluded.

## Exploring AI with the Intelligent Maze Program

One experiment you might undertake with this program would be to expand the size of the list of LAST_MOVES from the present 6 to 10 or 12. Does this make the game harder to win and the enemy harder to fool? Does it appreciably slow down processing?

Another interesting change to the program would be to try to eliminate the player's ability to outfox the enemy by making false moves. One way to do this would be to have the BLOCK procedure group contain a new procedure that would look for obvious stalling patterns like [L R L R L R], where the player is simply moving back and forth trying to confuse the program.

Finally, try expanding the size of the maze and drawing a more complicated one or have the program draw a maze at random so that the game is never the same twice. These kinds of changes don't have much to do with AI but would make the program more fun to use.

## Summary: What We've Learned about AI

This chapter has focused upon the important AI concept of pattern-matching. (We'll learn more about this in the next chapter when we apply pattern-matching to NLP programming.)

We explored three types of pattern-matching: template, micro and macro. We saw how predictive sequencing fits into the issue of pattern-matching and how such an approach can be used to create a seemingly intelligent enemy in a computer game.

```
            {INTELLIGENT MAZE ©1985, The Waite Group}
                  {Logo program by Ken Schieser}


{Main calling procedure}
TO SMART_MAZE
  SETUP_GAME
  PLAY
END

TO SETUP_GAME
  INITIALIZE_ROWS
  MAKE CIRCLE_COORDINATES [-40 50]
  MAKE SQUARE_COORDINATES [0 0]
  MAKE POSSIBLE_DIRECTIONS [U]
  MAKE BLOCK_DIRECTION NIL
  MAKE LAST_MOVES [0 0 0 0 0 0]
  DRAW_MAZE
  DRAW_CIRCLE
  DRAW_SQUARE
END

TO INITIALIZE_ROWS
  MAKE ROW1
    [[[20 -40][D R]][[30 -40][L R]][[40 -40][U D L]]]
  MAKE ROW2
    [[[-20 -30][D R]][[-10 -30][L R]][[0 -30][D L R]]
    [[10 -30][L R]][[20 -30][L U D]][[40 -30][U D]]]
  MAKE ROW3
    [[[-30 -20][D R]][[-20 -20][U L]][[0 -20][U D]]
    [[20 -20][U R]][[30 -20][D L R]][[40 -20][U L]]]
  MAKE ROW4
    [[[-30 -10][U D]][[0 -10][U D]][[30 -10][U D]]]
  MAKE ROW5
    [[[-30 0][U D R]][[-20 0][L R]][[-10 0][L R]]
    [[0 0][U D L R]][[10 0][L R]][[20 0][L R]]
    [[30 0][U D L]]]
  MAKE ROW6
    [[[-30 10][U D]][[0 10][U D]][[30 10][U D]]]
  MAKE ROW7
    [[[-40 20][D R]][[-30 20][U L R]][[-20 20][D L]]
    [[0 20][U D]][[20 20][D R]][[30 20][U L]]]
  MAKE ROW8
    [[[-40 30][U D]][[-20 30][U D R]][[-10 30][L R]]
    [[0 30][U L R]][[10 30][L R]][[20 30][U L]]]
  MAKE ROW9
    [[[-40 40][U R]][[-30 40][L R]][[-20 40][U L]]]
END

TO DRAW_MAZE
  CS
  PENPAT :DKGRAY
  PAINTRECT [-25 -15 -5 -5]
  PAINTRECT [-25 5 -5 15]
  PAINTRECT [5 -15 25 -5]
  PAINTRECT [5 5 25 15]
  PAINTRECT [-15 -25 -5 -15]
  PAINTRECT [-15 15 -5 25]
  PAINTRECT [5 -25 15 -15]
  PAINTRECT [5 15 15 25]
  PAINTRECT [-35 25 -25 35]
  PAINTRECT [25 -35 35 -25]
  PAINTRECT [-45 -45 -25 -25]
  PAINTRECT [25 25 45 45]
  PAINTRECT [-45 -25 -35 15]
  PAINTRECT [-25 -45 15 -35]
  PAINTRECT [-15 35 25 45]
  PAINTRECT [35 -15 45 25]
  PAINTRECT [-55 -55 -45 35]
  PAINTRECT [45 -35 55 55]
  PAINTRECT [-45 -55 55 -45]
  PAINTRECT [-55 45 45 55]
  PENPAT :BLACK
END

TO DRAW_CIRCLE
  MAKE X FIRST :CIRCLE_COORDINATES
  MAKE Y LAST :CIRCLE_COORDINATES
  PAINTOVAL(SE :Y-4 :X-4 :Y+4 :X+4)
END

TO ERASE_CIRCLE
  MAKE X FIRST :CIRCLE_COORDINATES
  MAKE Y LAST :CIRCLE_COORDINATES
  ERASEOVAL(SE :Y-4 :X-4 :Y+4 :X+4)
END

TO DRAW_SQUARE
  MAKE X FIRST :SQUARE_COORDINATES
  MAKE Y LAST :SQUARE_COORDINATES
  PAINTRECT (SE :Y-4 :X-4 :Y+4 :X+4)
END

TO ERASE_SQUARE
  MAKE X FIRST :SQUARE_COORDINATES
  MAKE Y LAST :SQUARE_COORDINATES
  ERASERECT (SE :Y-4 :X-4 :Y+4 :X+4)
END

TO PLAY
  GET_DIRECTION
  MOVE_CIRCLE
  IF FREE [MOVETO 30 -60 DRAWSTRING <<FREE>> STOP]
  IF (COUNT :POSSIBLE_DIRECTIONS) > 2 [BLOCK]
  PLAY
END

TO FREE
  IF :Y= -50[OP 'TRUE][OP NIL]
END
```

```
TO GET__DIRECTION
  MAKE HEADING READMOUSE
  MAKE DIRECTION HEADING__DIRECTION
  IF OR (NOT MEMBERP :DIRECTION :POSSIBLE__DIRECTIONS) BLOCKED
    [GET__DIRECTION STOP]
  MAKE LAST__MOVES UPDATED :LAST__MOVES
END

TO READMOUSE
IF BUTTON [OP GETMOUSE][READMOUSE]
END

{Function heading__direction analyzes mouse position and outputs a direction}
TO HEADING__DIRECTION
  MAKE X FIRST :CIRCLE__COORDINATES
  MAKE Y LAST :CIRCLE__COORDINATES
  IF :Y ≤ ((LAST :HEADING) - 10) [OP 'D]
  IF :Y ≥ ((LAST :HEADING) + 10) [OP 'U]
  IF :X ≤ ((FIRST :HEADING) - 10) [OP 'R]
  IF :X ≥ ((FIRST :HEADING) + 10) [OP 'L]
  OP NIL
END

TO BLOCKED
  IF EQUALP :DIRECTION :BLOCK__DIRECTION
    [OP 'TRUE][OP NIL]
END

{Function update is responsible for recording the last six valid moves}
TO UPDATE :LIST
  MAKE :LIST BF :LIST
  MAKE :LIST LPUT :DIRECTION :LIST
  OP :LIST
END

TO MOVE__CIRCLE
  ERASE__CIRCLE
  MAKE CIRCLE__COORDINATES NEW__COORDINATES
  GET__NEW__POSSIBLE__DIRECTIONS
  DRAW__CIRCLE
END

{Function new__coordinates analyzes direction & outputs new coordinates}
TO NEW__COORDINATES
  MAKE X FIRST :CIRCLE__COORDINATES
  MAKE Y LAST :CIRCLE__COORDINATES
  IF :DIRECTION = 'U [MAKE Y :Y - 10]
  IF :DIRECTION = 'D [MAKE Y :Y + 10]
  IF :DIRECTION = 'L [MAKE X :X - 10]
  IF :DIRECTION = 'R [MAKE X :X + 10]
  OP SE :X :Y
END

{Procedure get_new_possible_directions finds which row to search for the new
possible directions}
TO GET_NEW_POSSIBLE_DIRECTIONS
  MAKE BLOCK_DIRECTION NIL
  IF :Y = -40 [SEARCH :ROW1]
  IF :Y = -30 [SEARCH :ROW2]
  IF :Y = -20 [SEARCH :ROW3]
  IF :Y = -10 [SEARCH :ROW4]
  IF :Y = 0 [SEARCH :ROW5]
  IF :Y = 10 [SEARCH :ROW6]
  IF :Y = 20 [SEARCH :ROW7]
  IF :Y = 30 [SEARCH :ROW8]
  IF :Y = 40 [SEARCH :ROW9]
END

{Procedure search looks for a match between the present coordinates & the first set of
each element of the row}
TO SEARCH :ROW
  IF EQUALP SE :X :Y FIRST FIRST :ROW
    [MAKE POSSIBLE_DIRECTIONS LAST FIRST :ROW STOP]
  SEARCH BF :ROW
END

TO BLOCK
  ERASE_SQUARE
  MAKE DIRECTION PROBABLE_DIRECTION
  MAKE SQUARE_COORDINATES NEW_COORDINATES
  DRAW_SQUARE
  MAKE BLOCK_DIRECTION :DIRECTION
END

{Function probable_direction initializes variables for procedure find_probable then
output the most probable move}
TO PROBABLE_DIRECTION
  MAKE NEWCOUNT 1
  MAKE OLDCOUNT 1
  FIND_PROBABLE :LAST_MOVES
  OP :MOST_PROBABLE
END

{Procedure find_probable makes most_probable the direction.}
TO FIND_PROBABLE :LIST
  MAKE CHARACTER FIRST :LIST
  IF (COUNT :LIST) < 2
    [IF AND :OLDCOUNT = :NEWCOUNT MEMBERP
    LAST :LIST :POSSIBLE__DIRECTIONS
    [MAKE MOST__PROBABLE LAST :LIST] STOP]
  TEST (ITEM 1 :LIST) = (ITEM 2 :LIST)
  IFTRUE [MAKE NEWCOUNT :NEWCOUNT + 1]
  IFFALSE [IF AND (:NEWCOUNT ≥ :OLDCOUNT)
    (MEMBERP :CHARACTER :POSSIBLE__DIRECTIONS)
    [MAKE MOST__PROBABLE :CHARACTER
    MAKE OLDCOUNT :NEWCOUNT
    MAKE NEWCOUNT 1][MAKE NEWCOUNT 1]]
  FIND__PROBABLE BF :LIST
END
```

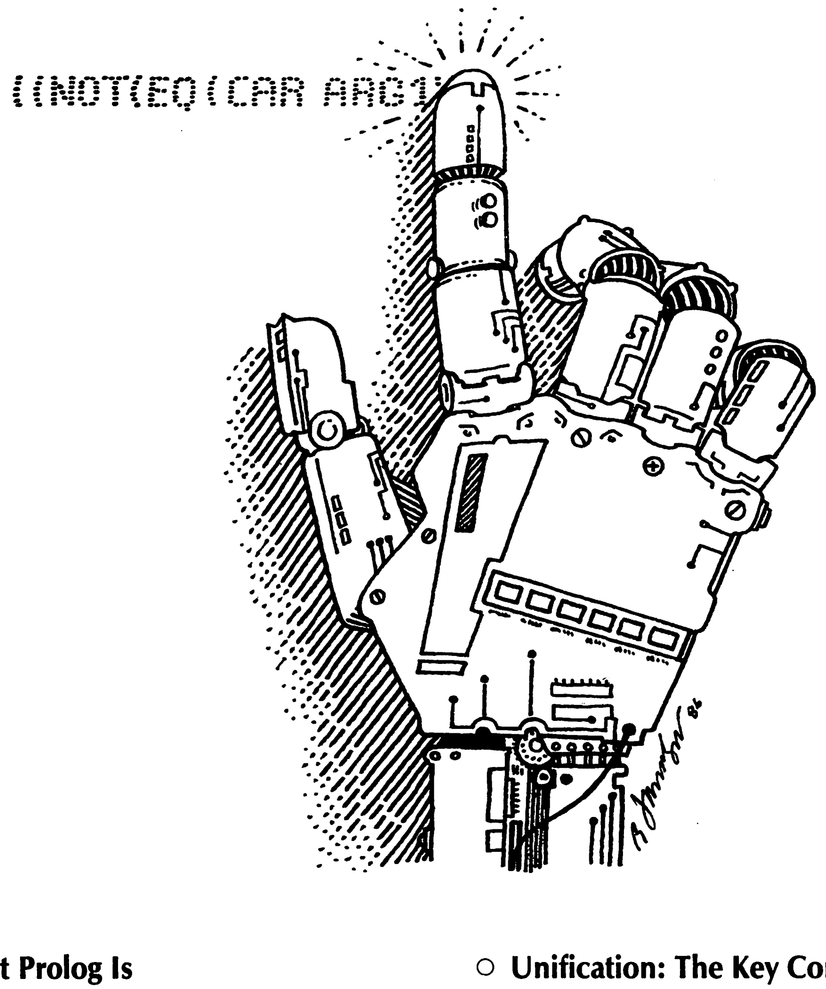
# II

# *Artificial Intelligence Data Bases*

CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL IN
INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELL

# 7

# A Prolog Interpreter



- What Prolog Is
- Why Prolog Is Important in AI
- Prolog in Expert Systems
- Unification: The Key Concept
- Goals, Environments, and Searches

In this and the next chapter we will concentrate on the *application* of AI programming techniques in programs with potential real-world uses. As a result, the chapters explore concepts and problem-solving rather than specific AI programming skills.

In this chapter we will learn to use a Logo program that implements a scaled-down version of a widely used AI programming language, Prolog. Because the interpreter is written in Logo, the program is called "Prologo." Chapter 8 carries this discussion beyond to the process of designing knowledge bases for Prologo for use as demonstrations of scaled-down expert systems.

## Why Prolog or Prologo?

Prolog is the best-known and most widely used example of *nonprocedural* programming language. Later this chapter will make clear what this distinction is and why it is important. We can gain a great deal of insight into programming languages by seeing Prolog, a nonprocedural language, written in a procedural language like Logo. The process of taking a language apart and dividing it into neat procedures goes a long way toward demystifying the language. In fact, in college AI programming courses, a common assignment is to write an interpreter for one language in a completely different language.

Beyond this pure learning process, you can learn more about AI from playing with this program than from dozens of theoretical discussions. Prolog is a powerful language in its full implementation and even though we will be dealing with a small subset of the language, you will soon grasp and appreciate the power of the language and the ease and elegance of programming in it. You may even want to run out and learn Prolog when you're done with this chapter! (If that bug bites, see the references in Appendix C for a place to start.)

## The Program's Background

Before delving into the intricacies of Prolog programming, we should acknowledge the source of the program we are about to review. Prologo was written by Steve Lurya of ExperTelligence. His work was built to some degree on an earlier implementation of the language in ExperLogo® written for ExperTelligence by John Worthington. Worthington, in turn, borrowed many of the ideas for his Logo implementation of Prolog from a program called PiL, an acronym for Prolog in LISP. That program, in turn, grew out of a paper entitled "The world's shortest Prolog interpreter?" by M. Nilsson of Uppsala University, published in *Implementations of Prolog* (Campbell 1984).

ExperTelligence introduced Prologo as one of several programs on a disk of demonstration programs to show the power of the ExperLogo® language. The company encourages widespread enhancement, duplication, and usage of the program.

## Prolog and Expert Systems

Prolog, in its full implementation, is an extremely powerful AI language. In fact, it is so powerful that the Japanese originally chose Prolog as the language to be used in carrying out their widely publicized and well-funded Fifth Generation Project. (The Japanese are becoming increasingly interested in LISP as the language of choice.) In Europe, particularly in the United Kingdom and France, Prolog is clearly the language of choice of AI experimenters. It is worth learning about Prolog if for no other reason than that a great part of the AI research community outside the United States relies largely or exclusively on this language for its extensive research.

In the United States, Prolog and its adaptations have been used for AI research. It has been used extensively in designing expert systems. Prolog is particularly well suited to the kind of expert systems design used in this chapter because it is data-driven (or, in other words, knowledge-based) rather than rule-driven. (More about that in a moment.)

## What Is Prolog?

Prolog is unlike any other programming language you will encounter. Most computer programs—including those written in Logo and LISP—consist primarily of statements that tell the computer what to do each step of the way in a problem-solving situation. Prolog, on the other hand, permits the user to describe the problem and as much as is known of the solution. The program then uses deductive reasoning techniques to fill in the blanks and solve the problem. So, in a sense, Prolog appears to be less a *language* than a *program* itself. It does *implicitly* what would have to be *explicitly* programmed in a more traditional, procedural language.

Because of this fundamentally different design, Prolog programs tend not to look very much like traditional computer programs. Here is a small example of such a program:

```
David-Senior father-of David-Junior
Daniel father-of Sheila
Alice mother-of David-Junior
Carolyn mother-of Sheila
David-Junior male
Sheila female
```

As it now stands, this program doesn't "do" anything. An interesting and important aspect of Prolog programs is that they *describe* relationships rather than "doing" something.

The statements contained in this sample descriptive knowledge base are all axioms. An axiom is a statement given as true; there need not be any connection between an axiom and reality. The instant we place such a statement as "Orange bears dance Irish jigs" in a Prolog program, the program accepts the statement as true. In Prolog, then, proving the validity of any statement about a knowledge base is accomplished by merely stating the axiom(s) of which the proof consists. No proof of the axioms is needed. Axioms are one of two main types of objects that are understood by Prologo. (We'll get to the other in a moment.) We run such a program by asking it questions like:

```
who (x : x father-of David-Junior)
```

We get an answer that looks like this:

**David-Senior**
**No (more) answers**

Prolog scanned its knowledge base, consisting of the facts presented earlier about family relationships, and found any that would "match" the pattern: blank father-of David-Junior. It filled in the blank and gave us the answer. Unless we tell it not to, it continues that search through the entire knowledge base to see if there are any other matches for our query.

Prolog—and, by extension, Prologo—operates on what is often referred to as the "closed-world assumption." In brief, this means that anything included in the knowledge base with which the program is running is true and everything else is nonexistent. In other words, the world stops at the borders of the knowledge base.

As shown in Figure 7-1, if the sharp, jagged lines represent the closed world that Prologo knows about, it can be said to know about the seven trees and two circles inside its world but to be completely ignorant of the trees and rectangles that lie just outside its borders of knowledge.

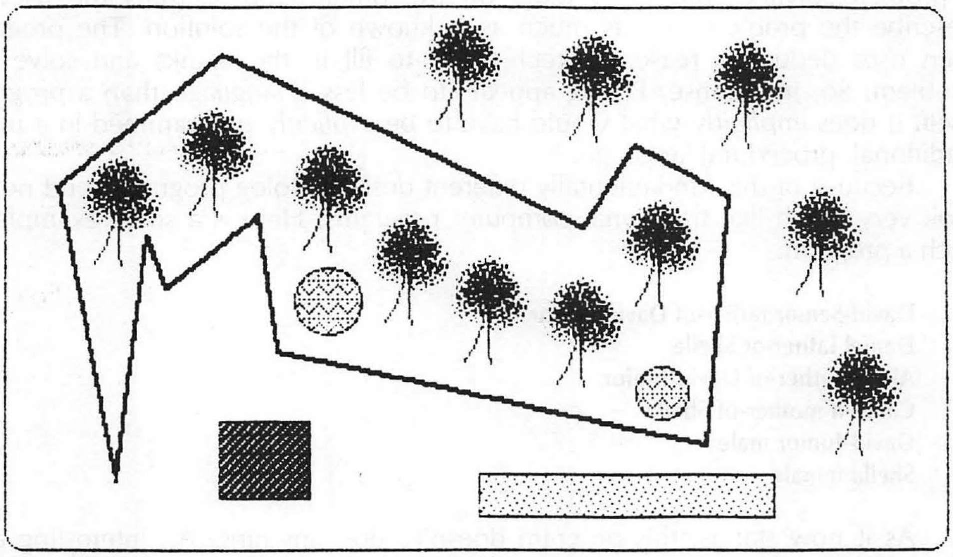


**Figure 7-1. The closed world of Prolog**

**More Complexity. . .**

If Prolog could merely go through a knowledge base and retrieve information, it would be neither useful nor unusual. But it *is* useful and unusual. A number of features contribute to these characteristics. Perhaps most important, the knowledge base may contain *rules* that define more general relationships, in addition to *facts* that describe specific relationships. These rules are the second type of Prolog object that may be found in a Prolog knowledge base.

For example, we could add to the existing knowledge base the rule:

**x parent-of y if x father-of y or x mother-of y**

Now, we could pose the following query, with the indicated result (computer response is indented):

```
which(x : x parent-of Sheila)
  Daniel
  Carolyn
  No (more) answers
```

The fact that Daniel is a *parent* of Sheila exists nowhere explicitly in our knowledge base. Nor does the fact that Carolyn is Sheila's parent. Rather, we have asked Prolog to *infer* from the information it has been given whether it can determine who Sheila's parents might be. It does so by applying the rule that a parent is either a father or a mother of a person. It is this ability to draw inferences that sets a Prolog knowledge base apart from the otherwise ordinary data base with which many programming languages work quite effectively (See Figure 7-2).

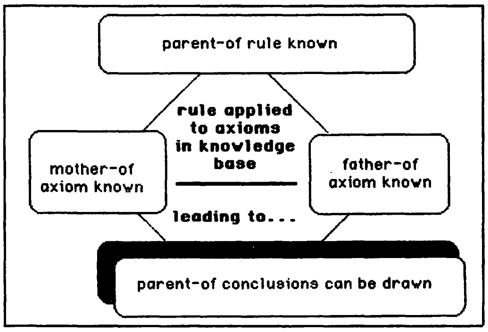


**Figure 7-2. Rules + axioms = conclusions not explicitly stated**

We can design extensive and seemingly intelligent programs in Prolog by defining complex rules and increasingly complex relationships.

**But It's All Descriptive**

Note that no program statement tells Prolog to follow a set of instructions like this:

1. Read the first fact in your knowledge base.
2. Compare the first element in your first fact list with the first element in the query being processed.
3. If those are equal, repeat step 2 with the next element in the fact list with the next element in the query being processed.
4. If you run out of facts in your fact list and elements in the query and nothing has been found to be different between them, report the fact list as an answer to the query and return to step 1 unless the knowledge base has been completely scanned, in which case print "No (more) answers" and quit.

This set of instructions doesn't take into account the drawing of inferences from existing rules; that list of instructions would be more than twice the length of the list above. But none of this appears in the Prolog program; Prolog is *designed* to do those things automatically by looking through a knowledge base in response to queries. In other words, you tell Prolog what you want to know, not how to find it out. (For a fuller discussion of Prolog, see Chapter 12.)

## Prologo: An Overview

A complete implementation of a programming language as rich and complex as Prolog would require more space than we have here. The program would also be so long that you'd *never* want to spend the many hours it would take to type it into your Mac and debug it. So we will implement a kernel of Prolog in Prologo.

This section describes Prologo as a language subset and discusses how to run Prologo. It also explains aspects of Prolog not implemented in this version. Additionally, we will look at the substantial, important differences between the syntax of Prologo and Prolog. Some of the differences occur because Logo looks at the world differently from Prolog. Others are due to the implementation limitations of ExperLogo® itself.

### Running Prologo

In previous chapters, you have been used to reading the materials first, then analyzing the program and finally, if you wished to do so, typing the Logo listing into your Mac and running it. This chapter is different.

Pause now to type Prologo into your Mac, save it, debug it, and get it ready to run. This will enable you to follow the discussion of the Prologo language by hands-on performance at your Mac's keyboard. Please don't go on until you've got Prologo ready to run.

Now that you've typed in Prologo and checked it out to be sure that it compiles correctly, it needs a knowledge base upon which it can operate. In the implementation of Prologo that we are using here, a knowledge base is created explicitly with a Logo "make" command. The procedure that follows creates the sample knowledge base. Type the procedure into an Edit Window in ExperLogo®, run it, and *then* execute Prologo.

```
make family_db
[[[father-of jim rick]]
[[father-of jim susan]]
[[grandparent-of _grandparent _grandchild]
  [parent-of _grandparent _parent]
  [parent-of _parent _grandchild]]
[[mother-of rhoda rick]]
[[mother-of anne jim]]
[[parent-of _parent _child]
  [mother-of _parent _child]]
[[parent-of _parent _child]
  [father-of _parent _child]]]
```

This knowledge base consists of a list of lists. Some of the lists (e.g., [[father-of jim susan]]) describe specific individual relationships. Others (e.g., [[grandparent-of _grandparent _grandchild][parent-of _grandparent _parent][parent-of _parent _grandchild]]) are the equivalent of Prolog rules. This particular one, translated into literal English, would read: "A grandparent-of relationship is defined as consisting of a grandparent and a grandchild. If a person called '_parent' (note that this is not the *relationship* called *parent-of*, but a new *person* called '_parent') is *both* in a parent-of relationship with the grandparent *and* a parent-of to another person called '_grandchild.'"

This will become clearer as we make inquiries of the knowledge base we call "family_db."

With this knowledge base entered into memory via the "make" instruction, we are now ready to run Prologo. We load the program and choose the Run All option from the ExperLogo® Run menu. Then we type:

```
Prologo :family_db
```

and Prologo responds with:

```
Welcome to Prologo
```

The program is waiting for an inquiry (which Prologo will refer to as an "assertion"). Prologo has two types of assertions. The first is the general inquiry that translates to "Tell me everything you know." The second is the more commonly used inquiry that seeks an answer to a specific question.

**What Do You Know?**

We use a single command to cause Prologo to print everything it knows or can deduce from the knowledge base that we have provided: _what. With the family_db knowledge base as input, the _what command produces the following result:

```
_what
  [father-of jim rick]
  what = father-of jim rick
  [father-of jim susan]
  what = father-of jim susan
  grandparent-of anne rick
  what = grandparent-of anne rick
  [grandparent-of anne susan]
  what = grandparent-of anne susan
  [mother-of rhoda rick]
  what = mother-of rhoda rick
  [mother-of anne jim]
  what = mother-of anne jim
  [parent-of rhoda rick]
  what = parent-of rhoda rick
  [parent-of anne jim]
  what = parent-of anne jim
  [parent-of jim rick]
  what = parent-of jim rick
  [parent-of jim susan]
  what = parent-of jim susan
  no (more) answers
  Another assertion? (y or n)
```

Since our knowledge base has only two rules—the grandparent-of rule and the parent-of rule—we can trace this listing through to determine how Prologo got these answers. I suggest you do so as an exercise in learning how Prologo "thinks." Incidentally, because of ExperLogo's® design, each response appears twice. Logo produces the list response—the first of each pair of identical answers. ProLogo produces the second, more English-seeming answer.

If you look through the procedures making up this Prolog subset, you won't find a statement that tells the program what to do if it encounters the command "_what." In fact, Prologo interprets an initial underscore followed by any text at all, to mean, "Tell me all you know." We use the term _what because it makes it clear to us that we are asking that question (or, more accurately, making that assertion). The underscore, not the word that follows, causes Prologo to give us a complete analysis of the knowledge base.

**Confirming a Single Fact**

Another thing Prologo can do besides completely "dump" what it "knows" is confirm a single fact by means of an assertion that the fact in question is true. If we assert a fact that is true as far as the knowledge base is concerned, it will repeat the fact to us; if our assertion is unknown to the knowledge base, it will not return an answer. Examine the following exchanges to see what I mean.

```
[father-of jim rick]
  [father-of jim rick]
  Another assertion? (y or n)y
[father-of jim steve]
  Another assertion? (y or n)y ← assertion was unknown
[grandparent-of anne rhoda]
  Another assertion? (y or n)n ← assertion was unknown
  Back to Logo
```

Notice that we do not use any words preceded by underscores in the above assertions. Prologo uses the underscore to designate unknown or variable values and the assertions in our example do not contain specific unknown information. We are merely confirming that an assertion we are making is valid according to the knowledge available to the program.

Notice, too, that each answer is printed only once. Again, that is because there is no variable for Prologo to "fill in" a value for—and as a result we get only Logo's displayed return answer.

The assertion that anne was a grandparent-of rhoda, received a negative response (i.e., Prologo did not print the assertion back to us, indicating that the information was not in its knowledge base). We know anne is a grandparent-of someone, but we we haven't selected a correct grandchild—though rhoda is someone about whom the program has information. This differs from our other wrong assertion, where steve simply doesn't exist as far as our knowledge base is concerned.

The important point here is that any assertion we make in an effort to determine its veracity will be negative whether it is a nonsense inquiry or simply inaccurate. If we asked our family_db knowledge base:

```
[better-of Prolog LISP]
```

its answer would be negative. It wouldn't say something like, "Come on, human, I only know about families." All it knows is that it has no such relationship as "better-of" and it never heard of Prolog or LISP.

These exchanges prepare us to understand why inquiries to the Prologo knowledge base are called "assertions." In each case we assert to Prologo that some relationship is valid and ask it to confirm the validity of the data. It is almost as if we were saying to Prolog, "[father-of jim rick], right?"

**Finding Unknown Information**

The most frequent use of Prolog in connection with a knowledge base is extraction of unknown data. This is its primary function in expert system designs. Expert systems help the user determine a course of action or an applicable diagnosis. Users employ variables to make assertions and Prologo fills in the variables with known data.

For example, to find out who is the father-of rick, we would conduct the following interchange with the program:

```
[father-of _who rick]
  [father-of jim rick]
  who = jim
  no (more) answers
  Another assertion? (y or n)
```

Prologo finds the name "jim" in the data base and puts it in the variable called _who. We have defined only one father for rick, so it returns one answer and then informs us it has no more responses. If we change the assertion slightly, we would get more than one response:

```
[parent-of _who rick]
  [parent-of rhoda rick]
  who = rhoda
  [parent-of jim rick]
  who = jim
  no (more) answers
  Another assertion? (y or n)
```

Here, two people—rhoda and jim—fit the rule for the parent-of relationship with rick so the program supplies both names as valid values for the variable _who. (See Figure 7-3.)

As you may have expected, the choice of the variable name is not important. We could use _name or _person or _bxtflk as the variable to be filled in and Prologo would handle the problem identically. We use _who because it describes what we are trying to determine with our assertion.

It seems strange to continue using the term "assertion" when we clearly are not asking Prologo to confirm the accuracy of a factual statement as we were in the examples in the previous section. But if we stop to think about it, it will make sense. When we pose the query:

```
[father-of _who rick]
```

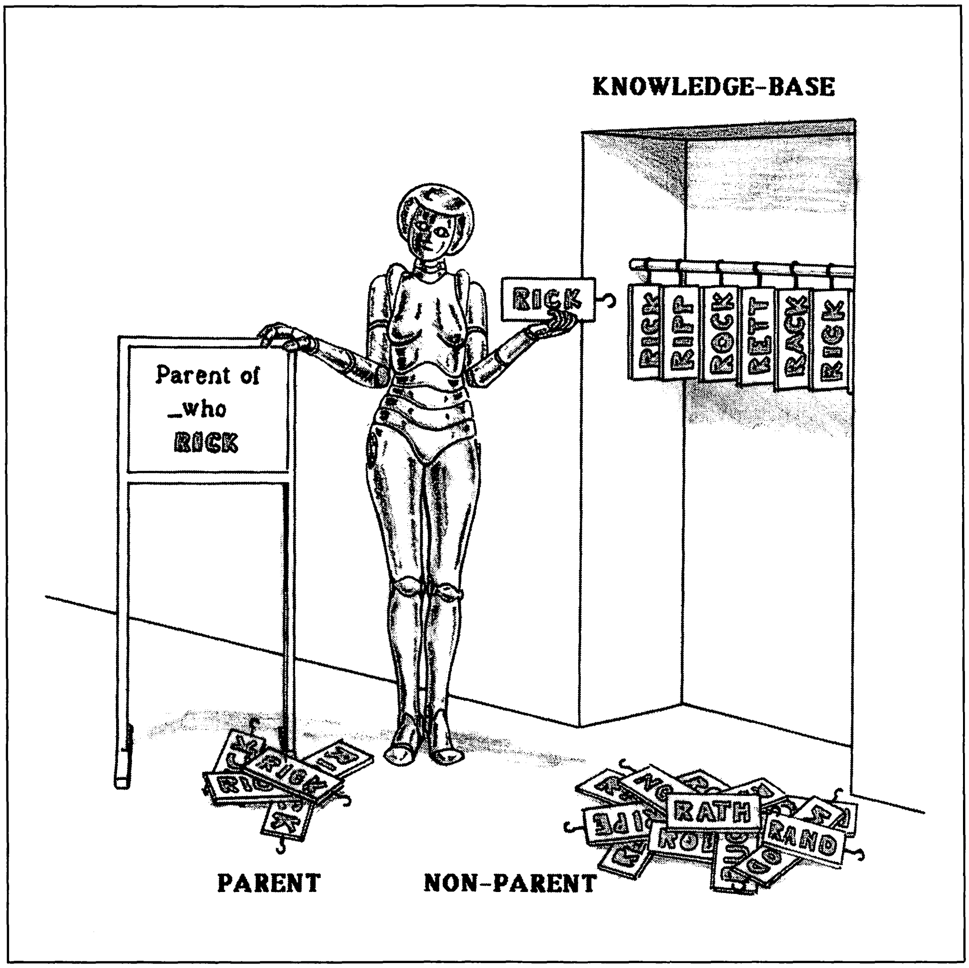


Figure 7-3. Some fit and some do not

we are really asking, "You have someone who is in a father-of relationship with rick, don't you?" Prologo responds by telling us who that is (if it knows) or simply ignoring our assertion (if the data is unknown).

So the term "assertion" is still used in such inquiries, even though we might think another alternate term would be more accurate. This makes it possible for us to talk more consistently about how we use Prologo.

This is one of the most obvious differences between Prologo and Prolog. In Prolog, we must explicitly cause the program to display the results of its analysis of our assertions before it will do so. (To see *how* to do so, refer to Chapter 12.) Prologo, on the other hand, supplies the variable name in an assertion that automatically displays the result of the analysis.

***Multiple Unknowns*** Sometimes we supply assertions in Prolog or Prologo that involve more than one missing piece of information. When we do so, the real power of the language comes to the fore. It fills in each variable for every occurrence it finds that makes the assertion true.

For example, suppose we want to know about all the parent-of relationships in the knowledge base. We supply an assertion like this:

```
[parent-of _parent _child]
```

This is the equivalent of asking Prologo, "You know about at least one parent-of relationship between two people, don't you?" Prologo dutifully responds by presenting the names of parents and children in each such relationship:

```
[parent-of _parent _child]
  [parent-of rhoda rick]
  child = rick
  parent = rhoda
  [parent-of anne jim]
  child = jim
  parent = anne
  [parent-of jim rick]
  child = rick
  parent = jim
  [parent-of jim susan]
  child = susan
  parent = jim
  no (more) answers
  Another assertion? (y or n)
```

Compare this result with the display Prologo produced from this knowledge base when we asked it what it knew and you will see that all parent-of relationships have been reported, with the variables filled in correctly and *only* parent-of relationships have been listed.

**Performing Calculations in Prologo**

Prologo can understand formulas as well as facts and rules. A formula is merely a special rule. To demonstrate we will use a new knowledge base.

Type the following small knowledge base called, immodestly, world_db, into an Edit Window in Prologo and run it. Then type:

```
Prologo :world-db
```

and follow the examples in the rest of this section. (Note that you now have two knowledge bases resident in Prologo; you choose which one to run by supplying its name after the Prologo command.)

```
make world_db
  [[[population usa 203]]
  [[population china 800]]
  [[area usa 3]]
  [[area china 4]]
  [[density-of _country _d]
```

```
        [population _country _p]
        [area country _a]
        [is _d[/ _p _a]]]
```

This knowledge base is similar to the family_db knowledge base with which we have been working, except in its last entry, which appears to define a rule called "density-of." Other than that, it gives the populations of the United States and China in millions, and the areas of the two countries in millions of square miles. The usual queries will produce the expected results:

```
[population usa _howmany]
  [population usa 203]
  howmany = 203
  no (more) answers
  Another assertion? (y or n)y
[area _whichone 4]
  [area china 4]
  whichone = china
  no (more) answers
  Another assertion? (y or n)n
  Back to Logo
```

If we make an assertion about the density-of relationship, we see what calculation is performed. Two variables will get all of the density-of responses the knowledge base has in it.

```
[density-of _country _density]
  [density-of usa 67.66666666666]
  country = usa
  density = 67.66666666666
  [density-of china 200]
  country = china
  density = 200
  no (more) answers
  Another assertion? (y or n)
```

When we inquire about the density-of relationship, Prologo divides the population by the area and comes up with a people-per-square-mile figure. In the case of the United States, the answer comes up 67.66666666666 people per square mile; in China, it is 200 people per square mile.

The "is" operator in the density-of rule makes the calculation possible. Translated into technical English, the rule about density-of would read: "The density-of relationship exists between two variables called _country and _d. This latter variable, _d, is calculated by looking up the population of a country called _country and storing it in a new variable called _p, then looking up the area of a country called _country and storing it in a new variable called _a, and then dividing _p by _a." In one sense, this is an *imperative* command that tells Prologo how to calculate the density of a country given information about its size and population. In another sense, though, you can see how density-of describes the relationship it names.

Prolog, but not Prologo, permits comparisons on the results of calculations defined with the "is" operator. For example, the following assertion would be valid in Prolog, though Prologo will act as if it weren't true:

```
[[density-of china _what]>[density-of usa
_what2]]
```

## The Program

We now turn our attention from how to run Prologo to an examination of how the Prologo program itself works. Along the way, we will look closely at unification—one of the most powerful AI programming concepts.

### An Overview of the Program

Prologo is the most complex Logo program in this book. Not only is its processing complex, it also uses some statements unique to ExperLogo®. These are explained in Appendix B. Appendix B also demonstrates how to convert between these ExperLogo® statements to more conventional Logo commands. That may dispose of the language complexities, but the processing complexities of the program are such that a thorough overview of its activity will help you understand the discussion that follows on the individual procedures.

### What Prologo Does

Prologo begins with a goal (an assertion we make about the knowledge base) and attempts to attain it by proving that the assertion of it is true in the knowledge base. As the program runs, it may create "subgoals" that must be proven before other goals can be established, including our original assertion.

When it begins to execute an assertion about a knowledge base, Prologo tries to match, or "unify," the current goal (the fill-in-the-blanks part of the assertion) with the first clause of the first axiom in the knowledge base. In the simplest case, the assertion and the first axiom match and Prologo's job is complete. Thus, in our earlier sample family knowledge base, the assertion:

```
[father_of jim rick]
```

is unified with the first clause of the first axiom in the knowledge base, which is identical to the assertion in content and structure.

The process of unification can be thought of as the process of finding something to fill in every blank or variable position in an assertion. If no variables are included in the assertion, the process boils down simply to trying to find the assertion present in the knowledge base as an axiom.

If, as is usually the case, the first clause of the first axiom in the knowledge base does *not* immediately unify with the assertion, then Prologo chops off the first axiom from the knowledge base and tries to unify the assertion with the first clause of the next assertion. It continues this process until it has searched through the entire knowledge base.

***If Unification Is Successful*** When some portion of a current goal is successfully unified with the first clause of an axiom in the knowledge base, Prologo has progressed toward satisfaction of the goal. In that case, Prologo appends the remaining part of this axiom to the beginning of the list of all other goals left unproved at that point. The program then tries to prove this newly established goal.

If the program is not successful in proving the new clause of the axiom, the entire axiom is dropped from the knowledge base, its once-promising role as a goal-prover no longer useful, and the process moves on to the next axiom in the knowledge base.

***How It Works*** Keep this description of how Prologo works in mind as we become more specific in the next few pages about the individual procedures, their relationships with one another, and the precise way in which the unification process occurs.

**Procedures and Relationships**

Figure 7-4 is a box diagram of the Prologo program. The primary top-level procedures are PROLOGO, PROLOGO__1, PROVE, TRY__EACH, and UNIFY.

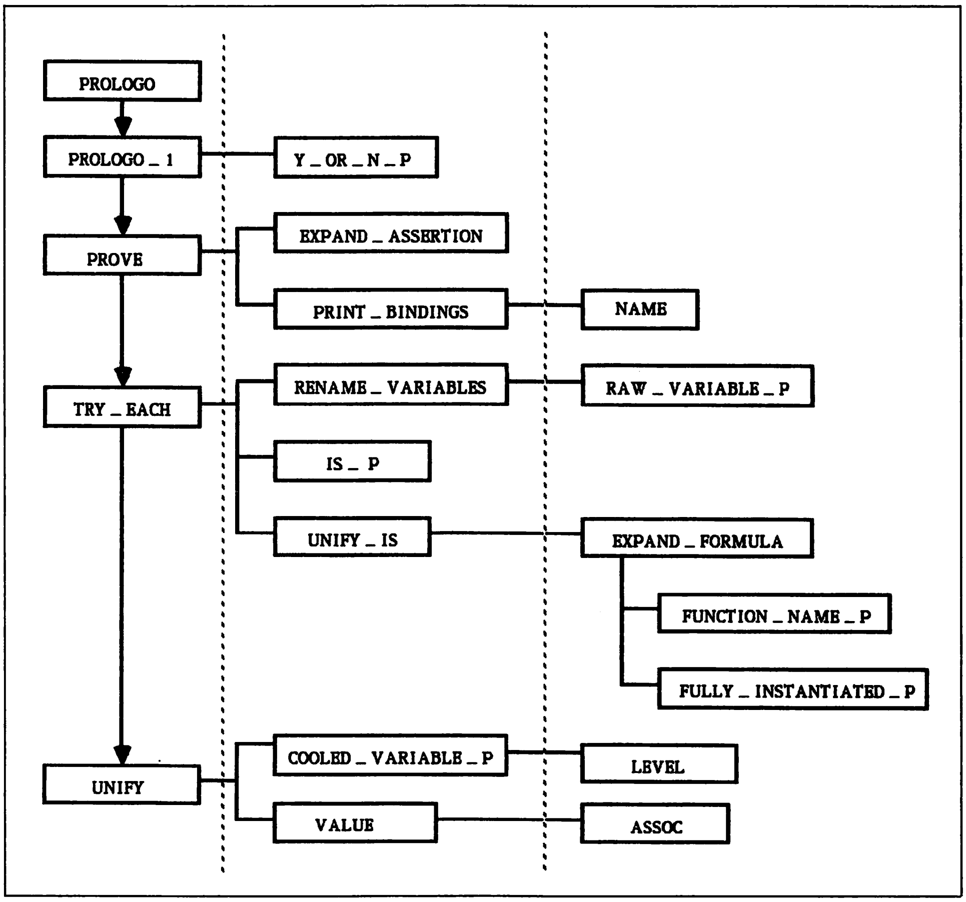


**Figure 7-4. Box diagram of Prologo**

## PROLOGO and PROLOGO_1

These two main driver routines set up Prologo to run and manage its top-level needs. PROLOGO itself prints a welcome message and then calls PROLOGO_1. When the user ends a session with Prologo, this procedure prints the "Back to Logo" message.

PROLOGO_1 is only slightly more complex. It reads an input, uses the RENAME_VARIABLES procedure to handle the designation of variables in the input, calls the PROVE procedure to attempt to prove the assertion in the entry, prints the message, "Another assertion? (y or n)," and then calls the Y_OR_N_P procedure. If the user responds "yes," the program returns to this procedure and uses the same knowledge base.

## The PROVE Procedure

PROVE is just one big *if* statement. It determines *if* there are any goals left to prove. *If* there aren't, then the task it is working on is complete. *If* any goals remain, though, PROVE calls TRY_EACH (discussed in a moment).

PROVE prints the result of any matches (i.e., successful unifications) by calling the PRINT_BINDINGS procedure and, if there are still goals left to deal with, calls TRY_EACH once again.

The list of arguments passed to PROVE comprises the current list of goals, the original goal, something called the "environment," the knowledge base (represented by the variable :database), and a level. Let's pause to discuss the list of goals, the environment, and the meaning of "level" in Prologo. Doing so here will clarify much of what follows in our analysis of the program.

***The List of Goals*** The first argument given to PROVE when it is called is a variable called :list_of_goals. The first time we run the PROVE procedure on a specific assertion, the list will correspond to the assertion being evaluated. But when Prologo encounters a *rule* in its knowledge base and attempts to prove its validity against the current assertion, the subgoal(s) involved will be passed back to PROVE. This is because rules, by their nature, include variables in their statements. The existence of the variable creates a subgoal that the program must prove before it can prove the original goal.

Figure 7-5 clarifies the way this works. When Prologo encounters the rule about grandparent-of in the family knowledge base, it creates a new goal or assertion, one level deeper into the analysis of the knowledge base.

| ***Goal Being Examined*** | ***Knowledge Base*** | ***Goal/Subgoal Added*** |
|---|---|---|
| grandparent-of anne [_who0] | [father_of jim rick] | none |
| grandparent-of anne [_who0] | [father_of jim susan] | none |
| grandparent-of anne [_who0] | grandparent_of] rule | parent-of [_grandparent 1] [_parent 1] |
| parent-of [_grandparent 1][_parent 1] | [father_of jim rick] | none |
| ⋮ | ⋮ | ⋮ |
| parent-of [_grandparent 1][_parent 1] | [parent_of ]rule | mother-of [_parent 2] [child 2] |

**Figure 7-5. Subgoal creation in Prologo**

The new assertion holds that it must attempt to satisfy the parent-of relationship between a person (called, in the rule, __grandparent) and another person (referred to in the rule as __parent). It does this because it has been instructed to satisfy a grandparent-of relationship assertion and has now found that satisfying the parent-of relationship between two people is a prerequisite to satisfying the grandparent-of relationship.

Once it has generated this subgoal, the TRY__EACH procedure starts at the beginning of the knowledge base again and goes through the process of attempting to verify this assertion. This process continues until a match is found or the knowledge base and list of goals are exhausted, in which case no further answers are obtainable.

***The Environment*** At any given time during the execution of a Prologo analysis of a knowledge base, the program will have created an "environment." When you look at the PROLOGO__1 procedure, you see a line that calls the PROVE procedure with several arguments, one of which is the list of the list called "bottom__of__environment." This is the artificially created environment with which Prologo begins its assignment.

When a subgoal is created, a new environment is created, consisting of the subgoal appended to the front of a list that ends with the "bottom__of__environment" marker.

From a technical perspective, the environment is actually a list of "bindings." A binding is the formal establishment of a connection between a variable and a value. (In other programming languages, we would think of this as an "assignment" rather than a binding.) A binding is a list that includes a variable and its value. The value, in turn, may be another variable, an expression (which may in turn contain variables), a number, or a constant piece of information.

The procedure called VALUE is given the task of determining what, if any, value is currently bound to a given variable. When it finds such a value, it returns that value to the calling program, which essentially decodes the binding. If it doesn't find such a binding, it returns the variable name, which informs the calling procedure that no binding yet exists for this variable.

***Level of Search*** Each time the TRY__EACH procedure (discussed in the next section in detail) is entered recursively, the program adds one to the current value of the variable LEVEL. This level is used in the Prologo program only by the RENAME__VAR procedure. Its purpose is to ensure that we don't run into collisions in the program between variables of the same name. This might occur because in going through the knowledge base we could use some variables more than once in attempting to unify goals and axioms and these goals and axioms often have identically named variables.

Credit goes to a programmer named Kenneth Kahn for developing the idea of adding a level number to the end of the variable name and incrementing that level number only as often as needed. The result is that we avoid a tremendous amount of very complex list-handling as we try to keep track of which variable is bound to which instance of the variable(s) it contains.

### TRY__EACH

We have an idea of what TRY__EACH does from what has gone before. The procedure is responsible for determining if there are any unexamined axioms left in the knowledge base at each pass through the rules and axioms. If there are no axioms left, then we know we are dealing with a false (i.e., unprovable) goal. If there *are* remaining axioms, then they must be examined before we know whether we have been successful. The variable :database__left always contains the values left in the knowledge base.

The program does different things depending on whether UNIFY succeeds in finding a match between a goal and a fact or rule interpretation in the knowledge base. TRY__EACH may lop off the most recently tested axiom from the knowledge base and run the process again, or it may add the newly matched information to the knowledge base and continue processing one level deeper in the search.

In the course of performing its chores, TRY__EACH deals with the fact that some of the information it passes to the UNIFY procedure will be known data (in our family__db, for example, the name of a known person). Other information will be in the form of variables (i.e., preceded with an underscore, according to Prologo's rules). To ensure that this information gets passed to other procedures in a format they can handle, TRY__EACH uses procedures called RAW__VARIABLE__P and RENAME__VARIABLES.

RAW__VARIABLE__P is a predicate-testing procedure. It is, therefore, similar in function to such built-in Logo functions as LISTP. Just as LISTP determines whether an argument passed to it is a list, so RAW__VARIABLE__P determines whether an argument passed to it is a "raw variable"—a variable with no value assigned yet.

The RENAME__VARIABLES procedure restructures information passed to it in order to isolate the variable information along with the level of search from the rest of the assertion. Thus, if we carry out the task:

```
RENAME__VARIABLES [father-of jim __who]
```

at the first level of search, the procedure will return that information in a new form:

```
father-of jim [__who 0]
```

As we have already seen, this approach results in efficient operation of the Prologo program in dealing with potential name conflicts among variables in axioms, rules, and goals.

***Special Tracing Function*** TRY__EACH has one additional feature worth mentioning. Notice the group of lines beginning with the third line of the procedure. These lines begin with an IF statement and have only one clause to execute when the IF statement is true. The clause consists of four statements which display information on the Mac's screen.

If you are working with Prologo and wish to see what is happening in goal and subgoal creation, and how Prologo is analyzing each step of the processing, then before you run Prologo, type:

```
MAKE TRACE__PROLOG 'TRUE
```

Each time it executes, TRY__EACH will print the current goal and the current axiom (i.e., rule or assertion in the knowledge base currently being worked on). When it proves (or disproves) the assertion, a complete record of its analysis and processing will be preserved in the ExperLogo® Listener Window. This helps identify how Prologo understands information given to it in knowledge bases.

### UNIFY: The Key Procedure

The Prologo unification process takes place in the UNIFY procedure, which attempts to match variable information, if any is present, with data in the knowledge base.

The procedure creates the variable NEW__ENVIRONMENT, discussed earlier in conjunction with the PROVE procedure. UNIFY is a recursive procedure that continues to call itself with new arguments, lopping off the first element of the environment in which it is operating, until it successfully unifies the assertion and information contained in or inferred from the knowledge base, or until it runs out of things to try.

### The Three-Way Interaction in Prologo

The three procedures, PROVE, TRY__EACH, and UNIFY, interact extensively. They are the heart of our implementation of Prologo. The partial flowchart in Figure 7-6 clarifies the interrelationships among these procedures. (A full flowchart of the relationships would be large and practically indecipherable.) Instead, we have focused on the points at which key decisions and passage of control between procedures occurs.

Figure 7-6 should provide a good understanding of how Prologo's main procedures work together for the common purpose of confirming information and extracting unknown data from knowledge bases.

### IS__P and UNIFY__IS Procedures

We won't discuss the many procedures Prologo comprises, but we should look briefly at the procedures labeled IS__P and UNIFY__IS. These work with the IS construct, which can be used to define processes in knowledge base rules in Prologo. IS__P checks to see if an IS is present in the axiom being evaluated. If so, it passes a "true" back to the program, which then calls UNIFY__IS to carry out the necessary processing. This processing is similar to that used in the UNIFY procedure discussed earlier.

However, any Logo expression used in conjunction with the IS function is passed to the APPLY procedure with no further processing inside Prologo. We'll see a number of examples of such usage in the next chapter.

## Exploring AI with Prologo

The most obvious way of learning about AI from the Prologo program is to build your own knowledge bases and then make assertions for it to test. Turn on Prologo's goal-axiom trace function by making the variable "trace__prolog" true and print out the resulting output.

Beyond that experimentation, you might undertake an even more ambitious task: enhancing Prologo as a language. ExperTelligence welcomes such enhancements and may even assist you in distributing your ideas to others. A

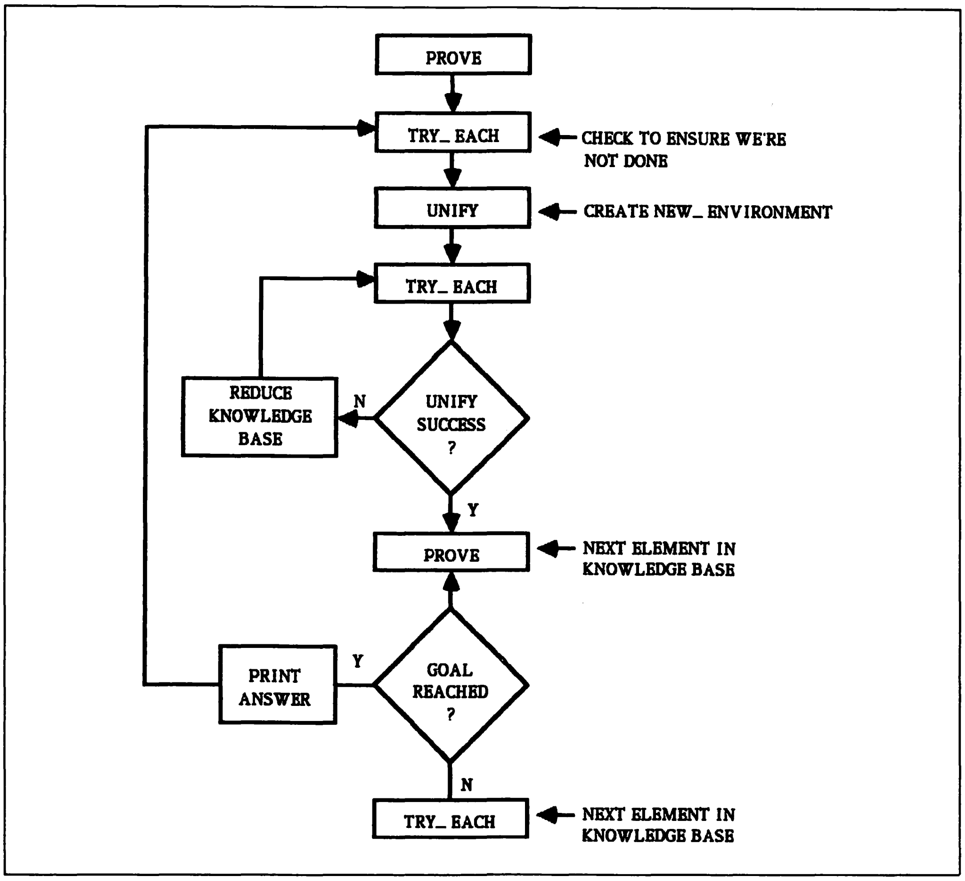


**Figure 7-6. Three-way interaction in Prologo**

wide range of things could be done to extend the power of Prologo. Let's look at four ideas.

**Create Interactive Knowledge Base Generators**

You could design new procedures—called, for example, ADD__CLAUSE, INSERT__CLAUSE, and DELETE__CLAUSE—to build and modify knowledge bases without the necessity of creating MAKE constructs and lists with seemingly endless brackets.

Prolog uses such functions as "add," "insert," and the special "accept" to create and modify knowledge bases. You might even implement the Prolog "save" and "load" functions to have knowledge bases stored on disk for retrieval during Prologo execution.

All knowledge bases are lists of lists. The process of creating and modifying them involves straightforward Logo procedures. But be careful to differentiate Prologo command names from their Logo equivalents or you'll find yourself in a very confusing situation indeed!

**Convert to Prolog Syntax**

Using the parsing ideas that we discuss in the chapters on natural language processing programming techniques, you might construct a parser that permits the use of such traditional Prolog inquiries as "which" and "all" and "is." This would be an extensive modification, and it would require transferring the idea of preceding a variable with an underscore to the inquiry process in a way that is consistent with the design of Prologo.

**Enable Use of All Logo Functions**

Using the IS construction, Prologo rules can adopt most, if not all, Logo commands. But we are unable to use these Logo statements inside assertions. Thus, the assertion:

```
density-of china _what]>[density-of usa
_what2]]
```

won't work in Prologo because the greater-than sign doesn't have a place in assertions.

Adding this capability would require a thorough understanding of the way Prologo handles instantiation of the built-in Logo commands in an IS statement.

**Implement Conjunctive Assertions**

The conjunctive assertion is one powerful feature of Prolog not available in Prologo. Conjunctive assertions involve AND and OR to pose complex assertions like:

```
[[mother-of _who rhoda] and [father-of
_who rhoda]]
```

The program would respond to this query only if the knowledge base contained both parents of Rhoda. Combining instantiation of Logo commands with such conjunctive assertions would go a long way toward converting Prologo to a real Prolog language system.

## Summary

In this chapter we have covered knowledge bases, how Prologo manipulates them and obtains information from them, and how one language is implemented in another language. Along the way, we studied examples of complex search pattern fulfillment in the form of unification, a very important AI concept.

We have seen how backward-chaining works to enable us to track through a search process in a knowledge base and to report findings even though they involve several levels of variable substitution.

```
                {Prologo Program by Steve Lurya}
        {Uncopyrighted program maintained by ExperTelligence}

to prologo :database
  print <<Welcome to Prologo>>
  prologo_1 :database
  <<Back to Logo>>
end
```

```
to prologo_1 :database ;;a top-level loop for Prologo
  ;;reads a form, proves it, and then iterates
  local goal
  make goal (list rename_variables first readlist [0])
  prove :goal :goal [[bottom_of_environment]] :database 1
  print <<no (more) answers>>
  if y_or_n_p <<Another assertion? (y or n)>>
    [prologo_1 :database]
end

to prove :list_of_goals :original_goal :environment :database :level
  ;;proves the conjunction of the list_of_goals
  ;;in the current environment
  if emptyp :list_of_goals
    ;;succeeded since there are no goals
  [print expand_assertion :original_goal :environment
  print_bindings :environment :environment
    ;;ask user if another possibility is wanted
    {not y_or_n_p <<More? (y or n) >>} nil]
  [try_each :database :database
    butfirst :list_of_goals
    first :list_of_goals :original_goal
    :environment :level]
end

to try_each :database_left :database :goals_left :goal :original_goal :environment
:level
  local assertion
  local new_environment
  if and boundp 'trace_prolog :trace_prolog
    [princ <<Goal:>>
    print :goal
    princ <<Current axiom:>>
    print first :database_left]
  if emptyp :database_left
    [[]]  ;;fail if nothing left in database
    [make assertion
      rename_variables first :database_left (list :level)
    if is_p :goal
    [make new_environment
      unify_is :goal :environment
    if emptyp :new_environment
      [nil]
      [prove :goals_left :original_goal :new_environment
        :database (:level + 1)]]
    [make new_environment
      unify :goal first :assertion :environment
    if emptyp :new_environment ;;failed to unify
      [try_each butfirst :database_left
        :database :goals_left :goal
        :original__goal :environment :level]
      [if prove append butfirst :assertion :goals__left
        :original__goal :new__environment
        :database (:level + 1)
        [t]
        [try__each butfirst :database__left
          :database :goals__left :goal
          :original__goal :environment :level]
      ]
    ]
  ]
end

to unify :x1 :y1 :environment
  local x
  local y
  local new__environment
  make x value :x1 :environment
  make y value :y1 :environment
  if cooked__variable__p :x
    [append (list list :x :y) :environment]
    [if cooked__variable__p :y
      [append (list list :y :x) :environment]
      [if (or atom :x atom :y)
        [and (equalp :x :y) :environment]
        [make new__environment
          unify first :x first :y :environment
        (and :new__environment
          unify butfirst :x butfirst :y :new__environment)]
    ]
  ]
end

to value :x :environment
  local binding
  if cooked__variable__p :x
    [make binding (assoc :x :environment)
    if emptyp :binding
      [:x]
      [value (first butfirst :binding) :environment]]
    [:x]
end

to name :var
  and cooked__variable__p :var butfirst first :var
end

to raw__variable__p :var
  and atom :var eq first :var'__
end
```

```
to cooked__variable__p :var
  (and listp :var raw__variable__p first :var numberp level :var)
end

to level :var
  and listp :var first butfirst :var
end

to rename__variables :term :level
  if raw__variable__p :term
    [append (list :term) :level]
    [if atom :term
      [:term]
      [append (list rename__variables first :term :level)
        rename__variables butfirst :term :level]
  ]
end

to print__bindings :environment__left :environment
  local variable
  if butfirst :environment__left
    [make variable first first :environment__left
      if (level :variable) = 0    ;;variable level
        [princ name :variable ;;variable name
        princ << = >>
        ;print value :variable :environment
        print expand__assertion :variable :environment]
      print__bindings butfirst :environment__left :environment]
end

to y__or__n__p :message
  local response
  print :message
  make response readlist
  if eq first :response 'y
    [t]
    [if eq first :response 'n
      [[]]
      [y__or__n__p :message]
    ]
end

to expand__assertion :assertion :environment
  local expression
  if cooked__variable__p :assertion
    [make expression value :assertion :environment
      if eq :expression :assertion
        [:expression]
```

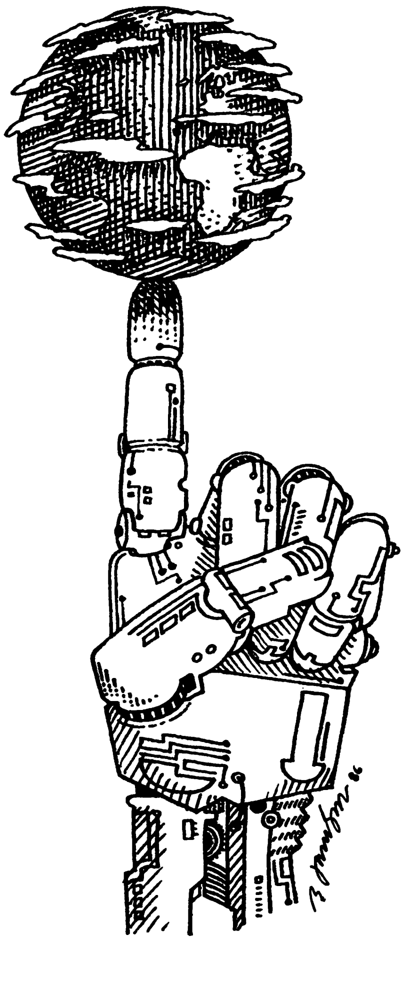
```
          [expand__assertion :expression :environment]]
        [if atom :assertion
          [:assertion]
          [append (list expand__assertion
            first :assertion :environment)
            expand__assertion butfirst :assertion :environment
        ]
    ]
end

to expand__formula :assertion :environment
  local expression
  if cooked__variable__p :assertion
    [make expression value :assertion :environment
  if eq :expression :assertion
    [:expression]
    [expand__formula :expression :environment]]
  [if atom :assertion
    [:assertion]
    [make expression append
      (list expand__formula
        first :assertion :environment)
      expand__formula butfirst :assertion :environment
    if (and fully__instantiated__p :expression
        function__name__p first :expression)
      [apply thing first :expression butfirst :expression]
      [:expression]]
  ]
end

to is__p :goal
  (and listp :goal eq first :goal 'is)
end

to unify__is :goal :environment
  local first__part
  local second__part
  make first__part
    expand__formula first butfirst :goal :environment
  make second__part
    expand__formula first butfirst butfirst :goal :environment
  unify :first__part :second__part :environment
end

to function__name__p :fn
  (and symbolp :fn boundp :fn functionp thing :fn)
end

to fully__instantiated__p :assertion
```

```
  if atom :assertion
    [t]
    [if cooked__variable__p :assertion
      [nil]
      [(and fully__instantiated__p first :assertion
        fully__instantiated__p butfirst :assertion)]]
end

to assoc :x :alist
  if emptyp :alist
    [:alist]
    [if equalp :x first first :alist
      [first :alist]
      [assoc :x butfirst :alist]]
end
```

E ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
INTELLIGENCE ARTIFICIAL
CIAL INTELLIGENCE ARTIFI
RTIFICIAL INTELLIGENCE A
E ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
INTELLIGENCE ARTIFICIAL
CIAL INTELLIGENCE ARTIFI
RTIFICIAL INTELLIGENCE A
E ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
INTELLIGENCE ARTIFICIAL
CIAL INTELLIGENCE ARTIFI
RTIFICIAL INTELLIGENCE A
E ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELL

8

# Two Prologo Knowledge Bases



- How Knowledge Is Acquired
- Defining Knowledge Base Domains
- Focusing Knowledge Bases
- Building Knowledge Bases from Scratch

In this chapter we see the real-life utility of the Prologo program presented in Chapter 7 by building two small expert systems. The first will know about literature: who wrote what, what kind of work a particular title is, and a few other facts. It is the smaller of the two systems, and it enables us to discuss the techniques and concepts involved in designing a knowledge base.

The second expert system will be a geography whiz that can tell us such facts as population densities of countries, relative locations of national capitals, and continental locations of cities and countries. In developing this knowledge base, we will examine a few of the more technical and specialized issues involved in knowledge base creation.

## Knowledge Acquisition: The Key Roadblock?

Dr. Edward Feigenbaum of Stanford University suggests that the biggest stumbling block to the creation of meaningful and useful expert systems is the problem of knowledge acquisition, a problem he termed the "Knowledge Acquisition Bottleneck." As we will see in building even relatively small expert systems, significant time, energy, and thought are required for the construction of knowledge bases.

To be useful, the acquisition of knowledge involves more than simply adding facts to a knowledge base. Whether we are discussing human or machine knowledge, knowledge acquisition involves *relating* something we are learning to what we already know. This integrative process is often quite complex, particularly in the human learning process.

But for our limited examination of the subject, we restrict our meaning of "knowledge acquisition" to the process of adding new factual information to the knowledge base of the Prologo system. To be sure, we will almost always *express* such information in the form of relationships, since this is how Prologo organizes and deals with facts. But we will not permit the program to integrate the new information; instead, we will explicitly provide the relationship data as we enter the new facts.

### What Experts Know

An expert system's job is to emulate the process by which a human expert analyzes problems and provides answers to questions. Before an expert system can function, it must be given knowledge similar to that upon which human experts base their decisions and recommendations. The process of transferring information from humans to computer programs is referred to as "knowledge engineering," a field that is certain to be one of the best growth opportunities for employment in the next 10 years.

As you might imagine, transferring knowledge from human to machine is not a simple process. The problems are myriad and we cannot hope to analyze all of them here. But we will look briefly at some of the more significant barriers that knowledge engineers must overcome.

### You Don't Know What You Know

One of the most intriguing problems faced by knowledge engineers is that experts don't know what they know. That is, on a conscious level, they are not aware of the knowledge, rules, and relationships they use in making judgments. They cannot completely describe the process of getting from a question to a conclusion or recommendation.

That doesn't affect human experts' usefulness as consultants, but as sources of information for knowledge engineering purposes, experts must be able to think about what information they need to get from one point to another.

Consider an example. Your car is not running right. It makes a strange noise when you slow down and sometimes it stalls. You take it to the expert mechanic at your local garage. He starts the car, opens the hood, moves a part back and forth, listens to the engine, and says, "You probably have a clogged fuel line." Curious, you ask him, "How did you know that?" If he's inclined to explain, his answer will probably be something like this: "Well, I pulled on the throttle and listened to the air sound going into the carburetor and concluded there was too much air and not enough fuel getting to it. The major reason for reduced fuel flow is a clogged fuel line."

If you pursue the point, you'll have to ask such questions as, "How did you know that the throttle was the right part to manipulate? How did you know that the carburetor was a likely place to look for the problem? How is the air going into the carburetor *supposed* to sound? How did you eliminate other possible causes of lack of fuel flow?"

True experts will probably have answers for most questions we raise about their fields of expertise, but they can't tell us what the questions need to be. In other words, the mechanic won't have thought in a conscious way about all the things he needs to know and do in order to troubleshoot your car.

The problem of not knowing what we know is not confined to experts, of course. Try explaining to someone who has never done so how to start a car, put it into gear, back it out of the driveway, and drive it to the store.

Much of what we know is buried information, subconsciously retrieved, so that extracting it intentionally to build it into an expert system is quite a task. In the course of constructing a knowledge base, the knowledge engineer must be able to determine when pieces of data are missing and what questions to ask to fill in those blanks.

## How Experts Think

Beyond the issue of *information*, there is the difficulty of stating the *process* by which the experts massage the information they have to reach their conclusions. In short, experts aren't conscious of how they think about a problem.

Superficially, this difficulty seems identical to that of submerged knowledge, but in reality it is quite different, particularly when viewed from a computer program's viewpoint. Simply having pieces of information labeled "A," "B," and "C" in a program running on a computer doesn't tell us what to do with that information. We need to know, among other things, how the pieces of data are interrelated, if they are; where they fit into the overall picture of problem-solving; and how reliable they are.

Your friendly mechanic, for example, might throw a wrench at you if you asked him to explain how his billions of neurons "knew" how to connect with one another to set up the right path so that he could solve the problem of why your car wasn't functioning correctly. On only a slightly less abstract level, you might ask him to describe the process of elimination he goes through as he first looks at a car, listens to its owner describe the problem, opens the hood, listens to the engine, and so on. He may not be conscious of the thousands of options he rejects as he narrows his focus to a method or approach to solving the problem, let alone the process by which this narrowing occurs in his mind.

From the computer's perspective, in such traditional programming languages as BASIC and Pascal, and even LISP, the knowledge is stored as information in an array, list, or other data structure. The process of dealing with that data is what the program is all about. That is not the case in Prolog—and, by extension, in Prologo. Knowledge engineers can focus all their attention and energy on the knowledge and relationships among various aspects of the knowledge and let the programming language take care of drawing inferences and conclusions. That is a primary reason Prolog is such a powerful language for expert systems design.

## The Experts Disagree

Another difficulty for knowledge engineers is that there are no absolute answers in many areas of human knowledge. If we consult three different experts, we might end up designing three different expert systems. The less precise the arena of knowledge, the more likely this disagreement will occur.

Such subjective fields as art appreciation, literary criticism, and the interpretation of historical events involve such disagreement. But even seemingly objective fields pose similar problems. Three mathematicians from diverse backgrounds will not disagree on the basic concepts of numbers, numerical symbols, and mathematical rules. But those same mathematicians might have quite divergent views on aspects of such issues as zero, infinity, and four-dimensional space, all of which are also concepts in mathematics.

Similarly, a group of attorneys may agree on the definition of what a contract is and yet draw strongly differing—and perfectly plausible—conclusions about whether a given document is or is not a contract.

So what do knowledge engineers do when faced with such disagreements? They have, in essence, three options.

***The Top Expert*** They can decide to let one expert be the person whose knowledge becomes the expert system's knowledge. This is particularly suitable if the expert is preeminent in his or her field or, better, the only "real" expert, the others having learned from her or him. (This is not as unlikely as it might seem. Westinghouse, for example, had one senior engineer who understood locomotive repair so much better than anyone else in the company that his expertise was used to build an expert system to be used by all of the other engineers.)

When one person is the acknowledged top expert in a field and confusion or disagreement is minor or unimportant to the design process, this approach may work quite well.

However, an inherent danger is that one person may hold some basic misconceptions that could creep into the expert system, flawing its ability to provide sound advice.

There is one other problem with this approach. When we get two or more experts together to discuss what they know and how they reach conclusions, we are more apt to obtain a complete and clear picture of the knowledge base to be constructed. This is true partly because each will fill in gaps in the other's knowledge, and, even more important, because in challenging one another, the experts will draw out answers to questions that the knowledge engineer will not have thought to ask.

***Consensus Expertise*** The second way for the knowledge engineer to deal with the conflicting experts' views is to continue to talk with them until something resembling consensus emerges. This will occasionally happen, perhaps even to the surprise of the experts. More often, though, it will not happen; the experts will continue to disagree, and the knowledge engineer will be stuck with a problem that appears to be insoluble.

Even when it can be reached, consensus is not necessarily the best view. In the same way that a camel has been said to be a horse designed by a committee, consensus may be a compromise of the best views on a subject. The tendency in such situations is for the experts to reach the lowest common denominator position because it is the least risky and least likely to lead to further disagreement.

Consensus knowledge might be, however, the only alternative when the knowledge needed is imprecise and experts of equal stature and background cannot agree.

***Weighting the Views*** If experts differ and if the views of each are important, the knowledge engineer can take a third road of giving each expert's views a weighted value in the expert system. This would be the equivalent of a human in need of expert consultation bringing the problem to several experts and then drawing a conclusion based on an evaluation of their credibility and experience.

The weighting of the certainty of various conclusions is easier to obtain from people than you might imagine. It is also more difficult to implement on a computer than you might think.

If you asked your auto mechanic how certain he is that your fuel line is clogged, it's fairly predictable that he won't say he's 100 percent confident. Other possibilities could lead to the same symptoms. So he'd probably say something like, "I'm 90 percent certain."

Most human experts, particularly in fields that allow some imprecision (as most do), are more or less certain about their conclusions. A doctor will say, "About 80 percent of the time when I see that combination of symptoms, I know the disease I'm dealing with is a cancer."

## It's Not That Simple

Dealing with information that is not clearly right or wrong ("fuzzy" knowledge) is a complex problem. (I mention it here so that you won't conclude from our far more limited experimentation later in this chapter that knowledge engineering is a cakewalk in which we simply look up facts someplace and stick them into a knowledge base. That is what we will do in this chapter, but only because of limitations of space, time, and expertise.)

## Transferring Expertise

The expert in the knowledge engineering process may be a human, a book about the subject, or our own storehouse of expertise. In all of these, the process of transferring the expertise to the computer is quite similar.

Among our first steps is the process of refining, defining, and focusing the expert system's domain of expertise. That process is similar to preparing to write a report or a book. We start with a broad, perhaps vague idea of what we want to write about. Then we look at how big that assignment would be, narrow our focus, and figure out how much valuable information we could convey to someone in a certain number of pages.

Let's say that we want to write a book about literature. That's a pretty broad subject; tens of thousands of books have been written on it already. That is not to say we cannot write a single book on the general subject of world literature; many textbooks are just that kind of book. But it is to say that if we want a student to emerge from having read the book feeling as an expert on the subject, we're going to have to either focus our topic or write a very long book.

So we decide to focus on American literature. That's still pretty broad, so we narrow it to twentieth century American literature. Now we're down to handling a few hundred major novelists, poets, playwrights, essayists, and journalists. That's still pretty broad for a book of, say, 500 pages. Gradually, we define our domain of expertise and end up with a book entitled, *The Novels of Ernest Hemingway*. Now we can write a fairly definitive (i.e., expert) book on this subject in a few hundred pages (assuming we know this subject or know someone who does).

Figure 8-1 depicts the refining process as a series of sieves through which each idea is filtered until we reach the point where the amount of information to be dealt with is manageable.

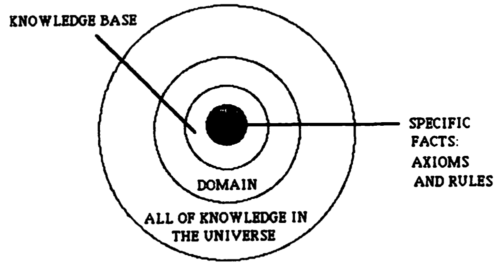

KNOWLEDGE BASE
SPECIFIC FACTS: AXIOMS AND RULES
DOMAIN
ALL OF KNOWLEDGE IN THE UNIVERSE

**Figure 8-1. Refining the domain of knowledge**

In effect, this process takes two steps. The first, about which we don't think too much as a general rule, is defining the domain of expertise. We are going to write about literature rather than about politics. The second, about which we do a great deal of thinking if we are successful in presenting a usable finished product, is the restriction of the knowledge base—American literature of the twentieth century and more specifically, novelists, and still more specifically, Hemingway.

As we construct the two knowledge bases in this chapter, we will take both steps and explain some of the thinking that goes into this process.

## Constructing Knowledge Bases: General Principles

Let us move on to the design and implementation of knowledge bases that Prologo can manipulate and use. As we do so, we will focus more sharply on the issues of knowledge engineering and acquisition as they affect our experiments.

### Defining the Domain

The first step in constructing a knowledge base is to define the domain of expertise within which the system will operate. We have already made some general observations about this subject, but now we'll concentrate on the computer-related decisions involved.

The myth that expert systems are only capable of being expert about one subject is not true. Single expert systems may know about any two domains of knowledge. For example, there is no reason—other than logic and reason—that we can't put the family tree and the world population knowledge bases from Chapter 7 into one MAKE statement. We could then ask questions both about population density and specific peoples' parenthood and grandparenthood from one knowledge base.

Such a combined knowledge base would, of course, have little or no *practical* value. Similarly, it may be that an expert system with expertise in more than one area might not be a marketable, usable, or practical product. But, because there is no *theoretical* limit to the number of topics about which we can construct a Prologo knowledge base, it is important to focus on the practical issue of domain definition as the first step in designing the knowledge base.

***Key Considerations*** Several critical issues must be addressed in domain definition. The most obvious and important is of the expectations of users of the expert system. What do they want to know? We generally do not create expert systems in a musty laboratory somewhere and then spring them on the world. The world might not be interested in an expert system that would classify the small Hermes-head postage stamps of Greece from the second half of the nineteenth century, no matter how efficiently or effectively it did so. But if you designed expert systems for a living and had a philatelist friend who would pay handsomely for such a system and the knowledge were available to you, you might be well advised to construct such a knowledge base.

The second issue determining how we define the domain is knowledge availability and accessibility. If all the information we need in order to construct a specific knowledge base is not known, we will find ourselves with an incomplete knowledge base and perhaps a nonfunctional expert system.

Beyond these two overriding considerations, however, other issues remain to be examined. Some of the more important are:

1. How many axioms and rules will be required to represent all of the knowledge in the domain?
2. How certain can we be of information in the domain?
3. How many conditional parts will meaningful questions addressed to the knowledge base have?
4. How important will a correct response be?
5. Will the user be knowledgeable enough to understand when an erroneous answer is received?

As you can see, these questions relate to the quantity and quality of knowledge and the use to which the expert system will be put.

**Restricting the Knowledge Base**

Even after we have specifically limited the *domain* of knowledge, we will probably face the additional task of restricting the amount and kinds of information we will store and analyze. For example, let's assume that we have decided to design an expert system on Hemingway's novels. Here is a partial listing of the kinds of information we could keep track of on this seemingly narrow topic:

Title
Year of publication
Publisher
Number of pages
Color of book cover
Number of printings
Foreign publications
Movie made
Main characters
Geographic setting(s)
Number of words
Number of uses of four-letter words
Price of book
Number sold
Weeks on *The New York Times Best-Seller List*
Sequence number (which novel for Hemingway)
Names of secondary characters
Number of male vs. female characters
Mentions of Madrid

You get the idea. Within the knowledge domain, we will select information to be stored, tracked, and analyzed by the expert system based on several criteria. The key considerations will include at least the following:

- purpose of the system
- knowledge level of users
- computer storage capacity
- language capabilities and limitations
- complexity of interrelationships of data.

If the point of the expert system is to answer trivia questions about Hemingway's novels, we will choose to include more information of more different kinds than if we are interested in a semantic analysis. If, on the other hand, our expert system is designed to analyze a given piece of writing and determine whether it is *likely* that it was produced by Hemingway, perhaps we would store a small number of *kinds* of data, but include a large number of pieces of information about each work.

The more knowledgeable our user is, the less information that is generally known must be stored. For example, if our users are all English professors, storing the fact that *For Whom the Bell Tolls* was written by Hemingway should be largely unnecessary. On the other hand, if the system is designed to help *stu-*

*dents* learn more about Hemingway and his style, that piece of information may be essential.

Similarly, if we develop our expert system on a computer with a megabyte of memory, we may choose to store information of marginal value, whereas if we are limited to the once-standard 64K, we may well be quite selective and store only essential knowledge.

If the language we are working with has a facility for dealing with relationships among pieces of information, we may want to store larger amounts of related data than if such relationships are difficult to depict.

Finally, if some bits of knowledge that we might include are only remotely related to other pieces of information, we might omit them because it is not likely that we will be able to frame intelligent questions requiring that information.

Numerous other such considerations, of course, are involved in restricting the knowledge base. As you gain experience developing expert systems you will undoubtedly meet dozens of other situations in which trade-off decisions about what to include and what to omit must be made.

**Structuring the Knowledge**

Having defined the domain and restricted the knowledge base, the next decision typically will be about the knowledge base structure. Quite often, this is dictated by the choice of language (a topic discussed at length in Part III).

For us, the decision about knowledge representation, covered extensively in Chapters 2 and 3, is relatively simple. We use lists because Logo has good list-handling capability.

To be more specific, as we will see shortly, we actually use lists of lists of lists to represent knowledge bases in Prologo. The fill-in-the-blanks format for a Prologo knowledge base construct will look something like this:

```
MAKE WORLD_DBASE
  [[[axiom1 axiom1a axiom1b]]
  [[axiom2 axiom2a axiom2b]]
  [[axiom3 axiom3a axiom3b]]]
```

Notice that each axiom in the listing consists of several parts. (If this seems confusing, please stop and review Chapter 7's discussion of Prologo's formatting of knowledge bases.) Each axiom is a list enclosed in two sets of square brackets, except that the first axiom is preceded by three brackets and the last axiom is followed by three, resulting in the construction we described as a list of list of lists.

Each axiom is a list of lists (because it is enclosed in *double* brackets) and the entire knowledge base is a list of these lists.

Why go into such depth about the structure of the knowledge base in Prologo? When we construct the two sample knowledge bases in this chapter, you want to be comfortable with the format so that we can concentrate on the content rather than the structure.

**Importance of Sequence in Prologo**

As we saw in Chapter 7 when we examined the Prologo program listing in detail, Prologo and Prolog analyze knowledge bases by *back-tracking*. This technique means that each time the knowledge base adds a new axiom to its list of things to be proved, it starts over at the beginning of the knowledge base.

As a result, it's a good idea to structure a Prologo knowledge base so that the axioms and rules appear in the sequence in which we expect most queries to be asked. This is not required, but is a good design idea. Thus all the information about the *titles* of Hemingway's novels should appear first in the knowledge base:

```
MAKE HEMINGWAY_DB
  [[[For_Whom_The_Bell_Tolls Title_Of Novel]]
  [[Old_Man_And_The_Sea Title_Of Novel]]
  . . .
```

In full-blown implementations of Prolog, incidentally, the process of grouping similar data together happens automatically. This enables Prolog to stop searching for proof of a statement or query when it reaches the last item of the type for which it is searching. Prologo does not include this design feature in its current implementation.

It will make our knowledge bases easier to understand and modify if we follow this sequencing principle even though Prologo will not execute any more quickly as a result.

## Querying Knowledge Bases

Once we have constructed knowledge bases, we call on Prologo to compile them and then we can make inquiries about their contents. This process of querying the knowledge base is at the heart of an expert system. In Chapter 7 we explored in detail this process as it relates to Prologo. In Chapter 12 the full-blown Prolog language is discussed in some detail.

It will be useful now to consider briefly the kinds of questions we can and cannot expect our Prologo knowledge bases to understand and respond to, since this will bear on our design decisions.

### Questions We Can Ask

We can ask Prologo any questions that are stated as axioms in our knowledge base. We can also ask questions whose answers can be inferred from axioms and rules in the knowledge base. For example, we could ask questions about grandparent relationships, in our family example in Chapter 7, even though such relationships are not directly defined, because we had a rule defining grandparenting.

### Questions We Cannot Ask

There is actually only one type of question we cannot ask in Prologo that we can ask in Prolog or other expert system languages. As we saw at the end of Chapter 7, Prologo does not implement conjunctive inquiries. We cannot ask questions about things related by the word "and" or by the word "or." For example, we cannot ask of our Hemingway knowledge base, "Which Hemingway novels mention Madrid and have mostly female characters?" even if we phrase that as a Prologo query.

Other than this limitation—which is fairly significant in terms of real-life expert systems—we can inquire anything of the knowledge base that it knows about or can conclude.

## The Literary Knowledge Base

The time has come to put all of this theory—and our knowledge of Prologo—to practical use. In the rest of this chapter, we will design, construct, and use two small knowledge bases. To get maximum benefit from the rest of this chapter, you should have Prologo running on your Macintosh® and type in the knowledge bases and queries as we go.

### The Domain: Two American Writers

Beginning with the idea that we wanted to create a knowledge base in the general subject area of literature, we began to narrow our field a bit at a time. We moved from literature to American literature to twentieth century American literature to fiction. Finally now, in the interest of time, we will focus on two modern American writers of some renown: Ernest Hemingway and John Steinbeck.

### The Knowledge Base: Purposely Limited

We will further restrict the knowledge base by choosing to tell it only about a sampling of the works of these two writers. For each work, the knowledge base will contain the type (novel, story, play, poem, and so on) and the year of publication.

Stopping there would result in our having created a data base management system but not an expert system, so we will add a few rules about what are "early" and what are "major" works of the two writers. Fortunately, they were contemporaries, so the time-related terms can be used somewhat interchangeably.

We will also store some basic information about the authors, just to make inquiries a bit more interesting.

### Constructing the Knowledge Base

As we know, building the knowledge base in Prologo consists of creating a MAKE command with a list of list of lists as its argument. The following represents the first pass at the knowledge base we are creating for the literary expert system. In an effort to simplify its entry and use, we will not yet include any rules in this version; later, we will modify it.

```
MAKE LITERARY_DB
  [[[HEMINGWAY HOME ILLINOIS]]
  [[STEINBECK HOME CALIFORNIA]]
  [[HEMINGWAY DOB 1899]]
  [[STEINBECK DOB 1902]]
  [[SUN_ALSO_RISES TYPE NOVEL]]
  [[SUN_ALSO_RISES WRITTEN_BY HEMINGWAY]]
  [[SUN_ALSO_RISES PUBLISHED 1926]]
  [[SUN_ALSO_RISES CLASS MAJOR]]
  [[FAREWELL_TO_ARMS TYPE NOVEL]]
[[FAREWELL_TO_ARMS WRITTEN_BY HEMINGWAY]]
[[FARWELL_TO_ARMS PUBLISHED 1929]]
[[FAREWELL_TO_ARMS CLASS MAJOR]]
[[DEATH_IN_THE_AFTERNOON TYPE NOVEL]]
[[DEATH_IN_THE_AFTERNOON WRITTEN_BY HEMINGWAY]]
[[DEATH_IN_THE_AFTERNOON PUBLISHED 1932]]
[[DEATH_IN_THE_AFTERNOON CLASS MINOR]]
[[FIFTH_COLUMN TYPE PLAY]]
[[FIFTH_COLUMN WRITTEN_BY HEMINGWAY]]
[[FIFTH_COLUMN PUBLISHED 1938]]
[[FIFTH_COLUMN CLASS MINOR]]
[[FOR_WHOM_THE_BELL_TOLLS TYPE NOVEL]]
[[FOR_WHOM_THE_BELL_TOLLS WRITTEN_BY HEMINGWAY]]
[[FOR_WHOM_THE_BELL_TOLLS PUBLISHED 1940]]
[[FOR_WHOM_THE_BELL_TOLLS CLASS MAJOR]]
[[OLD_MAN_AND_THE_SEA TYPE NOVELLA]]
[[OLD_MAN_AND_THE_SEA WRITTEN_BY HEMINGWAY]]
[[OLD_MAN_AND_THE_SEA PUBLISHED 1952]]
[[OLD_MAN_AND_THE_SEA CLASS MAJOR]]
[[KILLERS TYPE STORY]]
[[KILLERS WRITTEN_BY HEMINGWAY]]
[[KILLERS PUBLISHED 1934]]
[[KILLERS CLASS MINOR]]
[[NOBEL_PRIZE HEMINGWAY 1954]]
[[PULITZER_PRIZE HEMINGWAY 1953]]
[[PASTURES_OF_HEAVEN TYPE NOVEL]]
[[PASTURES_OF_HEAVEN WRITTEN_BY STEINBECK]]
[[PASTURES_OF_HEAVEN PUBLISHED 1932]]
[[PASTURES_OF_HEAVEN CLASS MINOR]]
[[TORTILLA_FLAT TYPE NOVEL]]
[[TORTILLA_FLAT WRITTEN_BY STEINBECK]]
[[TORTILLA_FLAT PUBLISHED 1935]]
[[TORTILLA_FLAT CLASS MAJOR]]
[[IN_DUBIOUS_BATTLE TYPE NOVEL]]
[[IN_DUBIOUS_BATTLE WRITTEN_BY STEINBECK]]
[[IN_DUBIOUS_BATTLE PUBLISHED 1936]]
[[IN_DUBIOUS_BATTLE CLASS MINOR]]
[[GRAPES_OF_WRATH TYPE NOVEL]]
[[GRAPES_OF_WRATH WRITTEN_BY STEINBECK]]
[[GRAPES_OF_WRATH PUBLISHED 1939]]
[[GRAPES_OF_WRATH CLASS MAJOR]]
[[SEA_OF_CORTEZ TYPE NONFICTION]]
[[SEA_OF_CORTEZ WRITTEN_BY STEINBECK]]
[[SEA_OF_CORTEZ PUBLISHED 1941]]
[[SEA_OF_CORTEZ CLASS MINOR]]
[[NOBEL_PRIZE STEINBECK 1962]]
[[PULITZER_PRIZE STEINBECK 1940]]]
```

This will suffice for the first version of our literary knowledge base. (See Table 8-1 for a summary of our information.) We will add to it later.

**Table 8-1. Literary Knowledge Base**

| *Author* | *Year* | *Publication Title* | *Type* | *Classification* | *Other Event of Note* |
|---|---|---|---|---|---|
| Hemingway | 1899 | | | | Birth (Illinois) |
| | 1926 | Sun_Also_Rises | Novel | Major | |
| | 1929 | Farewell_to_Arms | Novel | Major | |
| | 1932 | Death_in_the_Afternoon | Novel | Minor | |
| | 1934 | Killers | Story | Minor | |
| | 1938 | Fifth_Column | Play | Minor | |
| | 1940 | For_Whom_the_Bell_Tolls | Novel | Major | |
| | 1952 | Old_Man_and_the_Sea | Novella | Major | |
| | 1953 | | | | Pulitzer_Prize Awarded |
| | 1954 | | | | Nobel_Prize Awarded |
| Steinbeck | 1902 | | | | Birth (California) |
| | 1932 | Pastures_of_Heaven | Novel | Minor | |
| | 1935 | Tortilla_Flat | Novel | Major | |
| | 1936 | In_Dubious_Battle | Novel | Minor | |
| | 1939 | Grapes_of_Wrath | Novel | Major | |
| | 1940 | | | | Pulitzer_Prize Awarded |
| | 1941 | Sea_of_Cortez | Nonfiction | Minor | |
| | 1962 | | | | Nobel_Prize Awarded |

## Querying the Knowledge Base

Now that we have constructed this knowledge base with 56 items of information in it, we can make inquiries about the things it knows.

***Questions About Authors*** We might want to know something about Ernest Hemingway. A dialog like the following might take place (computer responses are indented throughout the rest of this chapter).

```
[_WHAT WRITTEN_BY HEMINGWAY]←our query
[SUN_ALSO_RISES WRITTEN_BY HEMINGWAY]
  WHAT = SUN_ALSO_RISES
  [FAREWELL_TO_ARMS WRITTEN_BY HEMINGWAY]
  WHAT = FAREWELL_TO_ARMS
  [DEATH_IN_THE_AFTERNOON WRITTEN_BY HEMINGWAY]
  WHAT = DEATH_IN_THE_AFTERNOON
  [FIFTH_COLUMN WRITTEN_BY HEMINGWAY]
  WHAT = FIFTH_COLUMN
  [FOR_WHOM_THE_BELL_TOLLS WRITTEN_BY HEMINGWAY]
  WHAT = FOR_WHOM_THE_BELL_TOLLS
  [OLD_MAN_AND_THE_SEA WRITTEN_BY HEMINGWAY]
  WHAT = OLD_MAN_AND_THE_SEA
  [KILLERS WRITTEN_BY HEMINGWAY]
  WHAT = KILLERS
  no (more) answers
  Another assertion? (y or n)
Y
```

Or we might want to know something more about the authors, such as the years they won the Nobel Prize.

```
[NOBEL_PRIZE _WHO _WHEN]
  [NOBEL_PRIZE HEMINGWAY 1954]
  WHEN = 1954
  WHO = HEMINGWAY
  [NOBEL_PRIZE STEINBECK 1962]
  WHEN = 1962
  WHO = STEINBECK
  no (more) answers
  Another assertion? (y or n)
Y
```

***Questions About Works*** Let's obtain a list of all the major works in the knowledge base:

```
[_WHAT CLASS MAJOR]
  [SUN_ALSO_RISES CLASS MAJOR]
  WHAT = SUN_ALSO_RISES
  [FAREWELL_TO_ARMS CLASS MAJOR]
  WHAT = FAREWELL_TO_ARMS
  [FOR_WHOM_THE_BELL_TOLLS CLASS MAJOR]
  WHAT = FOR_WHOM_THE_BELL_TOLLS
  [OLD_MAN_AND_THE_SEA CLASS MAJOR]
  WHAT = OLD_MAN_AND_THE_SEA
  [TORTILLA_FLAT CLASS MAJOR]
  WHAT = TORTILLA_FLAT
  [GRAPES_OF_WRATH CLASS MAJOR]
  WHAT = GRAPES_OF_WRATH
  no (more) answers
  Another assertion? (y or n)
Y
```

**Adding Some Rules**

So far, so good. The problem is that what we have so far is not really acting like an expert system—because we have not given the knowledge base anything but data; it needs rules from which to draw conclusions. Let's add two rules to the knowledge base.

The first rule will define a "big book" as a novel that is a major work. Go to the end of the current knowledge base, delete the third closing bracket at the end of the last line so that the last line ends with two brackets, like all of the other facts in the knowledge base, and type the following rule:

```
[[BIG_BOOK _TITLE]
  [_TITLE TYPE NOVEL]
  [_TITLE CLASS MAJOR]]]
```

After we run this knowledge base so that the LITERARY_DB includes this rule, we can make inquiries that require the knowledge base to draw some inferences:

```
[BIG__BOOK __TITLES]
  [BIG__BOOK SUN__ALSO__RISES]
  TITLES = SUN__ALSO__RISES
  [BIG__BOOK FAREWELL__TO__ARMS]
  TITLES = FAREWELL__TO__ARMS
  [BIG__BOOK
FOR__WHOM__THE__BELL__TOLLS]
  TITLES = FOR__WHOM__THE__BELL__TOLLS
  [BIG__BOOK TORTILLA__FLAT]
  TITLES = TORTILLA__FLAT
  [BIG__BOOK GRAPES__OF__WRATH]
  TITLES = GRAPES__OF__WRATH
  no (more) answers
  Another assertion? (y or n)
Y
```

If we assert that a specific title is a BIG__BOOK, as we know, the system will confirm if it is true and not respond if it is not true. So we could have the following exchange with Prologo:

```
[BIG__BOOK FAREWELL__TO__ARMS]
  [BIG__BOOK FARWELL__TO__ARMS]
  no (more) answers
  Another assertion? (y or n)
Y
[BIG__BOOK SEA__OF__CORTEZ]
  no (more) answers
  Another assertion? (y or n)
N
```

Let's add one last rule to our growing literary knowledge base. This one will require Prologo to carry out a calculation to determine whether a particular conclusion is true or not. It uses the IS feature of Prologo discussed at some length in Chapter 7. The rule tells us about "EARLY__WORK" titles, which we have arbitrarily defined as those that were published in 1935 or earlier.

To add this rule, follow the same procedure as before. Go to the last statement in the file and delete one right bracket so that the rule about BIG__BOOK ends with two right brackets rather than three. Now type the following new rule:

```
[[EARLY__WORK __TITLE]
  [__TITLE PUBLISHED __YEAR]
  [IS [≤ 1935 __YEAR]]]]
```

Notice that we use *four* sets of closing brackets this time instead of three. That's because the statement, [≤1935 __YEAR], which is evaluated by the IS expression in Prologo's syntax, is itself inside a pair of brackets. (Brackets must always balance—there must be exactly as many right brackets as there are left in any expression.)

Now let's ask Prologo to tell us about early works of the two authors.

```
[EARLY__WORK __NAMES]
  [EARLY__WORK SUN__ALSO__RISES]
  NAMES = SUN__ALSO__RISES
  [EARLY__WORK FARWELL__TO__ARMS]
  NAMES = FAREWELL__TO__ARMS
  [EARLY__WORK DEATH__IN__THE__AFTERNOON]
  NAMES = DEATH__IN__THE__AFTERNOON
  [EARLY__WORK KILLERS]
  NAMES = KILLERS
  [EARLY__WORK PASTURES__OF__HEAVEN]
  NAMES = PASTURES__OF__HEAVEN
  [EARLY__WORK TORTILLA__FLAT]
  NAMES = TORTILLA__FLAT
  no (more) answers
  Another assertion? (y or n)
Y
```

**Some Miscellaneous Thoughts**

By now, you're probably getting the hang of knowledge base construction and usage in Prologo. In fact, we've gotten about as complex as is possible in Prologo; additional levels of complexity are just matters of degree. For example, we could devise a rule that would enable us to find all early works that are also considered major titles by the author. Such a rule might look something like this (the bracketing depends on where the rule is placed in the knowledge base):

```
[[EARLY__BIGGIE __TITLE]
  [BIG__BOOK __TITLE]
  [EARLY__WORK __TITLE]]
```

If we had never defined BIG__BOOK and EARLY__WORK, we could enter the same rule as follows:

```
[[EARLY__BIGGIE __TITLE]
  [__TITLE TYPE NOVEL]
  [__TITLE CLASS MAJOR]
  [__TITLE PUBLISHED __YEAR]
  [IS [≤ 1935 __YEAR]]]
```

As we learned in Chapter 7, the more clauses we add after each rule definition (in this example we have four), the more AND combinations we're defining. Here, a book that will fit our definition of an EARLY__BIGGIE must be a novel, major, *and* published during or before 1935.

You can see how we could define more and more complex rules to design a Prologo knowledge base that would look intelligent to anyone making inquiries of it.

Next, we will construct a substantially larger knowledge base dealing with geography and geopolitics. It will incorporate relatively complex rules and do calculations using the IS function in Prologo.

## The Geography Knowledge Base

By the time we're done exploring this, you'll know about AI expert systems—with some information about geography thrown in at no extra charge.

**The Domain: World Geography**

One reason for choosing geography for our second knowledge base is that it permits us to establish and describe *relationships*. Our literary knowledge base showed a limited capacity to do this because of the subject matter; one item did not necessarily "belong" with another. But in this section, we are going to be able to discuss countries and cities as belonging to each other, to continents, and to various other groups. It is in defining and manipulating such related information that Prolog (and, by extension, Prologo) really shines.

**The Knowledge Base**

In the interest of protecting your fingers, your sanity, and the publisher's supply of paper, we'll limit the number of countries and cities in this knowledge base. We have also chosen to restrict our attention to a modest amount of information about each city, country, and continent. But within the context of those restrictions, and given the limits of both our study and our endurance, this knowledge base comes close to being a real expert system.

**Constructing the Knowledge Base**

Since we've already built one relatively substantial knowledge base, let's just dive right into this one. Type the following MAKE statement in your ExperLogo® environment, save it, and run all of its contents. Compiling it takes a few moments, so try to be patient!

```
MAKE WORLD_KB←note that we use "KB" for Knowledge Base here
  [[USA POPULATION 234]]
  [[USA AREA 3615]]
  [[WASHINGTON_DC CAPITAL_OF USA]]
  [[WASHINGTON_DC LATITUDE 39]]
  [[USA CONTINENT NORTH_AMERICA]]
  [[USA GOVERNMENT DEMOCRACY]]
  [[USA LANGUAGE ENGLISH]]
  [[VENEZUELA POPULATION 18]]
  [[VENEZUELA AREA 352]]
  [[CARACAS CAPITAL_OF VENEZUELA]]
  [[CARACAS LATITUDE 10]]
  [[VENEZUELA CONTINENT SOUTH_AMERICA]]
  [[VENEZUELA GOVERNMENT REPUBLIC]]
  [[VENEZUELA LANGUAGE SPANISH]]
  [[USSR POPULATION 273]]
  [[USSR AREA 8650]]
  [[MOSCOW CAPITAL_OF USSR]]
  [[MOSCOW LATITUDE 55]]
  [[USSR CONTINENT ASIA]]
  [[USSR GOVERNMENT SOCIALIST]]
  [[USSR LANGUAGE SLAVIC]]
```

```
[[ARGENTINA POPULATION 30]]
[[ARGENTINA AREA 1065]]
[[BUENOS_AIRES CAPITAL_OF ARGENTINA]]
[[BUENOS_AIRES LATITUDE -36]]
[[ARGENTINA CONTINENT SOUTH_AMERICA]]
[[ARGENTINA GOVERNMENT REPUBLIC]]
[[ARGENTINA LANGUAGE SPANISH]]
[[UNITED_KINGDOM POPULATION 56]]
[[UNITED_KINGDOM AREA 94]]
[[LONDON CAPITAL_OF UNITED_KINGDOM]]
[[LONDON LATITUDE 51]]
[[UNITED_KINGDOM CONTINENT EUROPE]]
[[UNITED_KINGDOM GOVERNMENT DEMOCRACY]]
[[UNITED_KINGDOM LANGUAGE ENGLISH]]
[[CHINA POPULATION 1022]]
[[CHINA AREA 3692]]
[[PEKING CAPITAL_OF CHINA]]
[[PEKING LATITUDE 40]]
[[CHINA CONTINENT ASIA]]
[[CHINA GOVERNMENT TOTALITARIAN]]
[[CHINA LANGUAGE CHINESE]]
[[RWANDA POPULATION 6]]
[[RWANDA AREA 10]]
[[KIGALI CAPITAL_OF RWANDA]]
[[KIGALI LATITUDE -23]]
[[RWANDA CONTINENT AFRICA]]
[[RWANDA GOVERNMENT TOTALITARIAN]]
[[RWANDA LANGUAGE FRENCH]]
[[FRANCE POPULATION 55]]
[[FRANCE AREA 210]]
[[PARIS CAPITAL_OF FRANCE]]
[[PARIS LATITUDE 49]]
[[FRANCE CONTINENT EUROPE]]
[[FRANCE GOVERNMENT DEMOCRACY]]
[[FRANCE LANGUAGE FRENCH]]
[[NORWAY POPULATION 4]]
[[NORWAY AREA 125]]
[[OSLO CAPITAL_OF NORWAY]]
[[OSLO LATITUDE 60]]
[[NORWAY CONTINENT EUROPE]]
[[NORWAY GOVERNMENT MONARCHY]]
[[NORWAY LANGUAGE NORWEGIAN]]
[[NIGERIA POPULATION 85]]
[[NIGERIA AREA 357]]
[[LAGOS CAPITAL_OF NIGERIA]]
[[LAGOS LATITUDE 7]]
[[NIGERIA CONTINENT AFRICA]]
[[NIGERIA GOVERNMENT TOTALITARIAN]]
[[NIGERIA LANGUAGE ENGLISH]]
[[WESTERN _COUNTRY]
```

```
    [_COUNTRY CONTINENT NORTH_AMERICA]]
  [[WESTERN _COUNTRY]
    [_COUNTRY CONTINENT SOUTH_AMERICA]]
  [[WESTERN _COUNTRY]
    [_COUNTRY CONTINENT EUROPE]]
  [[POPULOUS _COUNTRY]
    [_COUNTRY POPULATION _COUNT]
    [_COUNTRY AREA _SIZE]
    [IS _DENSITY [/ _COUNT _SIZE]]
    [IS [> _DENSITY .25]]]
  [[BIGGER _C1 _C2]
    [_C1 POPULATION _P1]
    [_C2 POPULATION _P2]
    [IS [< _P1 _P2]]]
  [[_C1 NORTH_OF _C2]
    [_C1 LATITUDE _LAT1]
    [_C2 LATITUDE _LAT2]
    [IS [> _LAT2 _LAT1]]]]
```

***Some Explanatory Comments*** Populations are rounded in millions of people and areas are rounded to thousands of square miles. A negative latitude indicates a location south of the equator.

The choice of labels for governments is somewhat arbitrary. Semantic differences between democracies and republics and between socialist and totalitarian are of less interest than how the knowledge base makes available several kinds of governmental types. In addition to the specific information, our knowledge base also contains four rules.

***Rule 1: WESTERN Countries*** This determines whether a country is part of the Western World—the Americas and Europe. It uses an OR construction by defining WESTERN three times with three different continents as clauses.

***Rule 2: POPULOUS Nations*** This arbitrary rule defines a populous nation as one with more than 250 people per square mile. Note that the rule contains two IS statements. The first calculates a variable called _DENSITY and the next compares the result of this calculation against our arbitrary yardstick 0.25. If the value tests true, the condition is satisfied and Prologo reports the country as populous. In the next section we will discuss the IS constructions and the order in which variables appear in them.

***Rule 3: BIGGER*** We can compare the sizes of two countries to each other, with population as the criterion, using the BIGGER rule. The rule extracts the populations of the two countries, then uses the IS construct to determine if the first is larger than the second.

It is worth noting that the last line of this rule statement, which contains the IS construction, *seems* backward but is not:

```
[IS [< _P1 _P2]]]
```

Once it finds the IS, Prologo scans the IS statements from right to left. So in this case we must put the value we would usually put first in a greater-than test in the second position. This rule expression, then, asks, "Is P2 less than P1?"

***Rule 4: NORTH__OF*** We can determine which cities are north of which other cities by using this simple rule. It extracts the latitude of each of two cities from the knowledge base and compares them. If the first is larger, the city it represents is north of the other city.

This rule demonstrates an important point about Prologo: the rule name—in this case NORTH__OF—need not be the first item in a rule statement. We want to be able to make inquiries that are more English-sounding than many of our other queries tend to be, so we put the city we want to test for "northernness" first, then the rule name, then the city to be tested against.

**Querying the Knowledge Base**

We can ask a wide range of questions about the knowledge base given the amount of data and the four rules it contains. Let's look at some examples of queries we can make.

***Queries About Capitals*** We can find out the capital of any country:

```
[_CITY CAPITAL_OF FRANCE]
  [PARIS CAPITAL_OF FRANCE]
  CITY = PARIS
```

We can get a list of all of the capitals known to the knowledge base, too.

```
[_CITY CAPITAL_OF _COUNTRY]
  [WASHINGTON_DC CAPITAL_OF US]
  COUNTRY = US
  CITY = WASHINGTON_DC
  [CARACAS CAPITAL_OF VENEZUELA]
  COUNTRY = VENEZUELA
  CITY = CARACAS
  .
  .
  .
  [LAGOS CAPITAL NIGERIA]
  COUNTRY = NIGERIA
  CITY = LAGOS
```

We can also identify the country, given the name of the capital. We leave the framing of this query to the user.

***Questions About Continents*** Because we've told the system the continent on which each country is located, we can find countries by continent. To obtain a list of all African nations, for example, we would simply assert the following query:

```
[_COUNTRIES CONTINENT AFRICA]
```

and wait for the system to inform us that Rwanda and Nigeria are African nations.

We can also find out which continent any given country is on.

***Questions About POPULOUS Countries*** Given our rule of 250 people per square mile being populous, we can inquire of the system which countries meet this criterion by querying:

```
[POPULOUS _WHICH]
```

Prologo would scan the knowledge base and inform us that the United Kingdom (density 595.7), China (density 276.8), and France (density 261.9) are populous countries according to our definition of the term.

To test the theory that China is a populous country, you could assert:

```
[POPULOUS CHINA]
```

and Prologo would affirm your assertion by repeating it.

***Questions About NORTHerliness*** Not only can we find out if one city is north of another by such a query as:

```
[WASHINGTON_DC NORTH_OF LONDON]
```

but we can also find out which cities are north or south of any city we pick. To locate all cities north of Paris, for example, we would query:

```
[_CITIES NORTH_OF PARIS]
```

On the other hand, we can find all the cities *south* of Paris without a SOUTH_OF function by the following Prologo query:

```
[PARIS NORTH_OF CITIES]
```

## Exploring AI Further with Prologo

We can ask our WORLD_KB knowledge base a great deal about these countries; you should certainly spend some time querying the knowledge base to see the kinds of responses you can retrieve. (Table 8-2 summarizes the information in tabular form.)

Beyond enhancements to these knowledge bases—by adding axioms (facts) or rules—you should now feel comfortable designing your own knowledge base for Prologo to manipulate. Pick a subject you're interested in and where you have some knowledge or experience. Start drawing up relationships and information about it and build a knowledge base. You might try designing an expert system that can predict the weather, based on such data as temperature, wind direction, amount of moisture the preceding day, and the like.

**Table 8-2. World_KB Knowledge Base**

| *Country* | *Population** | *Area*** | *Language* | *Government* | *Continent* | *Capital City* | *Latitude of Capital†* |
|---|---|---|---|---|---|---|---|
| USA | 234 | 3615 | English | Democracy | North_America | Washington_DC | 39 |
| Venezuela | 18 | 352 | Spanish | Republic | South_America | Caracas | 10 |
| USSR | 273 | 8650 | Slavic | Socialist | Asia | Moscow | 55 |
| Argentina | 30 | 1065 | Spanish | Republic | South_America | Buenos_Aires | −36 |
| United_Kingdom | 56 | 94 | English | Democracy | Europe | London | 51 |
| China | 1022 | 3692 | Chinese | Totalitarian | Asia | Peking | 40 |
| Rwanda | 6 | 10 | French | Totalitarian | Africa | Kigali | −23 |
| France | 55 | 210 | French | Democracy | Europe | Paris | 49 |
| Norway | 4 | 125 | Norwegian | Monarchy | Europe | Oslo | 60 |
| Nigeria | 85 | 357 | English | Totalitarian | Africa | Lagos | 7 |

*in millions
**in thousands of square miles
†in degrees north or south (−) of equator

## What We've Learned About AI

Thus far, expert systems are the only aspect of AI research that have commercial application and value. In this chapter we've explored expert systems and their position in the field of AI. We've seen how expert system development is largely a matter of knowledge acquisition and representation and we have reviewed some of the problems involved.

By designing and constructing two knowledge bases, we can appreciate the difficulty and importance of the design task facing anyone who wishes to construct an expert system.

Along the way, we've solidified our understanding of Prologo and developed greater insight into that language and into Prolog.

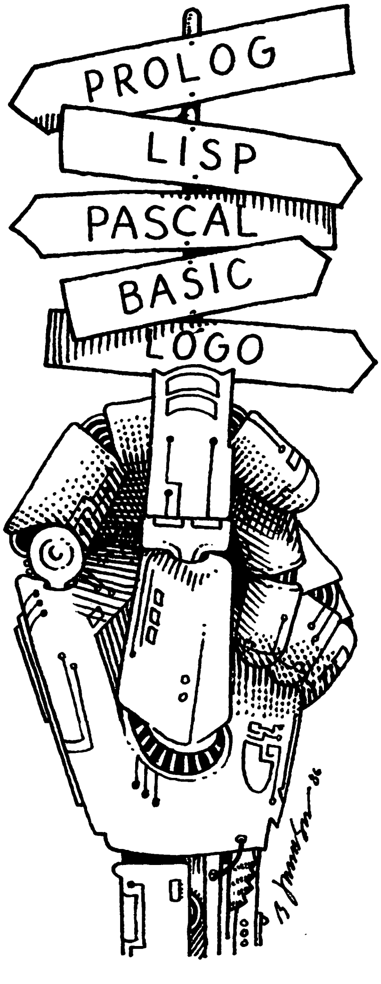
# III

# *Artificial Intelligence Languages*

CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELL
ELLIGENCE ARTIFICIAL INT
L INTELLIGENCE ARTIFICIA
ICIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELL
ELLIGENCE ARTIFICIAL INT
L INTELLIGENCE ARTIFICIA
ICIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELL
ELLIGENCE ARTIFICIAL INT
L INTELLIGENCE ARTIFICIA
ICIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELL

# 9

# *Does the Language Make a Difference?*



- Basic Language Issues in AI
- The Needs of a "Good" AI Language
- How Some Popular Languages Stack Up
- Choosing an AI Language

In this third part of our exploration of AI, we move from the topics of specific programming techniques to the interesting and challenging topics of languages.

This chapter looks at the subject from a broad perspective. In it, we'll examine issues involved in selecting an appropriate AI language, compare the most popular ones, and discuss dialects and offshoot languages. We'll also cover why BASIC (the most popular and widely available language on microcomputers) is not generally thought to be well suited to AI applications.

The remaining three chapters will summarize the essential ideas of each of the most popular programming languages for AI: Logo, LISP, and Prolog. My goal is to permit a person with some programming experience to pick up the main concepts of each language. The chapters are not full-fledged tutorials about the languages involved, but they are sufficiently detailed to serve as good introductions for anyone with programming experience.

Finally, Chapter 13 will examine how a programmer experienced in BASIC might convert the ExperLogo® programs in this book into that language. We'll discuss handling data structures that are peculiar to the list-oriented languages and not available in BASIC as well as the issues of procedural program design and recursion. We'll focus on Microsoft's BASIC 2.0 for the Macintosh®, the only full-fledged implementation of BASIC for the Mac with widespread distribution. Further, it overcomes some of the major obstacles to AI programming in that language.

## Where's the Superlanguage?

If we could have our way, we might wish for one all-powerful language that could do everything we would ever want a programming language to do—handle lists, arrays, matrices, strings, formulas, vectors, relationships, property lists, very large numbers, very small numbers, input/output operations, report generation, and a thousand other tasks and features with equal speed and facility. It would be easy to use, and it would look as much like English (or German or Russian, depending on our native language) as possible.

Such a language will *never* be developed because we can't even begin to envision what it would look like. Even the powerful human natural languages aren't rich enough to do the job. Think how many specialized dialects have developed in English, at least in part because there were not enough words of precise meaning in the standard language. There are sublanguages for doctors, lawyers, scientists, mathematicians, television producers, teenagers, philosophers, among others.

Some of these languages—particularly scientific ones—don't even use English words as their symbols; they use formulas, subscripts, superscripts, and the like. But they are languages nonetheless, because they use symbols to convey meaning, which is what a language does.

Selecting a language for an application, then, is a matter of deciding which features of the hypothetical superlanguage we need and which we can do without. We select a programming language containing the appropriate subset of all of our ideal features.

## What Programming Languages Do. . .and Don't Do

All of the languages we've discussed so far are incomprehensible to the computer on which they run. The computer only understands a peculiar binary language which consists of the symbols 1 and 0. Any High-Level Language (or HLL), as BASIC, LISP, Logo, Pascal, and others are called, is designed to make it easier for us to communicate our ideas and needs to the computer. Somewhere between us and the computer lies a program (or several programs) that translate this human-readable programming code into 1s and 0s for the computer's consumption.

An interesting result of this well known but not often considered fact is that no HLL can do anything that the machine isn't capable of understanding on a very elementary level. If your Mac responds when you click the mouse button on a pull-down menu, it's because someone has spent a lot of time, energy, and thought programming the machine to understand those actions in its own terms.

High-level languages were developed because it was essential for people to communicate effectively and efficiently with the computer. (One wag has suggested that the lower-level languages, which are close to or at the level of the machine's binary understanding, are useful when we want to get maximum speed from the computer and minimum speed from the programmer!)

Another major benefit of the creation of HLLs is that such languages often act as analytical aids, helping us to organize thinking about and analysis of the problem we are trying to solve. By forcing us to think of the problem in its component parts, label them meaningfully, and describe interactions among them precisely, such languages often help us to crystallize our thinking about a problem.

Beyond that, our HLL selection often shapes or even dictates the way we approach a problem or examine its solution. For example, BASIC, because of the way it was designed, lends itself to a lot of FOR-NEXT loops (where we count iteratively through a process and keep track of how many times or under what circumstances we continue to do so). So when we think of solving a problem in BASIC we often find ourselves thinking in terms of such loops, whether or not they are the best way to solve the problem in an abstract sense.

Two factors affect the difficulty in selecting languages.

First, some problems are difficult or impossible to solve using some approaches. For example, if arrays (or tables) aren't available in a language, a table look-up solution to a problem will be difficult (or impossible) to implement in the language. If a problem can only be solved by using a look-up table, the absence of such features may be a sufficient reason in and of itself for rejecting that language for solving the problem.

Second, while there is usually not just one "right" way to solve a particular problem, there is often a "best" way to do so. To the extent that we choose a language that permits us to implement these best solutions, we make things easier for ourselves in programming them.

## What an AI Language Needs

If we put three AI researchers into a room and asked them to list the features of an HLL that are appropriate to the kind of work they do, we would deprive the world of three AI researchers for a long, long time. There would be a great deal of discussion about what an AI language ought to compromise short of being the superlanguage.

But our researchers would take only a few minutes to agree on two features required in an AI language. And two other features would follow soon thereafter, if we allowed a two-out-of-three vote to prevail. The two consensus points would be the need for complex data structures and extensibility. The other two important features would be the ability to treat a program as a data structure and the need for powerful string-handling capability.

Let's take a brief look at why each of these four features is important in an AI language.

**Complex Data Structures**

When we build an accounting program or an educational quiz program, we can predetermine the size of data structures to be accommodated by the program (i.e., number of accounts, largest dollar value represented, number of questions, degree of accuracy of reporting the student's score). We guess on the high side because it is difficult to add to these structures later.

But when we are talking about parsing a sentence, planning the next move in a maze game against an intelligent adversary, or drawing inferences from a knowledge base in Prologo, we don't know in advance what the size and shape of the information will be—or what information we will generate during our analysis. This uncertainty means we must have a data structure that is capable of efficiently handling a large amount of data.

Further (as we have seen in our discussions about knowledge representation), associations and relationships among bits of information are sometimes more important than the information itself. An AI programming language, to be useful and efficient, should permit such representation of knowledge.

***Lists as a Solution*** From the mid-1950s (when LISP was developed), lists have been accepted as the proper data structure for AI tasks. Lists are capable of continual expansion by definition; we need not declare their size in advance or limit their size at any point. Items can be paired or associated within and between lists without predefining what those relationships might be or become. Property lists are an outstanding example of the power of list processing.

Beyond the innate power of a list as a data structure, the list manipulating languages like LISP and Logo have built-in instructions that make it very easy to extract, compare, analyze, modify, and store information stored in lists.

**Extensibility**

Extensibility means defining new terms in old words. With this capability, a language can overcome some of its deficiencies in terms of how we interact with it. A good example of this occurs in the Missionaries and Cannibals program in Chapter 2. We needed a word that meant we had solved the problem. We chose, logically enough, "success." Unfortunately, Logo does not understand that word at all. So we added the word "success" to Logo's vocabulary for the purpose of this program.

Again, Micro Blocks World (Chapter 5) required words for blocks, pyramids, and rearranging the tabletop, so we created them. This makes it possible for us to say things like REARRANGE TABLE, which is certainly clearer than GOSUB 9000, even though they may both have the same effect.

More important, having defined a new Logo word for a specific program's needs, we can use it in other procedures. This makes such languages as LISP and Logo extensible (Forth is also extensible in this way, incidentally).

It may have occurred to you that if a language were extensible enough, you could write another whole language in it. AI researchers have been doing it for years. They have created dozens of new languages that are written in LISP (there are even a few Prologs written in LISP and a LISP or two written in another dialect of LISP!). These other languages have names like PLANNER, CONNIVER, and IRIS, and their purpose is to permit experimentation with specific domains of problems and types of input.

### Programs as Data Structures

As they begin to learn LISP, one of the most difficult ideas to grasp for programmers with experience in BASIC is that the LISP program is a data structure. In BASIC, this is simply not the case; the program and the data are distinct and separate from one another. A BASIC program doesn't even *look* like a data structure: it is a collection of commands delineated (usually) by line numbers and it lacks inherent structure.

But LISP programs are definitely data structures. It is thus possible to have a LISP program that modifies itself—since it is just another data structure—and thus modifies its behavior based on its "experience." In an AI environment, where one objective is to make programs more humanlike, the ability of the program to adapt as it is used is certainly attractive.

### String-Handling Power

Most knowledge that we work with in AI is not numeric but includes ideas, concepts, interrelationships, and expressions of language. Thus the ability to manipulate strings—store them efficiently, extract information from and about them, modify them, and combine them—is one that most AI researchers would probably agree is important to a language used for AI research.

## Popular Languages and AI

With all of this background, we still have to face the question of which programming language to choose for our AI work.

The choice is arbitrary; if we are willing to put up with inefficiencies and implementation difficulties, we may choose virtually any microcomputer programming language and eventually develop an AI program with it.

Computer consultants are frequently asked by clients questions like, "Can Computer X run my Y-square-foot warehouse?" Sometimes, depending on which computer is in the sentence, and how large the warehouse is, the answer will be, "Sure. You can use that computer to do that. You could also use a motorcycle as a moving van if you wanted to hook a trailer to it, but that doesn't mean it's the best solution." The issue for us here involves the efficiency and effectiveness of various languages as AI programming environments.

Let's take a quick look at a few popular languages in this context.

### BASIC

Generally, BASIC doesn't deal with lists and has no equivalent data structure. It is also unstructured in most of its implementations, which creates problems in program design, whether we are discussing AI or other types of programs. Microsoft BASIC 2.00 for the Mac overcomes some of these problems but has no list-handling capability.

Still, just to prove that the decision is arbitrary, a whole book on the subject of expert systems has been produced using BASIC as the language for implementing the systems (see Appendix C for details on the book by Chris Naylor).

**Pascal**
Pascal is increasingly the language of choice on microcomputers. It is a substantial improvement over BASIC for AI work. Pascal is structured, and it has good string-handling capabilities; but it has no lists, is not capable of self-modification, and is not really extensible.

**Logo**
Logo is an offshoot of AI research. Dr. Seymour Papert developed the language at MIT's AI laboratory in the late 1970s after seeing how children could learn to think abstractly in a Logo programming environment. The language has powerful list-handling capability, good string manipulation, and is highly extensible. However, it cannot be modified by a program while the program is running.

**Prolog**
Since we discuss this language extensively in Chapters 7, 8, and 12, we will say here only that Prolog is a nonprocedural language and thus lends itself to a different kind of analysis as an AI language than the others we are considering here. Further, it is just becoming available on microcomputers.

**LISP**
LISP is one of the oldest programming languages, having been introduced in 1958. As an AI language, it has all of the desirable features. In addition, and not to be discounted, it has a rich history and a huge collection of programs already written in it, which serve as a key research base for future AI work.

The language has also been implemented in numerous versions on microcomputers and two are now available for the Macintosh®.

**Making the Choice**
When selecting an AI language on your micro ask yourself about your intended AI application:

1. Is a great deal of the knowledge or data base with which the program is expected to work made up of strings and relationships?
2. Are complex interrelationships among pieces of knowledge involved in problem-solving?
3. Does the application call for the ability to create a new "vocabulary" for the language?
4. How important is speed of programming?

In the first three cases, the more affirmative answers you give, the more likely you are to want to try LISP or Logo. The more negative answers, the more likely it is that a more traditional, conventional language will suit your needs.

## Summary and Looking Ahead

In this chapter you have a brief background against which to understand the debates in the AI community regarding selection of an appropriate language for

AI development work. We have discussed some of the considerations that go into the selection of a language for a specific AI application.

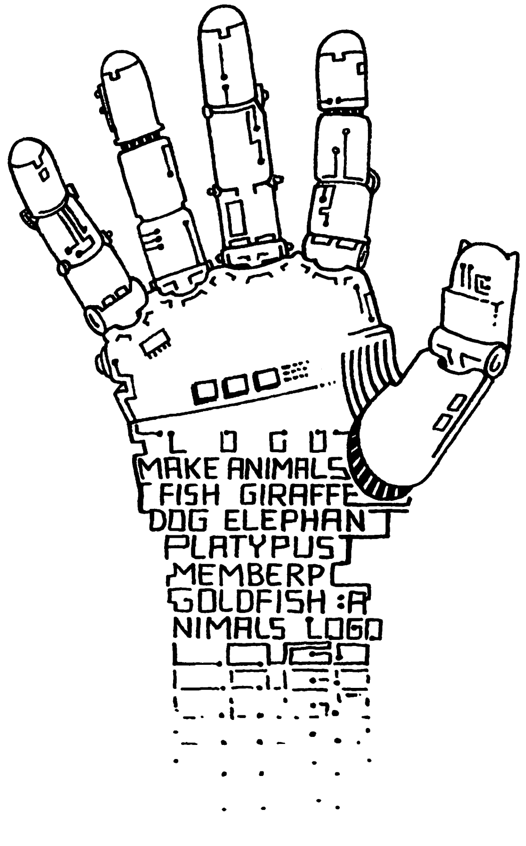
In looking at the desired features of an AI language and then holding each of several popular micro-based languages up to that standard, we have been able to assess the strengths and weaknesses of each language.

In Chapter 10 we begin our look at the operations and functions of specific languages.

CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTE

10

# A Quick Logo Refresher



- How a Logo Program Is Constructed
- Macintosh® QuickDraw Graphic Commands in Logo
- Key Logo Commands and Their Use
- Property Lists: Key Concept in AI Uses of Logo

This chapter constitutes a brief refresher course in the basic concepts, syntax, and commands of the Logo programming language. For the most part, it is a generic review of the language although it incorporates some commands unique to ExperLogo®. These unique commands are noted as they are reviewed.

After reading this chapter, if you wish to learn more about Logo, select one of the Logo programming introductions listed in Appendix C.

## Logo's Image

Logo has been both popular and maligned ever since its introduction in the 1960s. It has been popular, particularly with educators, because it is easy to learn and use and because it is an extensible language. This latter quality results from Logo's ability to use procedures inside other procedures, leading to Logo's apparent ability to "learn" new words and commands from the user. This easily understood way of programming has led people to conclude that Logo is an excellent language to use for exploring the world of computers.

At the same time, other people view Logo as a "toy" language that is quickly outgrown. This perception has been aided by the companies who published Logo language systems with very limited vocabularies, but occupying large amounts of the relatively small memory spaces available on the popular microcomputers of the late 1970s and early 1980s.

Most Logo implementations, for example, did not include any way of working with information other than with Logo procedures on disk files. So, writing real-world applications in Logo was impossible. Similarly, Logo was an *interpreted* language. To a great extent it still is. This makes most Logo programs run slowly.

But Logo enthusiasts always realized that Logo is a far more powerful programming tool than popular perception held. Logo is an incarnation of the powerful AI programming language known as LISP. Logo uses many of the same concepts of processing lists—a key concept in AI programming—as LISP does. Some Logo implementations are, in fact, written in LISP—notably the ExperLogo® chosen for this book. ExperLogo® is really a subset of ExperLisp with a special processor that interprets Logo commands and carries them out. In its Experlogo implementation, Logo is as fast and as powerful as such "serious" languages as C and Pascal.

## Anatomy of a Logo Program

A Logo program consists of one or more procedures. Each procedure carries out a step in the process the program is designed to handle. Each procedure has approximately the same *structure* as the other procedures.

### The TO Requirement

A Logo procedure must begin with the reserved word TO, which signals Logo that what follows is a definition of a procedure to be remembered and carried out as one sequence of steps. This differentiates the *definition* of a procedure from the actual *execution* of a procedure. Thus, we would type as the first line of a procedure definition to draw a box:

```
TO BOX
```

To ask the computer to execute that procedure we would type:

```
BOX
```

**Passing Variables**
Some Logo procedures require information be passed to them from outside their boundaries. Such a procedure includes the names of the variables to be used as part of the TO procedure definition line. For example, if our BOX procedure was to be told how big the box is to be, the first line of the procedure might be:

```
TO BOX :SIZE
```

Notice that variable information needed by the program is preceded by a colon; this signals Logo that the name which follows is the name of a variable, not another procedure about which we expect Logo to "know."

There is no theoretical limit to the number of variables that may be used by a Logo procedure.

Using variables tends to make Logo procedures generalized and thus potentially usable in similar programs that may require the same procedure with slightly different parameters.

**Using Variables in a Logo Procedure**
We often use variables inside Logo procedures. In fact, if we use the method of passing variables to a program, we will always use the variable information somewhere inside the procedure. Otherwise, why send the information along?

With variable usage inside its boundaries, our BOX procedure might look something like this:

```
TO BOX :SIZE
  REPEAT 4 [FD :SIZE RT 90]
END
```

Notice that the variable name is preceded by a colon; this is standard Logo syntax and common to all versions or dialects. Variable names are preceded by colons when their *values* should be used. When *assigning* values to variables or changing their contents we do *not* precede the variables' names with a colon.

***Using MAKE to Assign Variable Values*** One of the most common built-in Logo functions is MAKE. It is used to assign a value to a variable name. MAKE is followed first by the name of the variable to be created or changed and then by the value to be assigned to that variable. The following examples are all valid uses of the MAKE command:

```
MAKE VAR1 13
MAKE VAR2 '13 ←VAR2 becomes the letters "1" and "3"
MAKE STRING1 'THIS IS A STRING, PEOPLE
MAKE LIST1 [THIS IS A LIST]←more about lists later
```

A string value is assigned to a variable by preceding the value with a single quotation mark, as we did in assigning VAR2 and STRING1 their values.

Notice that, as we have previously indicated, the variable name is not preceded by a colon in these examples because we are using the variables as variable names, not using their values. Once we have defined VAR1 to be the number 13, though, we could add 5 to its value by coding:

```
MAKE VAR1 :VAR1 + 5
```

Now, in the second use of the variable's name, we intend to refer to its contents—the number 13—rather than its identifying name, so we precede it with a colon.

**Calling Other Procedures**

As we saw earlier, one feature that makes Logo so powerful is the ability to call procedures from inside other procedures. There are dozens of instances of this common Logo technique in the programs in this book.

If we tell Logo to do something it doesn't have a procedure defined for, an error message will appear saying something like:

```
I don't know how to BRXFMJX
```

The message may vary from one Logo to another, but the idea is the same; we must *tell* Logo how to carry out a procedure before we try to use it.

A procedure can use any other procedure defined in the program exactly as if the created procedure were in Logo's large built-in vocabulary. (We'll review some of the terms in Logo's vocabulary later in this chapter.) As we define procedures, our newly defined "words" are added to Logo's vocabulary. To observe this, we can use our own procedure to define a function that Logo knows by some other name. Let's take the Logo command BUTFIRST, for an example. (This command returns everything but the first element of whatever list it is told to operate on.)

Let's say we like the word TAIL instead of the longer BUTFIRST (deciding that the FIRST is the head of whatever we are working with and BUTFIRST the tail). We can define a new word TAIL by writing this procedure:

```
TO TAIL :OBJECT
  BUTFIRST :OBJECT
END
```

Now we can use TAIL in other procedures exactly as if we had used the built-in BUTFIRST. Thus, we could program a procedure that printed each letter of a word or list on a separate line:

```
TO PRINT_FIRST_LETTER :OBJECT
  PRINT FIRST :OBJECT
  MAKE OBJECT TAIL :OBJECT
  IF EMPTYP :OBJECT [STOP]
  PRINT_FIRST_LETTER :OBJECT
END
```

It isn't important that you understand what this procedure is doing. What is important is that you note that it calls our recently defined TAIL procedure which substitutes for BUTFIRST. Logo programs are almost always constructed of procedures calling procedures that call other procedures. Ultimately, everything is defined in terms of the Logo's built-in vocabulary.

**The END Requirement**
Every Logo procedure *must* end with the word END. Some programming languages do not require the explicit termination of a procedure, but Logo does.

## The Language of Logo

The rest of this chapter briefly discusses a number of often-used Logo commands and functions.

## Graphics Commands

Logo is a very graphic language. The use of the turtle as a mythical drawing character in early implementations of the language brought the term "Turtle Graphics" into the English language. The first Logo that used a turtle was Microsoft's LCSI Logo® for the Macintosh®. Turtle Graphics also led to the creation of organizations called "Friends of the Turtle" and "Turtle Trackers." ExperLogo® has departed from this tradition, choosing "Bunny Graphics" as the name of its three-dimensional and spherical drawing animals—because, of course, the language is so much faster.

This section presents the most often used Logo graphics commands. (We have avoided ExperLogo's® 3-D and spherical drawing commands, which are not used in this book and are not part of standard Logo.)

**CLEARing Space to Draw**
Drawing in any Mac version of Logo takes place in a Graphics Window. It's a good practice to make sure that the contents of the window are erased before creating new art in it. The usual Logo command to clear the Graphics Window is CLEARSCREEN, usually abbreviated CS. Some implementations of Logo on the Macintosh® use CG to clear the Graphics Window and CT to clear the Text Window.

**HOME**
Whether you're drawing with a turtle or a bunny (or any other member of the menagerie, for that matter!), you'll frequently use a universal Logo command called HOME. HOME returns the turtle or bunny to its original position, "pointing" toward the top of the screen. The original position on the Mac is the center of the Graphics Window.

HOME does not usually clear the screen and will work whether or not the turtle or bunny is visible (more on this as we discuss other commands).

**SHOWTURTLE and HIDETURTLE**
The commands SHOWTURTLE and HIDETURTLE—and their ExperLogo® equivalents, SHOWBUNNY and HIDEBUNNY—determine whether or not the shape used to draw in the Graphics Window is visible. These commands are abbreviated ST, HT, SB, and HB.

Whether or not the turtle or bunny is visible, any drawing it is directed to do is displayed. In most Logos, including ExperLogo®, the drawing "animal" is really a triangle. We'll stop the confusion of mixing turtles and bunnies and use the more conventional turtle from now on.

**Direction Commands**

When we draw using Logo commands, we maneuver the turtle and its accompanying pen inside (or outside) the Graphics Window.

We can instruct the turtle to move forward or backward, turn left or turn right. We must tell the turtle how far to move or turn.

Any movement made inside the Graphics Window with the pen down (see next section) *results in a line being drawn.* Moving the turtle forward requires the FORWARD (FD) command. To cause the turtle to move forward 50 steps from its current position, we write:

```
FD 50
```

Backward movement requires the BACK (BK) command. Like the FD command, it takes one command to tell the turtle how far to move backward.

We turn the turtle to the left by using the LEFT (LT) command, accompanied by the number of *degrees* we wish to turn. This is an *absolute* amount to turn, not a movement relative to a fixed point. Thus, if the turtle is already pointing at an angle, and we use an LT command to move it to the left 30 degrees, it will move from where it is 30 degrees farther to the left. If it started out pointing straight up (i.e., at 0 degrees), this results in the turtle pointing at a 30-degree angle. But if, starting at 0 degrees, we execute two consecutive LT commands:

```
LT 30
LT 45
```

the turtle now points 75 degrees left of the straight-up position. We emphasize this because it is a frequent point of misunderstanding for inexperienced Logo programmers.

As you might expect, RIGHT (RT) works the same way. The back-to-back commands:

```
LT 90
RT 90
```

cancel each other out. They serve no purpose unless the turtle is visible and you just want it to wiggle.

Figure 10-1 shows the results in a Graphics Window of executing in ExperLogo® the following command sequence:

```
CS HOME FD 50 RT 120 FD 50 RT 120 FD 50 LT 60 FD 60 LT 30 BK 90
```

You can tell where things started because the first drawing command is executed after a HOME instruction. The initial line, then, is the one drawn straight toward the top of the screen.

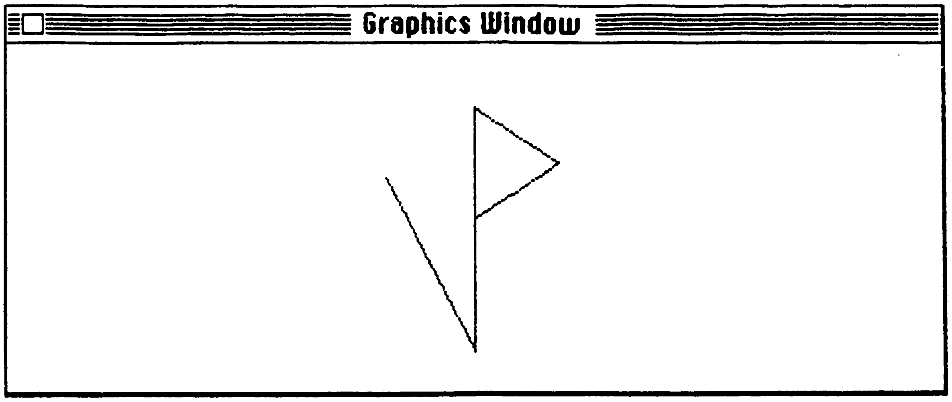


**Figure 10-1. Result of Logo drawing with turtle hidden**

If you want to shift the position of the turtle to a specific angle, regardless of where it is when things start, use the SETHEADING command. For example, if you want the turtle moving horizontally left, write:

```
SETHEADING 90
```

**PEN Controls**

The turtle doesn't actually do the drawing in the tiny world of Logo. It carries a "pen" which does the actual drawing. The pen is either "up" or "down" (not drawing or drawing when the turtle moves). To move the turtle without drawing anything, use the PENUP (PU) command. Use a PENDOWN (PD) command in order to draw after a PU command has been carried out. In addition, Logo on the Mac permits the pen to do two other things: erase and reverse.

Using a PENERASE (PE) command makes the pen draw in the same color as the background. That's usually white, but can be changed by the program as we'll see in the next section. With the pen in the erase mode, any lines previously drawn that you retrace are erased.

PENREVERSE (PX) is a close cousin of the PE command. It, too, erases any lines it passes over. But it also draws lines where there were none before. So, in other words, PX reverses the situation, drawing lines where none were and erasing lines where they had been drawn. Figure 10-2 demonstrates the difference.

Figure 10-2b results from using a PE command and drawing across a portion of the figure shown in Figure 10-2a. Figure 10-2c results from using the PX command on the same path.

The pattern and size of the pen can also be set by Logo commands on the Mac; check the documentation for your version of Logo for specific instructions.

**BACKGROUND Control**

ExperLogo® permits control of the background pattern of the Graphics Window with the SETBACKGROUND (SETBG) command. Not all Logos for the Mac allow this. The SETBG command in ExperLogo® takes as an argument the name of a pattern, chosen from Figure 10-3.

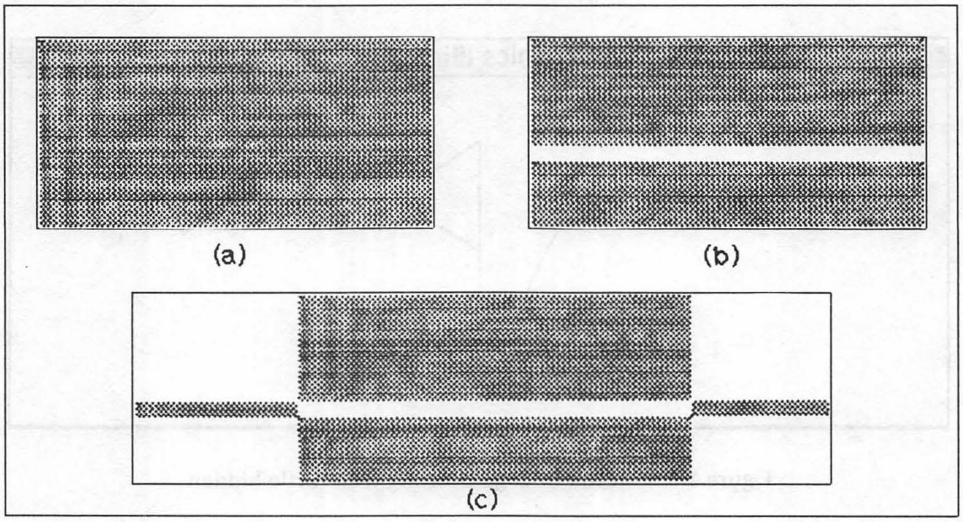


**Figure 10-2. PE compared to PX**

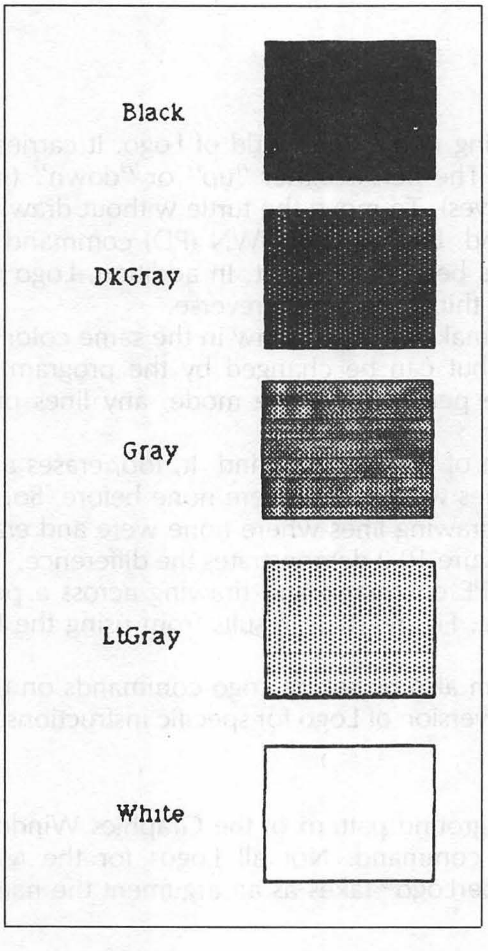


**Figure 10-3. Background patterns**

## Macintosh® QuickDraw Graphics Commands

One feature that makes the Mac such a powerful machine is a set of built-in routines known collectively as the QuickDraw Manager. These routines are familiar, at least in terms of their function, to anyone who has ever used MacPaint.

**In General: Accessing QuickDraw from Logo**
All implementations of Logo on the Mac permit you to use the Mac's QuickDraw routines, but the method of using the routines varies by dialect. Consult your language manual to see how your version of Logo handles QuickDraw graphics. ExperLogo® handles the QuickDraw routines just as they were designed by Apple Computer. Microsoft Logo®, on the other hand, permits access to all of the routines, but in a different syntactical way.

**QuickDraw Shapes and What to Do With Them**
Four shapes are available in the QuickDraw routines: arcs, ovals, rectangles, and round-cornered rectangles. Each of these can be drawn (usually called "framed"), filled in with a pattern, erased, or reversed. The two methods for filling in the shapes are filling and painting. Filling requires you to tell the computer which pattern to use; painting fills in with the current pen pattern.

The commands in ExperLogo® to carry out these QuickDraw routines combine the action (FILL, PAINT, FRAME, ERASE, INVERT) with the shape name (RECT, ROUNDRECT, OVAL, ARC). The command FRAMEROUNDRECT, along with the proper arguments, draws a round-cornered rectangle of a given size at a given point in the Graphics Window.

**Cursor Control**
The cursor (sometimes referred to in Mac books and manuals as the "pointer") displays the current position of the mouse. ExperLogo® includes commands to cause the cursor to be displayed, hidden, or temporarily hidden. SHOWCURSOR displays the cursor so you can see mouse movements. HIDECURSOR causes the cursor to be invisible, though mouse movements are still tracked in the system. OBSCURECURSOR hides the cursor until the mouse is moved. Many Mac programs use this ability to obscure the cursor. For example, most text editors available on the Mac hide the cursor while you're typing so the mouse's cursor doesn't get confused with the text pointer.

**Other QuickDraw Commands**
The size, font, and style of text to be drawn in the Graphics Window can also be controlled by Logo commands. In addition, versions of Logo for the Mac will also permit you to manipulate complex graphic objects such as pictures, regions, and polygons. (A discussion of these topics is beyond the scope of this chapter. See the reference manual that came with your Logo for more information.)

## Input/Output Commands

Like programs written in other languages, Logo programs are nearly useless unless we can put information into them and get results out. The commands to handle displaying information and getting data from the keyboard are relatively standard from one Logo dialect to another. Unfortunately, disk file input and output vary a great deal.

**PRINT and Its Relatives, PRINC and TYPE**

To display information in the Text (or Listener) Window of the Mac, use the PRINT command. As a rule, anything that follows a PRINT command will be displayed in the active Text Window. To print the sentence, "This will display in the Text Window," we would code the following line:

```
PRINT [This will display in the Text Window]
```

In ExperLogo®, the use of guillemets marks strings to be displayed. So, in ExperLogo®, our command could also be written:

```
PRINT <<This will display in the Text Window>>
```

The guillemets are produced by using the [OPTION] and back-slash key together for the opening pair and the [SHIFT][OPTION] and backslash key for the closing pair.

There are occasions when we wish to print something without having the program insert a [RETURN] and line feed at the end of the statement being printed. This is particularly true when we write programs that include prompts for input from the user. Programming the line

```
PRINT [Enter a number between 1 and 100:]
```

will result in the user's response being entered on the line below the request. But, using the PRINC command in ExperLogo® suppresses the carriage return and line feed so that writing this line:

```
PRINC [Enter a number between 1 and 100:]
```

results in the user's answer being placed on the same line as the prompt. In Microsoft's Logo and most other dialects of the language, the TYPE command has the same effect.

**READing Data**

To obtain information from outside the program for use in the program, we can use either the READLIST or the READCHAR command in Logo. There are two difference between the commands. First, they differ in terms of how the user indicates that the requested information has been entered. Second, they differ as to the data type of the information entered.

READLIST accepts all of the user's input to the first [RETURN] and stores the result as a list. READCHAR looks only for the first character entered by the user, places that key in a character variable and continues processing without waiting for the [RETURN].

Neither command will *store* anything unless it is explicitly told to do so with a MAKE command or a similar assignment command.

If we code this line:

```
MAKE VALUE1 READLIST
```

at one place in a program and this line:

```
MAKE VALUE2 READCHAR
```

and in response to both input requests we type the words "THIS IS MY ANSWER," the result is that VALUE1 will contain the list:

```
[THIS IS MY ANSWER]
```

while the variable VALUE2 will contain the single letter "T."

### File Usage in ExperLogo®

One thing about language design that plagues programmers is the inconsistency in how file input/output operations are handled on the same machine, even by different dialects or implementations of the same language. The user manual *ExperLogo® for the Macintosh®* from ExperTelligence contains a brief tutorial on disk file I/O. Some programs, including "Poetry Maker," in this book use file I/O relatively extensively.

A detailed discussion of file I/O in Logo is beyond the scope of this book. Refer to your user manual for information on file handling specific to your software.

## Conditionals and Program Flow Control

Generally, the flow of a Logo program follows a relatively straight line. First an instruction is executed, then the next one in the procedure is executed, and so on, until a new procedure is called or the program ends. Sometimes we want to alter that straight-line flow. Basically, there are three ways we can do this: conditional processing using IF testing, loops using REPEAT constructions, and unconditional branching using GO and LABEL commands in conjunction with one another.

### IF Testing and Conditional Processing

When a program needs to take one course of action in one set of circumstances but a different course of action in another set of conditions, an IF construction will obtain the desired result.

In ExperLogo®, as in most dialects of the language, the IF statement is followed by a *list* of instructions to be carried out if the statement is true and, optionally, a second list of instructions to be carried out if the statement is false. If the optional list is not present, and the IF statement turns out false, the program skips the first statement list and continues processing normally.

Two simple examples will help to clarify this usage of the IF construct.

***IF With One Statement List*** The first, simpler, example involves just one statement list to be executed when the IF condition is true:

```
TO SHOW__IF
  PRINC [Give me a number between 1 and 10, please:]
  MAKE VALUE READLIST
  IF (OR :VALUE <1 :VALUE >10) [PRINT [SORRY, TRY AGAIN] SHOW__IF]
  NEXT__PROCEDURE
END
```

The IF statement uses the OR logical function (discussed later) to determine if an acceptable value has been entered by the user in response to the instruction. If it has been, the procedure NEXT__PROCEDURE is executed. Otherwise, if the answer is smaller than 1 or greater than 10, the message "SORRY, TRY AGAIN" is displayed and the procedure is run again.

***IF With Two Statement Lists*** Sometimes, the user will be given two clear alternative responses. For example, if we are writing an educational program that requires a correct answer for a question, we want different types of feedback to the user. The program needs to react to right and wrong answers. We might use a procedure like this to accomplish this task:

```
TO ASK__IT
  PRINT [WHO WROTE A MIDSUMMER NIGHT'S DREAM?]
  MAKE ANSWER READLIST
  IF EQUALP ANSWER [WILLIAM SHAKESPEARE]
    [PRINT [YOU ARE ABSOLUTELY CORRECT!]]
    [PRINT [SORRY, TRY AGAIN.] ASK__IT]
END
```

When users provide the right answer, the program indicates that they are right and the procedure ends. Otherwise, a message informs them that they are wrong and the procedure runs again. Notice the important consideration here: both courses of action are provided as *lists* of instructions, even when only a single instruction is to be carried out.

**Testing Alternatives**

The key part of the IF construction is the portion that evaluates a condition to determine which course of action is to be taken. This almost always involves comparing the values of two different items against one another. We can test them to determine if they are identical, or if one is larger than the other or if they are not equal to one another. Figure 10-4 provides a list of the most common Logo tests for conditional processing.

Sometimes we want to combine two tests, as we did in our sample SHOW__IF procedure. In such a case, we use any combination of three *logical operators*: AND, OR, and NOT. These Logo operators are almost self-explanatory, but a brief example or two will help to fix them in your mind.

If we use the AND logical connective, then both tests (or all of them if there are more than two) must be true for the IF statement to be found to be true. For example, when this line executes:

```
IF AND :VALUE>1 :VALUE<10 [PRINT [FINE.
PROCEED] NEXT__PROCEDURE]
```

| *Symbol* | *Meaning* |
|---|---|
| = | Exactly equal to |
| < | Less than |
| > | Greater than |
| <> ≠ | Unequal |
| ≤ <= | Less than or equal to |
| ≥ >= | Greater than or equal to |

**Figure 10-4. Conditional tests**

the message "FINE. PROCEED" will print and the procedure NEXT_PROCEDURE will be carried out, if and only if *both* conditions are true—that is, if the value of the variable called VALUE is *both* greater than 1 *and* less than 10.

On the other hand, when this line of Logo code is encountered:

```
IF OR :VALUE>1 :VALUE< -10 [PRINT [FINE.
PROCEED] NEXT_PROCEDURE]
```

and the variable called VALUE is *either* greater than 1 *or* smaller than −10, processing will continue with the procedure called NEXT_PROCEDURE after the message is printed. (Note that we changed the value 10 from our AND example to −10 here. To choose between values of greater than 1 or less than 10 would exclude no numbers whatsoever, since all known numbers are *either* greater than 1 *or* less than 10!)

**Looping With REPEAT**

At times we want to repeat a given instruction or set of instructions a certain number of times. In those circumstances, Logo uses the REPEAT function as shown in this example:

```
REPEAT 4 [FD 50 LT 90]
```

This is the classical one-step Logo procedure to draw a square 50 pixels on a side. We can use a variable for the count, too, as in:

```
REPEAT :COUNT [PRINT [HELLO THERE, YOU BRIGHT OPERATOR]]
```

(It's probably not a good idea to set the value of COUNT too high; someone might come into the room while we are shamelessly letting the Mac compliment us.)

**GO and LABEL**

There are those who argue that using GO and LABEL constructions defeats one of the main attractions of Logo—its clean, structural organization. However, there are programming problems that seem to have no clean solutions. Generally, we only *think* that is the case; we can almost always solve the problem in the far more Logolike manner of writing a new procedure and calling it.

The two commands work together. A LABEL command defines a point to which the procedure can later be told to GO. The following example shows how they work together.

```
TO COUNTDOWN :NUM
  LABEL COUNTER
  IF :NUM > 0
  [PRINT :NUM MAKE NUM :NUM  - 1 GO COUNTER]END
```

The line that sets up the label called COUNTER gives the instruction list following the IF construct a place to go in the program when it has finished printing out the number it has and decreasing it by one.

We have avoided the use of GO and LABEL commands in our programs whenever possible. We suggest you do the same in your Logo programming.

## Math Functions

Routine math operations like addition, subtraction, multiplication, and division can be performed in ExperLogo® in one of two ways. The first, more familiar way, is called "infix" notation; the second is called "prefix" notation.

### Infix Forms of Math

In an "infix" form of mathematics, the symbol to carry out an operation is placed between the numbers or variables that it applies too. Thus we would write:

```
3 + 4
```

to add the numbers 3 and 4. Similarly,

```
PRINT [3 * 4]
```

prints the answer found by multiplying 3 by 4. This is how most math on the Mac will be performed in Logo. Division uses the slash (/) and subtraction the minus sign (−) Exponentiation—raising a number to a power—uses the up-pointing caret (^) made by using a [SHIFT] − 6 combination.

### Prefix Forms of Math

There are times when we wish to use a different approach to math, putting the function to be performed first. (Never mind why we would wish to do so; suffice it to say that there *are* times when this approach is more efficient than the infix method.) In such situations, Logo provides us with prefix forms. These forms are SUM, PRODUCT, QUOTIENT, DIFFERENCE, and REMAINDER.

We can add 3 and 4 by coding:

```
SUM 3 4
```

in exactly the same way we coded "3 + 4" earlier. The result will be the same. Similarly, PRODUCT handles multiplication, QUOTIENT division, and DIFFERENCE subtraction. The REMAINDER function looks at a division problem and provides the remainder rather than the answer (quotient). Thus, if you write a line of Logo code like this:

```
PRINT QUOTIENT 15 4
PRINT REMAINDER 15 4
```

the computer's answer will be 3 and 3 because 15 divided by 4 is 3 with a remainder of 3.

**RANDOMizing Activities**
The RANDOM function is one other math function which finds frequent use in some kinds of Logo programs. This function produces a random number between 0 and any number it is given, minus 1. Thus, writing this line of Logo code:

```
MAKE RESULT RANDOM 10
```

will produce a random number between 0 and 9.

**Other Math Commands and Functions**
Several other useful math operations can be performed with Logo commands on the Mac and most other machines. These include square root extraction, trigonometric functions, absolute value calculation, and truncation or rounding.

***Square Roots*** A square root can be extracted in Logo by using the SQRT built-in command. To find the square root of 37, for example, code:

```
MAKE ANSWER SQRT 37
```

The answer is 6.08276.

***Trigonometric Functions*** All basic trigonometric functions—SIN, COS, TAN, and ARCTAN—are available in Logo using those names as the commands. In all cases, the angle must be furnished in radians. Using these functions, any needed trigonometric function can be derived.

***ABSolutely!*** There are times when we don't care about the sign of a number, only its value. That is referred to as the number's *absolute value* and it is found in Logo by the ABS function. The following two lines produce the same result when the value of the variable NUM1 is 45 and the value of the variable NUM2 is $-45$.

```
PRINT ABS :NUM1
PRINT ABS :NUM2
```

In both cases, the program will print 45. This function finds its most frequent use when we want to know the distance between two points or the difference between two values, not caring whether the result of the subtraction is positive or negative.

***TRUNCation and ROUNDing*** There may be times when a math problem's answer given to us by Logo is too large to suit our purposes. For example, if we want to know how much it will cost us to buy 13 lemons at $1 per dozen, the answer $1.0833333. . . isn't going to be particularly helpful. We can figure out the answer, of course, but it's messy and makes the report we print look bad.

In such situations, we can either truncate the answer or round it off. When we truncate or round off in Logo, we eliminate the fractional decimal portion. Thus without some intermediate step, truncating or rounding the value $1.08939 would lead to the same answer: $1. To get around this problem when we really want to truncate or round to the hundredths place, we multiply the value by 100, then truncate or round it, then divide it by 100 again. We'll avoid that nicety in the next few paragraphs, but keep it in mind when you write programs using these functions.

Truncating the value $1.08939 leads to the answer $1.08. Rounding it off leads to the answer $1.09. The Logo functions are TRUNC and ROUND:

```
PRINT TRUNC :VALUE1
PRINT ROUND :VALUE1
```

Most of the time, accuracy is important, so we round values.

## List-Processing Commands

One of the features that makes Logo particularly well suited to programming AI applications is its powerful list-handling ability. Treating many kinds of information as lists of data to be manipulated not only makes intuitive sense—i.e., it matches the way we do things in real life—but it enables us to do very flexible things with the contents of those lists.

Logo offers us a strong set of commands to extract parts of lists (and, optionally, redefining the list itself in the process), adding to or combining lists, obtaining information about lists, and modifying the contents of lists.

### Extracting Parts of Lists

We can extract from a list its "head," (first element), its "tail," (everything but the first element), its last element, or all but the last element. We can also extract any given item from the list.

The Logo commands that provide us with this capability are FIRST, BUTFIRST, LAST, BUTLAST, and ITEM. The first four operate like one another and will be clarified by this example.

```
[THIS LIST [CONTAINS LISTS [WITHIN LISTS]] BUT IS STILL A [LIST]]
```

Let's call this LIST1. Note that the list is composed of eight elements, as shown in Figure 10-5.

The following shows the results of the various extractions. Note particularly how combinations of list extractions can be used effectively to narrow our extraction to exactly what is desired.

$$\underbrace{\text{[THIS}}_{①}\ \underbrace{\text{LIST}}_{②}\ \underbrace{\text{[CONTAINS LISTS [WITHIN LISTS]}}_{③}\ \underbrace{\text{BUT}}_{④}\ \underbrace{\text{IS}}_{⑤}\ \underbrace{\text{STILL}}_{⑥}\ \underbrace{\text{A}}_{⑦}\ \underbrace{\text{[LIST]]}}_{⑧}$$

**Figure 10-5. The eight elements of LIST1**

```
PRINT FIRST :LIST1
  THIS ← all underlined material is computer's response
PRINT LAST :LIST1
  [LIST]
PRINT BUTFIRST :LIST1
  [LIST [CONTAINS LISTS [WITHIN LISTS]] BUT IS STILL A [LIST]]
PRINT BUTLAST :LIST1
  [THIS LIST [CONTAINS LISTS [WITHIN LISTS]] BUT IS STILL A]
PRINT FIRST BUTFIRST :LIST1
  LIST
PRINT FIRST BUTFIRST BUTFIRST :LIST1
  [CONTAINS LISTS [WITHIN LISTS]]
```

Using the ITEM command, we can select any given item in the list if we know its relative position. Staying with LIST1 as our example, notice the effects of the ITEM command in these operations:

```
PRINT ITEM 4 :LIST1
  BUT
PRINT ITEM 9 :LIST1
  nil ← LIST1 has only eight items; Item nine is empty
PRINT ITEM 2 BUTFIRST BUTFIRST :LIST1
  BUT
```

ExperLogo® contains one other unique command which we have had occasion to use in the programs in this book. The ELEMS command is a variation on ITEM; it permits us to extract a number of adjoining items in one step. ELEMS takes three arguments: the first tells it which item or element to start extracting from; the second tells it how many items or elements to extract; and the third tells it the name of the list on which to operate. If we wanted to take the second and third elements of the list we have called LIST1 in this example, we would write:

```
PRINT ELEMS 2 2 :LIST1
```

(The computer's response, incidentally, would be to tell us that the second and third items of the list are, respectively, LIST and [CONTAINS LISTS [WITHIN LISTS]].)

### Adding to and Combining Lists

There are three Logo commands for adding information to a list or combining two or more lists: SENTENCE, FPUT, and LPUT. SENTENCE, abbreviated SE, takes an arbitrary number of items or lists and converts them into a single list. If two objects are used, it is written:

```
MAKE NEWLIST SENTENCE :LIST1 :LIST 2
```

If more than two lists are involved in the creation of the SENTENCE, we can use parentheses:

```
MAKE NEWLIST (SENTENCE :LIST1 :LIST2 :LIST99)
```

FPUT takes two arguments from any kind of Logo object: lists, words, or strings. It creates a new object—note that it is not necessarily a list, depending on the objects provided as arguments—with the first object in front of the second. LPUT operates similarly, except that it puts the first object *after* the second object.

If we start with two lists:

```
MAKE LIST1 [THE]
MAKE LIST2 [END]
```

then the following results will be obtained using these three commands:

```
PRINT SE :LIST1 :LIST2
  [THE END]
PRINT FPUT :LIST1 :LIST2
  [END THE]
```

**Obtaining Information About Lists**

We can find out how many elements are in a list by using the COUNT command. (Note that an *element* in this sense can also be a list within a list, so counts don't always return the results we might expect to get by counting each "word" in a list!). Here's an example:

```
PRINT COUNT [THIS LIST [CONTAINS SEVERAL] ITEMS]
  3
```

The list [CONTAINS SEVERAL] is counted as one item in the larger list. So even though it *looks* like there are five items in the list, Logo only sees four.

We can also determine if an item *is* a list using the LISTP test. Related to this are the STRINGP and WORDP tests which determine whether an object is a string or a word. Look at this example:

```
MAKE OBJECT1 [APPLE ORAÑGE TOMATO]
MAKE OBJECT2 'ALASKA ←string preceded by single quotation
MAKE OBJECT3 <<Stringing me along, eh?>> ← string in guillemets
PRINT LISTP :OBJECT1
  T
PRINT LISTP :OBJECT2
  nil
PRINT :STRINGP :OBJECT1
```

```
  nil
PRINT :WORDP :OBJECT2
  T
```

When we manipulate lists in Logo, we often reduce their size by removing elements after we have performed operations on them. But, if we try to perform operations on nonexistent items, we will encounter serious problems. So Logo provides us with a way to find out if a list is empty. It is called, logically enough, EMPTYP, as shown in this example:

```
IF EMPTYP :LIST1 [STOP]
```

One final but very important thing we can find out about a list is whether a given object can be found in its contents. We use the MEMBERP command for this purpose. An example will clarify how MEMBERP works.

```
MAKE OBJECT1 'POUND
MAKE POETS [ELIOT SHAKESPEARE POUND FROST DING-PAO]
MEMBERP 'POUND :POETS
  [POUND FROST DING-PAO]
```

Our MEMBERP function finds that POUND is indeed a member of the POETS list and returns the list from the point where it locates the requested object to the end of the list. This can be quite a useful technique when searching through lists. Because the IF function views any result other than "nil" as meaning "true," we can write statements such as:

```
IF MEMBERP 'POUND :POETS [PRINT [HE CERTAINLY IS!] STOP]
```

**Modifying the Contents of Lists**

ExperLogo® is particularly rich in list-processing commands that enable us to make selective and significant changes in lists. These are peculiar to ExperLogo®.

The SUBST command (for "substitute") is one of the most powerful of these commands. It replaces all occurrences of a given element in a list with a new element. (In fact, SUBST works on words, arrays, and strings as well as lists, but we will confine our discussion here to lists.) For example:

```
MAKE LIST1 [BIG IDEAS MAKE BIG PROBLEMS FOR BIG COMPANIES]
SUBST 'BIG 'LITTLE :LIST1
  [LITTLE IDEAS MAKE LITTLE PROBLEMS FOR LITTLE COMPANIES]
```

ExperLogo's® DELETE and REMOVE functions can be used to modify lists by removing elements from them. DELETE will erase all occurrences of an element in a list (or other object) except for the first position. REMOVE will erase all occurrences of an element anywhere *at the top level* of a list. REMOVE will not erase an occurrence of an element inside a list within a list. Let's look at these examples for clarification:

```
MAKE TESTLIST [BOOKS [MANY BOOKS] ARE GOOD BOOKS ARE GOOD] DELETE
'BOOKS :TESTLIST
```

```
  [BOOKS [MANY] ARE GOOD ARE GOOD] ←left first "books"
REMOVE 'BOOKS :TESTLIST
  [[MANY BOOKS] ARE GOOD ARE GOOD] ←left second "books"
```

Two other commands warrant mentions. RPLACA and RPLACD replace, respectively, the FIRST and BUTFIRST elements of a list with the new information provided in the command. Again, a brief example will help clarify.

```
MAKE LIST1 [L O G O B O O K]
RPLACA :LIST1 [A N Y Z O]
  [A N Y Z O O G O B O O K]
RPLACD :LIST1 [A N Y Z O]
  [L A N Y Z O]
```

## Arrays

Arrays are a special type of variable that is quite popular in many programming languages, particularly those that deal primarily with numeric data. (The Micro Blocks World program in Chapter 5 makes extensive use of arrays.) Most implementations of Logo do not include arrays. Strangely, both ExperLogo® and Microsoft Logo® for the Mac *do* permit the creation and manipulation of arrays, although Microsoft Logo® limits arrays to one dimension, making them more like lists of predetermined size and therefore less useful than those in ExperLogo®. We will discuss two-dimensional arrays here; one-dimensional arrays are similar and easier to learn.

Essentially, an array is a table that has rows and columns of information. Each row and column in the array is numbered, starting with zero for the upper row and left-most column. The *index position* is given in brackets after the name of the array. Data is placed into or extracted from an array by providing the index position of the array to be affected. For example, to put the number 43 in the upper left corner of an array previously defined as ARRAY1, we would write:

```
MAKE ARRAY1[0 0] 43
```

In ExperLogo®, we would define an array to be a 6 x 6 matrix or table of objects with a command such as this:

```
MAKE ARRAY11 MAKE_ARRAY [6 6]
```

Then we could put the name "Terwilliger" in the third row, fourth column, by writing:

```
MAKE ARRAY11 [3 4] 'Terwilliger
```

Information is retrieved from an array by the same indexing method. Thus to find out what is in the third row, fourth column, of the array called ARRAY11, we could code:

```
MAKE ANSWER (ARRAY11 3 4)
  Terwilliger
```

Note that the extraction must be enclosed in parentheses. That is because the number of dimensions—and therefore, arguments—in the statement is not predetermined. ExperLogo® uses the parentheses to know when to stop looking for data about the extraction assignment.

## Predicate Commands: Testing

Logo offers a number of predicate commands. These are characterized by the letter P as their final letter. We have already discussed some of these such as EMPTYP, MEMBERP, LISTP, EQUALP, and WORDP.

One of the most useful of these commands is the KEYP instruction, which tests to see if a key has been pressed on the keyboard since the last time the program checked. NAMEP, another useful predicate function, checks to see whether a particular name has been assigned a value in the program.

All these predicate functions return a "t" or true response if the tested condition is found to be valid and a "nil" if it isn't. Therefore, the most common use of these commands is in IF constructions.

## Property Lists

The property list is a special, and useful, type of Logo list. The programs in this book make extensive use of property lists.

### General Characteristics

Structurally, a property list is composed of pairs of values. Because of its special character, a property list cannot be manipulated with the commands that work for other Logo lists.

Like an ordinary list, a property list is associated with a name. If the programmer does a good job of such things, the name will reveal something about what the list contains. For example, if we are working with descriptions of computer systems, we might name each property list after the appropriate computer: IBM_PC, FAT_MAC, C64. Each property list will then set up certain *attributes* we wish to track: maximum memory capacity, operating system, suggested retail price, availability of color, and other data. Each computer's property list would contain an entry—or *value*—for each attribute. For the property list called FAT_MAC (Apple's 512K Macintosh®), we might find that the attribute MAXMEM has the value 512 associated with it. Its operating system, defined by an attribute OS, might have the word "proprietary" tied to it. Similarly, PRICE might be 2200 and COLOR would have the value NO.

The property list for each of the other computers would have as its contents similar pairs of attribute-value relationships. In a sense, then, property lists are structured lists that are well suited to storing data for later retrieval. The property list is a superb data structure to be used when designing data base applications in Logo.

### Creating Property Lists

We don't use the usual MAKE command to create a property list. Instead, we use the Logo command PPROP—which stands for Put PROPerty—to create lists.

This command also adds more data to an existing property list, but if no property list of that name exists, Logo creates a new one.

PPROP requires three arguments. The first names the property list; the second names the attribute; the third is the value itself. For example, to put the price of a 512K Macintosh® into a property list called FAT__MAC, we would write:

```
PPROP 'FAT__MAC 'PRICE '2200
```

Notice the use of single quotation marks to delineate the contents of the list. The quotation mark before the last item, the value itself, is not used if the value to be stored is for numerical calculation. Here, the price is not available for calculation, so string storage is sufficient.

The rest of the property list for the FAT__MAC would be built the same way, with PPROP statements.

**Displaying Property Lists**

The Logo command PLIST displays the contents of a property list. We simply supply the name of the property list. Writing the Logo command:

```
PLIST 'FAT__MAC
```

will result in the computer displaying the property list for the FAT__MAC:

```
MAXMEM 512 OS PROPRIETARY PRICE 2200 COLOR NO
```

We use the Logo command GPROP (for Get PROPerty) to extract a single attribute's value from a property list. Having retrieved the piece of information, we can use it to test for conditions in IF statements, display it, or use it some other way. For example, to find out whether the FAT__MAC has color capability, we might code:

```
IF EQUALP GPROP 'FAT__MAC 'COLOR 'NO [PRINT [SORRY, NO COLOR!]]
```

**Modifying Property Lists**

We can modify a property list in two ways: add more attributes and values to it (or, by the same method, change an existing attribute's value) or remove a property from the list. As indicated earlier, the PPROP command will put a new property value on a list. If Apple releases a Color Macintosh, for example, we could update our property list by adding:

```
PPROP 'FAT__MAC 'COLOR 'YES
```

If we decide to stop tracking color availability for our computer data base, we use the REMPROP (for—you guessed it!—REMove PROPerty) command and tell Logo from which property list to remove which attribute-value pair:

```
REMPROP 'FAT__MAC 'COLOR
```

Perhaps the most common use of the REMPROP command is to reduce the size of a property list we are working with while leaving the entire property list on a disk file unchanged. If we had just read the FAT_MAC property list from a disk file of property lists describing computers and only wanted to compare maximum memory sizes, we could make our program run more efficiently by removing the other property list entries:

```
MAKE TEMPLIST :FAT_MAC
REMPROP 'TEMPLIST 'OS
REMPROP 'TEMPLIST 'PRICE
REMPROP 'TEMPLIST 'COLOR
```

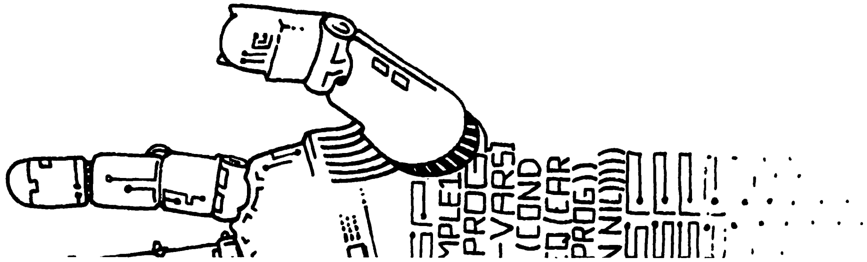
## Workspace Management

The remaining two Logo commands are used mainly in program writing rather than in program execution. The ER command will erase any object from the workspace. (The "workspace" is the name given to the portion of the computer's memory in which we are working and storing information as we develop Logo programs.) This is often useful during program debugging to ensure that no list or variable is assigned an initial incorrect value. ERALL is a closely related command that has the effect of dramatically clearing the workspace. You will probably use this only when you have reached a major milestone in your program development and want to start with fresh lists, variables, and procedures.

CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELL
ELLIGENCE ARTIFICIAL IN
L INTELLIGENCE ARTIFICIA
ICIAL INTELLIGENCE ARTI
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELL
ELLIGENCE ARTIFICIAL IN
L INTELLIGENCE ARTIFICIA
ICIAL INTELLIGENCE ARTI
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELL
ELLIGENCE ARTIFICIAL IN
L INTELLIGENCE ARTIFICIA
ICIAL INTELLIGENCE ARTI
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTEL

# 11

fresher



○ Basic LISP Syntax
○ Unusual Data Structures Lend Themselves to AI Design
○ List Manipulation Power: Key to LISP Popularity in AI
○ LISP's Extensibility

This chapter is a highly condensed presentation of the key ideas in the LISP programming language. It is aimed at the reader who is familiar with LISP and has programmed in it, but whose experience may not be recent. The chapter is neither exhaustive nor detailed. If you are unfamiliar with LISP or you wish more in-depth education in this intriguing and unique language, refer to one of the several LISP books mentioned in Appendix C.

## Introduction to LISP

LISP is an acronym that stands for LISt Processing. Since lists are generally viewed as the best way to store and manipulate information for artificially intelligent programs, LISP has become the language of choice for AI program development in the United States.

Like virtually all other programming languages, the creators of LISP began with the idea of standardizing it so that any program written in LISP would run on any computer with LISP available to it. Like virtually all other programming languages, LISP unfortunately eluded standardization. There are many dialects of LISP on the market and in use in universities and research laboratories. The most widely used dialects today are probably InterLisp, MACLISP, ZetaLisp, and Common Lisp.

### Dialects of LISP and the Mac

As of this writing, only two dialects of LISP are available for the Macintosh®: XLISP and ExperLisp. The former is a public domain program available free from many Apple bulletin boards. It appears to be primarily a MACLISP-compatible approach to the language. ExperLisp, the first implementation of LISP available for the Mac, combines ZetaLisp and Common Lisp with some new routines to accommodate the Mac's unusual interface and design.

New versions and dialects of LISP continue to become available for the Macintosh®. Because the LISP examples in Appendix A are written in ExperLISP, and because no real standard for the language exists yet, we will focus on the portions of the LISP language that are used in all dialects of the language. Where a function unique to ExperLisp is presented, I will point that out.

## Basic LISP Syntax

A first-time LISPer examining the first listing in this peculiar language notices two things about LISP programs. First, we notice that program statements—if they can be located at all in the midst of the structures—seem reversed from the usual way of thinking about languages. For example, adding two numbers in LISP requires that we write a line such as:

```
(PLUS 45 53)
```

This is different from the way we would expect to phrase this problem in English or BASIC, where we would be inclined to write (and say):

```
45 PLUS 33
```

In LISP, the *function* name always comes first in a program statement. The function name is followed by arguments or operands, which provide information for the procedure to use in carrying out its instructions.

How, in the midst of all of these functions and arguments, does LISP know where one program instruction ends and another begins? The answer lies in the huge number of parentheses present in all LISP programs.

**Parentheses Are Important. . .And Confusing!**

Parentheses *are* important in LISP; if they are not properly used, your programming results can range from unpredictable to disastrous. Each instruction in a LISP program is enclosed in parentheses; if the instruction itself contains instructions as part of *its* structure, then those instructions, too, are enclosed in parentheses.

We'll have more to say about this subject when we discuss program structure in LISP. For now, just note that parentheses are important.

## Data Structures in LISP

All data structures in LISP are referred to generically as "S-expressions" (for symbolic expressions). There are three types of S-expressions: atoms, strings, and lists.

### Atoms

An atom is the smallest data element in LISP. Atoms are of two types: literal and numeric. A literal atom consists of a letter which may be followed by additional letters or digits. The following are all literal atoms in LISP:

```
FOO
F1
F
FOOLISH-ATOMS
```

A numeric atom may begin with an optional plus or minus sign, to be followed by a digit, which may be followed by additional digits. The following are all numeric atoms in LISP:

```
+14
-11
23
987654321
```

### Strings

Strings in LISP are always enclosed within double quotation marks. Anything that appears between two sets of such marks is a string. The following are all legitimate strings in LISP:

```
"This is a string"
"This is also a string"
"This is a (string) with parentheses"
"test-it-yourself"
```

**Lists**

A list is defined (somewhat confusingly) as a collection of other S-expressions, though it is itself also an S-expression. You can always recognize a list in LISP, though, because it begins with a left parenthesis and ends with a right parenthesis. In between, a list contains any number and any combination of types of other LISP S-expressions with individual elements separated by spaces. Here are some basic lists:

```
(LIST1)
(ITEM1 ITEM2 ITEM3)
(A B C D E)
(APPLES ORANGES BANANAS MANGOS KIWI PLUMS TANGARINES GRAPEFRUIT)
```

Unlike most other programming languages, LISP uses the parentheses not only as delimiters but also as important parts of the S-expressions being defined. The list:

```
(X)
```

is a list containing the atom "X." But the list:

```
((X))
```

is a list that contains a list which is in turn a list containing the atom X. This distinction is important and is perhaps the single most difficult concept to grasp about LISP data structures.

## Program Structure in LISP

At a basic level, LISP program structure is elegantly straightforward: every S-expression is a *syntactically* legal LISP program. This is because LISP treats programs written in it as lists that can be manipulated, giving the effect of a person looking in a mirror that reflects a different mirror and so on, producing a nearly endless series of images. A LISP program is a list containing lists of lists that in turn comprise lists, and so on.

Just because every S-expression is a legal LISP program, however, don't suppose that every S-expression is a useful or functional LISP program. Functionally correct LISP programs are subject to being evaluated as proper by the LISP interpreter or compiler. This evaluation process is designed so that one S-expression is handled at a time. The evaluation of each S-expression results is another S-expression. In the following examples, this is depicted by the expression:

**S-expression** → value

According to Dr. Eugene Cherniak (1980), the LISP interpreter has four basic rules it follows:

**LISP Interpreter Rules**

• *RULE 1.* If the expression is either a number or "T" (for true) or "nil" (for false or empty), then the expression's *value* is itself. So if we type into the LISP interpreter the S-expression:

```
5
```

the LISP interpreter will return to us the value 5.

• *RULE 2.* If the S-expression consists of a function followed by one or more arguments, then the interpreter evaluates each argument (which, of course, may in turn be an S-expression) and then calls on the function with these values as arguments. For example, addition in LISP is handled by a function called variously +, PLUS, or SUM. ExperLisp permits the first and third variations. Multiplication is called for by the function *, PRODUCT, TIMES, or MULTIPLY. ExperLisp permits both * and PRODUCT to signify multiplication. Thus the following expression evaluation would take place with the S-expressions shown:

```
(SUM 14 8)→ 22
(PRODUCT 11 7)→ 77
(SUM 14 8 (PRODUCT 11 7))→ 99
```

• *RULE 3.* If the S-expression is a list beginning with a reserved word that is not a function, the value depends on the way the reserved word is implemented. This is a subtle but important variation on rule 2. For example, a reserved word common to all LISPs is SETQ, which assigns a value to a variable.

```
(SETQ X 5)
```

makes the variable X have the value 5. (In LISP terminology, this is called "binding" a variable.) The X is not evaluated; it is merely used. But the expression which follows the variable name *is* evaluated. Thus a LISP list which said:

```
(SETQ X (SUM 14 8 (PRODUCT 7 11)))
```

would bind the value 99 to the variable X.

• *RULE 4.* If the S-expression is an atom, its value is the last value assigned to it. If no value is assigned, an error condition results. After we've carried out the last SETQ operation, typing the atom X results in the value 99 being returned:

```
X → 99
```

There is one other thing you should know about SETQ. If it is designed to bind a literal string to a variable name, the literal string must be preceded by a single quotation mark. Thus we would have such constructions as:

```
(SETQ A1 'TESTING)
(SETQ ITEM43 'CARPET_CLEANER)
```

As you can see, except for the operation of reserved words, the interpretation of LISP S-expressions is simple and straightforward.

The bulk of the rest of this chapter deals with LISP functions and reserved words and how they work.

## Operations on Lists

Since lists are the primary building blocks of LISP data structures and programs, it's not surprising that a great many built-in functions can operate on lists to extract information from them, create and modify them, and analyze their contents.

### CAR and CDR

The two most basic and widely used LISP list manipulation functions are CAR and CDR (pronounced "cudder" to rhyme with "rudder"). CAR evaluates a list and returns the first element of that list (i.e., the first S-expression). CDR evaluates a list and returns everything *except* the first S-expression. Starting with the list (APPLES ORANGES BANANAS WATERMELON), CAR would return APPLES and CDR would return the *list* (ORANGES BANANAS WATERMELON). (Note that CAR does not return a list unless the first S-expression is also a list.)

The following exchange with a LISP interpreter will be self-explanatory and will clarify how CAR and CDR work on different kinds of lists. (The material typed by the user is printed in ALL CAPS and the information typed by the interpreter is all lower case.)

```
(SETQ LIST1 '(DAN CAROLYN (DON RAE)))
(dan carolyn (don rae)) ← LISP always prints what it returns
(SETQ LIST2 '((BOB VAL) ALBERT MARILEE))
((bob val) albert marilee))
(CAR LIST1)
dan
(CDR LIST1)
(carolyn (don rae))
(CAR LIST2)
(bob val)
(CDR LIST2)
(albert marilee)
```

### Combinations of CAR and CDR

Using just CAR and CDR, we can extract any portion of the list that we want. It may get complicated, but it is possible. We just use CAR and CDR in various combinations. For example, if we start with the list (DAN (BOB VAL) CAROLYN (ALBERT MARILEE)) and want to extract CAROLYN from this list, we would write:

```
(CAR (CDR (CDR LIST-OF-NAMES)))
```

This sequence would be evaluated from the inside out. First the system would evaluate (CDR LIST-OF-NAMES), dropping DAN from the list. It would then take that resulting list and evaluate the CDR of it, resulting in the list (BOB VAL) being eliminated. CAROLYN is now the first element of the remaining list, so CAR extracts her name and we're done.

You can see how complicated such a process could get. Add to that complexity the fact that this is one of the most commonly needed operations in LISP programming, and it's no wonder that most implementations of the language include shorthand ways of doing these things. ExperLisp has no less than *28* such special combination functions that permit fast extraction of information from lists. Each such special function begins with the letter "C" and ends with the letter "R" and has (in the case of ExperLisp) up to four letters in between. Each letter is an "A" (for CAR) or a "D" (for CDR). Our example above would be handled in ExperLisp by coding:

```
(CADDR LIST-OF-NAMES)
```

The first A executes a CAR, the first D a CDR, and the second D another CDR; the combined CADDR yields the same result as our example which called those three operations separately.

**CONS and LIST for List Construction**

We can use one of two LISP functions to build a list: CONS or LIST. The choice of which to use depends on the kind of list to be constructed.

CONS is short for CONStruct. It takes two arguments and puts the first argument onto the list made up of its second element. Writing:

```
(CONS 'WATERMELON '(APPLE ORANGE))
```

leads to a new list that has WATERMELON as its first element:

```
(WATERMELON APPLE ORANGE)
```

To build a list from scratch, we use the special LISP value nil as the right-hand argument:

```
(CONS 'WATERMELON NIL)
```

which produces the single-element list (WATERMELON). From that, we can add another element:

```
(CONS 'APPLE '(WATERMELON))
```

CONS always requires two arguments. To build a two-element list with a watermelon and an apple as its members we would have to write:

```
(CONS (CONS 'APPLE NIL) 'WATERMELON)
```

LIST provides a more straightforward way of doing this. It takes any number of arguments and simply puts them all into one list. Thus we could write:

```
(LIST 'APPLE 'WATERMELON)
```

or even:

```
(LIST 'APPLE 'WATERMELON 'MANGO 'PEACH 'PLUM 'PEAR)
```

and LISP would create a list of the appropriate size.

We can also create a list by the same means we used to create variables in LISP, using SETQ or one of its close variations. We'll have more to say about this later.

**APPENDing to a List**

The APPEND function in LISP combines two lists into one large list. (The technical name for this is *concatenation*.) The following illustration demonstrates the use of APPEND.

```
(APPEND '(PLEASE MAKE) '(A BIGGER LIST))
(please make a bigger list)
```

APPEND normally requires precisely two arguments, both of which must be lists. But in ExperLogo, we can use as many arguments as we wish. All arguments must still be lists, however. The following statement is perfectly legal in ExperLisp:

```
(APPEND '(PLEASE MAKE) '(THIS A) '(BIGGER) '(LIST))
```

but it would not be permitted in most LISP dialects.

**SUBSTituting Items in a List**

To replace every occurrence of one symbol for another each time the latter appears in a string or list, we use the SUBST function. The following interaction with the LISP interpreter will clarify what is meant.

```
(SUBST 'THE 'A '((A GIRL)(AND A BOY)(WALKED DOWN A STREET)))
((the girl)(and the boy)(walked down the street))
```

The first argument is the new value to be substituted, the second is the value to be replaced, and the third is the list (or the name of the list if it is bound to a variable).

**MEMBER**

Strictly speaking, the MEMBER reserved word isn't a list manipulator, but it does work on a list. It finds out whether the first argument is found in the second argument. It returns the part of the list starting with the word to be found, or it returns "nil" if the word isn't present.

An example may help to clarify the operation of the MEMBER reserved word.

If we begin with the list (BEEF PORK CHICKEN FISH) and wish to determine if PORK is among its members, we might find ourselves doing this:

```
(SETQ 'FOOD1 '(BEEF PORK CHICKEN FISH))
(beef pork chicken fish)
```

```
(MEMBER 'PORK FOOD1)
(pork chicken fish)
```

**Many Other Functions**

As I said at the outset, there are a great many commands designed to be used in conjunction with lists in LISP. Check your particular version of LISP to see what other capabilities it has.

## Defining Functions

Like most useful modern computer languages, LISP is extensible. Simply put, "extensible" means that we can define new procedures or functions which we can then use in our programs. This is equivalent to inventing a new LISP function or reserved word, one feature of LISP that makes the language especially intriguing.

The LISP function DEFUN is perhaps the most commonly used means of creating new functions in AI programs. The structure of a DEFUN instruction is:

```
(DEFUN NAME (ARG-LIST) (BODY))
```

When DEFUN is carried out, the LISP interpreter returns NAME as the value. This means that when DEFUN is used successfully to create a new function in LISP, its name is returned by the interpreter. (There is, of course, no guarantee that the function created was the one you wanted to create!)

To *use* the function, you use the newly defined NAME and pass whatever arguments are required, as defined by the ARG-LIST. The BODY portion of a DEFUN instruction provides the list of operations to be carried out by LISP when the function is called. Let's see how this works.

```
(DEFUN CUBE (N) (PRODUCT N N N))
cube
(CUBE 3)
27 ← typed by computer in answer to previous instruction
(CUBE)
0 arguments sent, 1 required ← error message may vary with version of LISP
```

## Setting Up Variables

Variables are an important part of LISP, as they are with all useful programming languages. We use the reserved word SETQ to define variables and give them their initial values. This command takes two arguments. The first is a variable name. The second is an S-expression that LISP evaluates in accordance with its rules in order to determine a value to assign to the variable name.

We have already encountered SETQ, so we need not spend a great deal of time on it here. The following interaction exemplifies the use of variables in LISP with SETQ. (Assume the LISP session has just started; nothing yet is set up.)

```
R
unbound variable R ← message may vary; LISP doesn't "know" R
(SETQ R 'TRIAL)
trial
R
trial
(SETQ R (SUM 4 5 6))
15 ← computer's response
R
15 ← computer's response
```

We begin a LISP session with no variables bound to values. Typing a variable's name will result in an error message indicating the variable is unbound. SETQ assigns the variable a value; to change that value, use SETQ again with the same variable name.

## Conditional Processing

AI programs, more than any other kind, require a great deal of conditional processing. We want the program to react differently to various kinds of inputs or situations. An expert system such as the one we developed in Chapter 8 must be able to analyze numerous inputs in various combinations and "learn" how to react to each new set of combinations it faces.

LISP has two basic conditional processing instructions: IF and COND. In many ways they are similar; some AI programmers would argue that they are stylistically different but functionally identical. But syntactically, they are quite different, so we will examine them both briefly here.

### The IF Statement

The IF statement in LISP has three arguments: a condition to be tested, a list of instructions to carry out if the condition produces a result of T (true), and a list of instructions to carry out if the condition produces a result of NIL (false). The third argument is optional.

The condition to be tested is usually a predicate form that evaluates information and compares it to other information. There are several such predicate forms in LISP. For now, don't worry about the purpose of a particular predicate form; focus instead on the use of the form in an IF statement.

In LISP, a predicate is any function that returns a result of T if it evaluates to true and NIL if it evaluates to false. These yes-no forms usually end in the letter P to remind us that they are predicate forms.

Here's a small example of an IF construction in LISP:

```
(IF (LISTP (X)) 'YES 'NO)
```

This statement group would take an input called "X" and determine if it was a list. If so, it would print "YES"; if not, it would print "NO." The conditional structure here is

```
(LISTP (X))
```

which uses one of LISP's predicate forms. The first instruction list is the simple literal S-expression 'YES, which prints YES. This instruction will execute if LISTP indicates that X is a list. Otherwise, the second instruction list—another simple literal S-expression 'NO—prints "NO" if X is not a list.

**COND Constructions**

More traditional LISP implementations use COND rather than IF. ExperLisp uses both forms, as do most modern implementations of the language. COND consists of a series of conditional clauses, each of which contains a condition to be tested and a list of instructions to be carried out if the condition evaluates to be true. A COND construction stops executing the instant it reaches a clause with a condition that tests to be true.

An advantage of the COND form is that it permits easy implementation of a number of consecutive circumstances to be checked, whereas IF is limited to a true-false result of one condition. (Actually, we can use *nested IF* statements to achieve something like a COND construction, but it is far less efficient to do so.) The following example tests for five conditions and acts based on which condition tests true.

```
(DEFUN MENU-CHECK (ITEM1)
  (COND ((EQUAL ITEM1 1) (PRODUCT 10 10)
    ((EQUAL ITEM1 2) (PRODUCT 10 10 10)
    ((EQUAL ITEM1 3) (PRODUCT 5 5 5 5)
    ((EQUAL ITEM1 4) (SUM 1 2 3 4)
    ((EQUAL ITEM1 5) (QUOTIENT 475 5))))))))
```

This function examines a variable called ITEM1. If it is a 1, the function multiplies 10 by 10, prints the results, and quits. If ITEM1 is a 2, the function multiplies 10 by 10 by 10, prints the results, and quits. You can follow the other steps in the COND construction.

***Using T as a Test Condition*** The COND construction has a fatal flaw: if the value being tested (in the example, ITEM1) meets none of the values shown, an error results. The error will vary from system to system and LISP to LISP, but an error will definitely appear. To program around this problem, LISP software designers use T—for true—as a condition to be tested against. T is always true; that is its meaning. So if we put a T followed by a statement list at the end of each COND construction, we never encounter a situation the program won't handle. Following that procedure, our example becomes:

```
(DEFUN MENU-CHECK (ITEM1)
  (COND ((EQUAL ITEM1 1) (PRODUCT 10 10)
    ((EQUAL ITEM1 2) (PRODUCT 10 10 10)
    ((EQUAL ITEM1 3) (PRODUCT 5 5 5 5)
    ((EQUAL ITEM1 4) (SUM 1 2 3 4)
    ((EQUAL ITEM1 5) (QUOTIENT 475 5)
    (T 'HUH?))))))))
```

Now when we run MENU-CHECK with ITEM1 set to, say, 45, the program will simply print "HUH?" on the screen and go on.

## Predicate Forms for Testing

As we have indicated, LISP contains a number of predicate forms for testing conditions in the system. A complete discussion of these forms is beyond the scope of this refresher, but the more significant ones are ARRAYP, BOUNDP, CHARACTERP, EQ, EQUAL, FUNCTIONP, LISTP, NUMBERP, SYMBOLP, and VARIABLEP.

***EQUAL Is Not the Same as EQ*** One of the most confusing things about LISP is the way it evaluates S-expressions for equality with one another. In fact, it's so confusing that LISP has not one but two different predicate forms for testing for equality: EQUAL and EQ (pronounced "eek").

EQ is perhaps the more commonly used form. EQ will evaluate to a true result *only* when the items given it are *identical* in every respect. In fact, what EQ does is to examine the computer address at which the information for the two variables is stored. If both are at the same address, then they are EQ. Otherwise, even though the information may *look* the same, it is not EQ.

Using EQUAL results in slightly slower execution. The command checks the contents of variables rather than their address. EQUAL matches each element to determine if one variable is equal to another.

The following interaction demonstrates the differences between EQ and EQUAL.

```
(SETQ A1 '(A B C))
(a b c)
(SETQ B1 A1)
(a b c)
(SETQ C1 '(A B C))
(a b c)
(EQUAL A1 B1)
t
(EQUAL A1 C1)
t
(EQ A1 B1)
t
(EQ A1 C1)
nil
```

All three lists—A1, B1, and C1—have the same *contents*: the list (A B C). But lists A1 and B1 occupy the same address in the computer's memory because we defined them to be identical. B1 *is* A1. The list C1, on the other hand, was created separately and although its contents are EQUAL to those of A1 and B1, it doesn't have the same computer address; it is a separate list.

Most LISPs, including ExperLisp, have variations of some or all of their predicate forms, one variation that uses EQ and another that uses EQUAL. The results of using these are often the same, but the difference is crucial in some situations.

## Combining Predicate Forms: Logical Connectors

Often we want to construct predicate forms that are more complex than a simple one-element test for equality or for the presence or absence of a list. LISP provides two logical connectors that make the task of constructing such complex forms straightforward, if not simple. These two connectors are AND and OR. Using them in combination, we can create highly complex IF and COND constructions.

When AND connects two conditional constructions, it means that *both* conditions must produce a true result before the condition is satisfied. When OR is used, it means that if either condition—or both of them—produces a true result, the condition is satisfied.

Like other LISP functions, the two connectors precede the list of information they connect. To test if it is true that a variable called "A" is "HELP" and "B" is less than 20, we would write:

```
(IF (AND (EQUAL A 'HELP) (B 20)) 'OKAY 'REJECT)
```

If the conditions outlined are both true—i.e., A *is* HELP and B *is* less than 20—the 'OKAY S-expression will evaluate, and the word "OKAY" will print. Otherwise, the computer will "REJECT" the input. To accept the entry if either condition is true, we would code:

```
(IF (OR (EQUAL A 'HELP)(< B 20)) 'OKAY 'REJECT)
```

## Essential Input/Output

Basic I/O routines are common to all LISP implementations, even though there are many variations on the themes and nuances in these variations from one LISP to another. The basic I/O functions are READ and PRINT. These two commands get information from the user at the keyboard and display results on the screen or printer.

### READing from Keyboard

The READ function in LISP creates a pause in the execution of a program and waits for data to be entered from the current input device (the keyboard, unless it is explicitly changed). The input continues until encountering a carriage return.

A common use of the READ function is to ask for and receive a value from the user during program execution. Generally, this process would use an S-expression such as:

```
(SETQ X (READ))
```

The READ function also retrieves information from a disk file. We'll discuss the processing in disk I/O later.

### PRINTing Results and Requests

Most LISP output functions use PRINT. This function simply prints whatever information it is given *and returns the information as its value*. This means that information displayed by LISP will usually print twice on the screen.

The information to be printed may be enclosed in quotation marks, guillemets, or parentheses, or it may be preceded by a single quotation mark.

PRINT can display strings or the values of variables. Here are some examples of PRINT (with the computer's response indented).

```
(SETQ XY 43)
  43
(PRINT 'TESTING)
  testing
  testing
(PRINT '(TESTING))
  (testing)
  (testing)
(PRINT XY)
  43
  43
XY
  43 ← only one line when we type variable name without PRINT

(PRINT (TESTING))
the value TESTING is not a valid function
```

The last example shows what happens when we put something we intend to print inside parentheses as a literal string but forget to use the single quotation mark in front of it.

Note that the computer prints a result twice when we use the PRINT function. That's because PRINT, like all LISP functions, must return a result. In the case of PRINT, its result is the item it has been told to print. So, the first occurrence of the word "testing" is the result of PRINT carrying out our instructions. The second occurrence is a result of PRINT's requirement to return a result.

***Variations of PRINT*** LISP offers two (in some dialects, even more) variations of PRINT. PRINC acts the same as PRINT except that it doesn't insert a carriage return at the end of the line. It is most useful for printing prompts on the screen when the answer will look best appearing at the end of the line requesting the information.

PRIN1 displays the information exactly as it is given, including punctuation marks and special characters which might otherwise be suppressed by PRINT. Its primary use is in disk file I/O, where we want to ensure that special symbols are stored on the disk for later recovery.

## Disk File I/O Overview

Using a disk file on the Mac under ExperLisp requires that the disk file be properly opened. Data may then be placed in the file using PRINT or PRIN1 where special characters form a key part of the information to be stored. Information is retrieved using READ.

***OPENing Disk Files*** The three OPEN commands in ExperLisp are: OPEN_WRITE, OPEN_READ, and OPEN_APPEND. OPEN_WRITE creates a new file of the name supplied. If a file with that name already exists, OPEN_WRITE writes the new information over the contents of the old file. If no file exists with the requested name, ExperLisp creates one.

OPEN_READ opens a file from which we wish to retrieve information. No data can be written to a file that has been opened for reading only.

OPEN_APPEND permits us to add information to an existing file without erasing the file's present contents. If no file of the name you provide exists, ExperLisp creates it.

With all OPEN commands, the programmer supplies the file name as a string in quotation marks. To facilitate later use of the file, we assign its "stream" (a LISP term referring to the I/O process) as a variable using SETQ. Once the file is open, READ and PRINT retrieve information from and write data to the file, respectively. When we are done with a file, we close it with the CLOSE command.

The following dialog with ExperLisp demonstrates the process.

```
(SETQ CURFILE (OPEN_WRITE "TEST1")) ← file "TEST1" is CURFILE
  function #540111B4 ← response value may vary
(PRINT '(HELLO THERE) CURFILE) ← writes first record
  (hello there)
(CLOSE CURFILE)
  t
(SETQ CURFILE (OPEN_READ "TEST1)))
  function #540111B4 ← response value may vary
(PRINT (READ CURFILE))
  (hello there)
(CLOSE CURFILE)
  t
(SETQ CURFILE (OPEN_APPEND "TEST1)))
  function #540111B4 ← response value may vary
(PRINT '(ARISTOTLE THINKS) CURFILE)
  (aristotle thinks)
(CLOSE CURFILE)
  t
(SETQ CURFILE (OPEN_READ) "TEST1)))
  function #540111B4 ← response value may vary
(PRINT (READ CURFILE))
  (hello there)
(PRINT (READ CURFILE))
  (aristotle thinks)
(CLOSE CURFILE)
  t
```

ExperLisp adds three other file I/O functions: WITH_OPEN_READ, WITH_OPEN_WRITE, and WITH_OPEN_APPEND. These operations are similar to those we have just reviewed except that these permit a body of data to accompany the commands so that one instruction can open a file *and* write data to it. This approach works well for adding a small amount of information to a file (i.e., one or two records), but it is inefficient in other applications.

## Math Operators and Functions

Because function is the first element in a LISP instruction list, math is carried out using *prefix* notation, as in some handheld calculators—especially those made by Hewlett-Packard. In other words, the math operation to be performed precedes the values. This takes some getting used to but it is not difficult to manage.

The usual math symbols are available in LISP— + for addition, – for subtraction, / for division, and * for multiplication. The following are examples of the proper use of these functions in LISP.

```
(+ 4 3)
7
(– 43 11)
32
(/ 88 6)
14.66666666
(* 12 18)
216
(+ 4 (– 43 11))
36
```

LISP also allows some words to carry out the math functions: SUM for addition, DIFFERENCE for subtraction, QUOTIENT for division, and PRODUCT or TIMES for multiplication. ExperLisp uses SUM, DIFFERENCE, QUOTIENT, and PRODUCT. (The use of these words in newer LISP programs reflects the desire to make them compatible with older ones. For there is no other logical reason, except perhaps readability, to type PRODUCT instead of "*.") The function names are used as their corresponding math symbols are.

Another LISP function, REMAINDER, is quite useful in division problems. In the example in which we divided 88 by 6 and got 14.666666, REMAINDER would return the value 4, the remainder when we divide 88 by 6 and get 14.

### ADD1 and SUB1

It is often necessary in computer programs to add 1 to or subtract 1 from a number. This is particularly important when using counters to keep track of how many times we've done something, how many lines we've printed on a report, and the like. LISP provides two built-in functions to accomplish these tasks, called ADD1 and SUB1. Here is how they work:

```
(SUB1 43)
42
(ADD1 43)
44
(SUB1 (ADD1 43))
43
```

### RANDOMness

When designing games or creating codes or encryption techniques, we sometimes want unpredictability as a result. LISP offers a RANDOM function to generate a random number between 0 and any number we choose. The function (RANDOM 1234) will return a number between 0 and 1233. Normally, this is not what we want: a result of 0 will be unusable and we really want the possibility of having 1234 as an answer. So we usually use RANDOM in conjunction with ADD1, like this:

```
(SETQ VAL1 (ADD1 (RANDOM 1234)))
576 ← response will vary
```

**ROUND and TRUNCATE**

Sometimes, we want numeric results rounded off. LISP's ROUND function enables us to do this. It rounds its argument to the nearest integer number, raising it to the next higher value if the decimal part is 0.5 or greater and leaving it as is if the decimal part is less than 0.5.

TRUNCATE is related to ROUND. It eliminates the fractional part of a number, but does so simply by lopping it off, not by rounding the integer part of the number.

```
(ROUND 15.49)
15
(ROUND 15.5)
16
(TRUNCATE 15.49)
15
(TRUNCATE 15.5)
15
(TRUNCATE 15.9999999)
15
```

**Trigonometric Functions**

LISP provides access to the common trigonometric functions of ACOS, ASIN, ATAN, COS, SIN, and TAN. The angle must be supplied in radians.

**Miscellaneous Math Operations**

We can calculate the square root of a number with the LISP built-in function SQRT, as shown here:

```
(SQRT 25)
5
```

Natural logarithms of numbers can be calculated using LN and LN1 (the latter calculating the natural logarithm of 1 greater than the value furnished). Thus (LN 3) and (LN1 2) have the same answers. We can calculate logarithms to the base 2 with LOG2 and LOGB. The latter truncates the answer as if LOG2 had been used.

The "absolute value" of a number is the value when the sign is ignored. The absolute value of a number can be obtained with the LISP function ABS. (DIFFERENCE 2 11), for example, returns a −9, while (ABS (DIFFERENCE 2 11)) returns a 9.

ExperLisp includes functions that determine the larger or largest or the smaller or smallest of two or more numbers. These functions are called, respectively, MAX and MIN. Here's how they work:

```
(MIN 4 8 16 9 23 96 14)
4
(MAX 4 8 16 9 23 96 14)
96
```

It is also possible in ExperLisp to determine whether a particular numerical atom is odd or even using ODDP and EVENP. As with most predicate forms (see our discussion of "Conditional Processing"), these return the results of "t" or "nil."

## LISP: Rich and Extensible

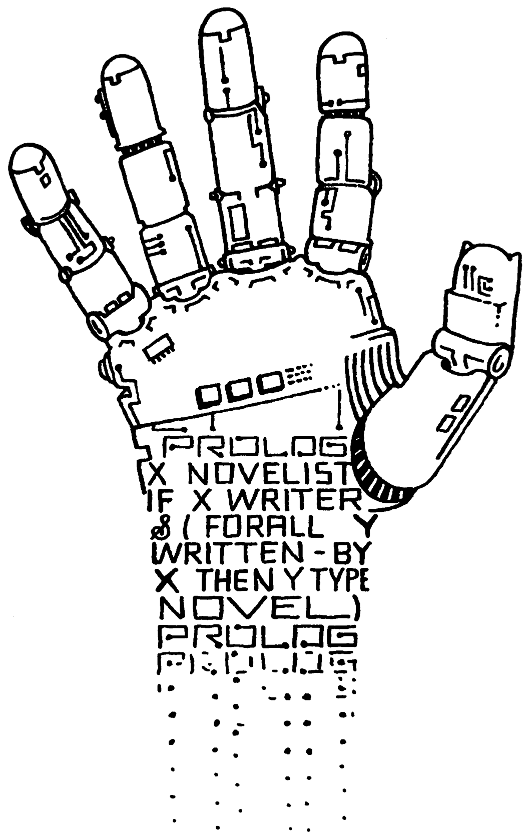
We have only been able to cover here some of the most important LISP functions. LISP is a rich language and, because of its extensibility, it has as many variations and implementations as there are people who want to use it. As with all languages, the key to becoming efficient in LISP lies in being willing to try new ideas and to explore the language.

Until recently, there were no easily accessible texts on LISP that a person without a great deal of programming background could grasp. That void has been filled and the annotated bibliography in Appendix C offers some excellent choices.

CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI

12

# A Brief Prolog Tutorial



- Is Prolog a Language? Does it Matter?
- Describing Data Relationships in Prolog
- Nonprocedural Language: Basic Concepts
- Creating Rules with Conditionals
- List-Processing Power

## Introduction and Purpose

This chapter presents a brief overview and tutorial of the basic concepts in Prolog. My purpose is not to train you in the ins and outs of Prolog programming, but to provide a refresher for you if you have previously worked with Prolog. If you have no exposure to Prolog, you should be able to learn enough in this chapter to undertake minimal programming tasks.

Unlike Logo, the language used for most of the programs in this book, and LISP, the most widely used AI language, Prolog may seem unrelated to the purpose of this book. But, I have chosen to include it for a host of reasons, two of which are paramount.

First, it could be argued that an introduction to the world of artificial intelligence on micros that didn't include a Prolog discussion would be woefully inadequate to its task of preparing the reader for the real world of AI. Much AI work is being done in Prolog; Prolog dialects and derivatives are used at AI research facilities all over the world. It is without doubt the second most important AI language available today.

Second, and of more immediate concern, since we spend a fair amount of time in this book discussing a scaled-down version of Prolog (Prologo in Chapter 7), it seems fitting that you should get a feel for the power,' flexibility, and structure of the language in a full implementation. Otherwise, you might be left with an unfair impression of the limitations of Prolog.

### Prolog: Is It a Language?

There are those who argue that Prolog is not really a *language* at all but more a method of describing decision-making to a computer. To be sure, when compared with most computer languages, Prolog lacks certain things that the others (more or less) have in common. This is because it is a totally different way of looking at the task of programming.

In *micro-PROLOG: Programming in Logic*, Clark and McCabe depart from the tendency to describe most computer languages as procedural and Prolog as nonprocedural. Instead, they refer to Prolog as a *descriptive* language and other programming languages as *imperative* by nature. This distinction has some value.

### What's the Difference?

In BASIC, LISP, Logo, Pascal, and virtually all other computer programming languages, programs consist of sets of commands or instructions to the computer. "Do this. Now do that. If this is true, do this; otherwise, do that." We are speaking in the imperative, or command, voice of the language. In other words, we tell the computer how to do what we want done, step-by-step.

Prolog programs, however, describe a set of information and knowledge and a set of relationships among those pieces of data. Running a Prolog program consists of asking the program to respond to certain queries. To respond, the program needs to understand which data to manipulate, how it is arranged, and what the question is. We need not tell the program to enter or stop running loops, to count values up or down to keep track of where we are in a system, or provide any other *directions*. Instead, we say, in effect, "Here is some knowledge. Given that that information is true, tell me this. . . ." We don't explain to the program at any point *how* to get the information, only that it is available and that we want it.

**How Prolog Programs Compute**

As we ask questions of Prolog, it computes answers by looking for values in its knowledge base that match up with the variable(s) contained in the inquiry. Each match it finds must, of course, derive from the definitions of the knowledge furnished in the knowledge base. Prolog accomplishes this by searching through all sentences for each condition in an inquiry and matching the condition with the conclusion of the sentence. Each time it finds a match, it looks at the new sentence from the knowledge base to see if, in turn, it has preconditions which must be met to solve the original problem.

When it finds preconditions in the new sentence, it goes through the same pattern-matching process with the newly formed query. If it doesn't find preconditions, it has solved the problem—i.e., located the answer to the query—and stops processing and reports its findings to the user.

The process of matching knowledge to conditions in order to form a new query is called "pattern-directed, rule-based programming." This approach to AI programming is finding increasing use in the field, particularly in expert system program development.

If this sounds a bit abstract, we'll look at some concrete examples of this after we discuss Prolog syntax. For now, we understand that Prolog programming is different in a fundamental way from programming in any other language.

**Knowledge Base Description**

The primary task of Prolog programming is describing the knowledge base with which we want to work. From a logical viewpoint, a knowledge base is nothing more than a set of facts that defines relationships. For example, we might have a knowledge base containing information about world population. Two of its records might be:

```
United-States population 230
China population 842
```

Of course it is easy for us to read such a knowledge base to determine what the computer "knows." That is one of the beauties of Prolog programming: it has many elements of intuitive clarity.

Now, to query the knowledge base about the population of the United States, we can type a sentence such as this:

```
which(x : United-States population x)
```

(For the moment, don't worry about the syntax; we'll explore that subject shortly.) In technical English, this sentence is translated: "Which variable x can be found in this knowledge base such that it properly completes the logic sentence 'United-States population x'?" The answer comes back "230." The first x inside the parentheses results in the answer being displayed on the terminal.

**Invertible Programs**

Our use of queries to access information in a knowledge base in Prolog leads to an interesting phenomenon called "invertible programs." To demonstrate, look at a built-in arithmetic function in Prolog called TIMES. TIMES is part of the knowledge base that composes Prolog, so we need not specifically define it. It is set up by the system to describe a fact such as:

```
TIMES(x y z)
```

This statement is true only if x multiplied by y results in the answer z. TIMES(4 8 32) is a true statement. To multiply, then, we would code a Prolog sentence such as:

```
which(x : TIMES (4 8 x))
```

The answer comes back "32." In essence, we could think of this process as looking up the answer in a multiplication table, which consists of a set of facts about multiplying.

The invertible nature of this approach is evident when we decide to *divide* 32 by 8. In that event, our Prolog inquiry might be:

```
which(x : TIMES(32 x 8))
```

Notice that there need not be any relationship between the variable names used in queries and those used in describing the fact in the knowledge base. We are free to use any legal variable in a query. Prolog simply remembers the positions of each unknown in a statement of factual information.

**Upper-Lower Case Significance**

In virtually all Prolog implementations, upper and lower case are significant; "x" is not the same thing as "X." This is important to remember when writing Prolog programs. Most programming languages ignore case, so we might forget that in Prolog that is not the situation.

## Describing Data Relationships

Essentially, programming in Prolog can be thought of as consisting of two steps: setting up a knowledge base and querying it. This section describes the process of setting up a Prolog knowledge base. Querying its contents will be covered in the next section.

**Prolog Sentences**

Three types of Prolog sentences can describe facts in a knowledge base. All other knowledge base descriptions build on these three basic sentences.

***Binary Relations*** The first sentence type describes a *binary* (i.e., two-way) relationship between a pair of individuals. It has the same form as our examples describing the population of countries.

```
first-individual name-of-relationship second-individual
```

The name-of-relationship can be any word group (see the "Name Rules" section) that describes a connection between two individual pieces of information.

In the world population example, each of the first individuals is a country name and each of the second is population figure in millions. We could have changed "population" to "has-population-of," which would be a more understandable word group for inquiries, but the shorter form is easier to work with during queries and is often preferred by Prolog programmers.

***Property Sentences*** A sentence may want to give us some information describing a specific individual rather than describing a relationship between two individuals. In this case, a Prolog sentence takes the form:

**individual-name property-name**

For example, we might wish to record in our world information knowledge base that the United States is a Western nation while China is an Asian nation. We could do that with the following two Prolog sentences:

**United-States Western**
**China Asian**

***Nonbinary Sentences*** Sentences about relationships that aren't pairings of values and value-holders are written as follows:

**relationship-name(individual-1 individual-2. . .individual-n)**

There are two occasions for the use of this form of Prolog sentence. The first has to do with list processing (discussed later), where one relationship applies to a number of individuals separately, as in:

**hard-working(Dan Don Ken John)**

The other situation in which this form of Prolog sentence is used is where a single relationship ties together three or more individual pieces of information. In natural language processing, for example, "Steve threw Mary the ball" could be represented in Prolog as:

**threw(Steve Mary ball)**

Such a form makes pattern-matching in natural language processing much easier than requiring that the sentence be in its usually displayed form.

Notice that the name of the relationship that exists among the individuals comes first and that individual pieces of information to which the relationship applies are enclosed together in parentheses, separated by spaces.

**Name Rules**

Prolog has only two rules about names of individuals or relationships: they must begin with a letter (not a number or symbol) and they must not include spaces. By convention, most Prolog programmers use hyphens to connect what would otherwise be separate words in describing relationships and naming individuals in a knowledge base. Thus the fact that Steven is the father of Steven, Jr., would be described:

**Steven father-of Steven-Jr**

**Listing Prolog Programs**
While the methods for listing a Prolog program are to some degree implementation-dependent, virtually all Prologs trigger the action by using the imperative "list." There are two forms of the "list" command. "List all" lists all the relationships, facts, and rules of which the knowledge base is constructed. To see only the statements relative to a specific relationship, we use a command like "list population-of."

**Modifying Prolog Programs**
Making changes to Prolog knowledge bases is also somewhat implementation-dependent. Some Prologs on microcomputers come with full screen editors which make modification quite simple. But within a Prolog program and in virtually all Prolog implementations, we can add new sentences to the knowledge base and can delete either individual items of information or whole collections of relationships.

***ADDing Sentences*** In most Prologs, we put new information into the knowledge base by typing:

```
add SE
```

where SE is a sentence. This command adds the SE to the end of that part of the list of the knowledge base where information about the relationship is stored. Prolog keeps all sentences about one relationship together, regardless of the order in which the information is entered.

If we want to have a newly added fact become the third item in the group of sentences on the relationship involved, we can use the alternate form:

```
add 3 SE
```

If we use this form and there are fewer than two sentences already in the knowledge base, the sentence SE simply becomes the last sentence. In other words, it acts like "add" without a number.

It is important to understand that we can add facts to the knowledge base dynamically, (i.e., during program execution). Thus the knowledge base can grow as we gain experience and as the program encounters more items to assimilate.

***DELETE and KILL*** To remove a specific sentence from the knowledge base, we simply use a "delete" command. Thus, if we have changed our minds and no longer need to keep track of China's population (or, more likely, we want to update it), we type:

```
delete China population 842
```

On the other hand, perhaps we only want to delete the last sentence we added for a particular relationship. If we know it is the fifth sentence, we can use the command:

```
delete population 5
```

If we don't know the number of the relation, we can use a number so high that it won't be in the group of information about this relationship. In that event, Prolog deletes the last item in that portion of the knowledge base.

If we wish to remove all references to a particular relationship, we use the "kill" command. The command has two forms. The first deletes references to a relationship. The second removes all knowledge from the knowledge base; this should only be used after the knowledge base has been saved and we wish to move on to something else.

To remove all references to the "parent-of" relationship, we would write:

```
kill parent-of
```

To delete the entire genealogical knowledge base—after saving it on the disk—we type:

```
kill all
```

***A Word of Caution*** Let me point out that the particular implementation of Prolog you may be using may vary somewhat from these approaches to program modification. Consult the documentation that came with the program to be sure you are running things the way your Prolog interpreter expects.

## Queries

There are essentially two kinds of queries to be made of a knowledge base. Confirming a single fact (equivalent to answering a simple question) or retrieving unknown information that may encompass many facts. In either of these types of query, we may wish to make very simple inquiries or we may need to phrase a complex set of conditions.

The next several sections discuss how Prolog inquiries should be designed and phrased.

### Confirming a Single Fact

This is the simplest form of query. We may want to know, for example, if the population of the United States is 230 million. To find out in Prolog, we would write a sentence beginning with the key word "is":

```
is (United-States population 230)
```

Prolog would oblige us by answering "YES." If we simply want to know whether the population of China is even known to the knowledge base, we can use a variable (which we'll discuss in more detail shortly), as in:

```
is (China population x)
```

Prolog searches through its knowledge base to see if it has any relationship defined that looks like "China population" followed by any value. If it finds one, it prints the answer "YES," meaning it has access to the information, but it does *not* print the answer. To have it do so, we use a form of query designed to retrieve information, not merely confirm a fact. Before we move on to more complex and useful queries that retrieve unknown data from a knowledge base, let's consider the use of variables in Prolog.

**Variables**

Variables play two essential roles in a Prolog program. They are used to retrieve information and to display the results of queries. The two are, of course, intimately interrelated; typically we retrieve information in order to display it!

Every variable in Prolog begins with one of the six letters x, y, z, X, Y, or Z. This doesn't mean that a Prolog program can only have six variables in it (a limitation which would be totally unacceptable). A variable must *begin* with one of those letters, but it is *followed* by a unique integer. The following variables are all different from one another and legal in Prolog:

```
x1
X1
z43
y22
```

The variables x1, x01, and x000001 are all the same because the integer of each evaluates to the number 1.

**Retrieving Unknown Data**

Most often, we don't just want to confirm a fact or to confirm the existence of a fact in our knowledge base, but we want to ask the system something more complicated. The most common query will require the knowledge base to fill in the blanks in our query with information it has that fulfills the requirements posed.

To make the following discussion a bit more interesting, let's assume our world knowledge base has been expanded and now includes the following Prolog descriptive sentences.

```
United-States population 230
United-States Western
United-States democracy
United-States area 3
United-States language English
China population 824
China Asian
China dictatorship
China area 4
China language Chinese
United-Kingdom population 53
United-Kingdom Western
United-Kingdom parliamentary-democracy
United-Kingdom area .9
United-Kingdom language English
```

We have already seen how to retrieve a single fact from such a knowledge base. For example, to find out what language is spoken in China, we could write a Prolog query such as this:

```
which(x : China language x)
```

Prolog would *retrieve* the result, "Chinese," and print it as shown here:

```
Chinese
no (more) answers
```

The second line, used in most Prolog implementations, informs us that the search has been completed through the entire knowledge base and the answer(s) printed are the only ones that satisfy the criteria of the query.

Some queries obviously have more than one answer. For example, we could write the following Prolog inquiry to extract from our knowledge base the names of all countries where the principal language is English:

```
which(z : z language English)
```

Prolog's answer would be:

```
United-States
United-Kingdom
no (more) answers
```

Notice that in "which" queries the variable appears twice. The first time it is followed by a colon. This is called the "answer pattern" and it tells Prolog what to print. The second occurrence is the "query condition" itself. Sometimes, we will want to print more than one variable from a search (an idea we explored in some detail in Chapter 7) and so we will have more than one variable in the answer pattern position.

We would take a similar approach to finding out which countries in our knowledge base are Western (the computer's responses are indented in the text below):

```
which(x23 : x23 Western)
  United-States
  United-Kingdom
  no (more) answers
```

Note that the variables used must match. Using different variable names won't achieve the desired result. Thus this Prolog inquiry:

```
which(x23 : x15 Western)
```

will not print the answer to the query because we've asked Prolog to retrieve information with one variable name and print it with another.

Besides retrieving information that is an exact match with our condition (i.e., language English), we can use built-in Prolog primitives to do comparison selection from the knowledge base. For example, to see all of the countries with a population less than 250, we could write this Prolog query sentence:

```
which(y : y population LESS 250)
  United-States
  United-Kingdom
no (more) answers
```

If we make an inquiry that has no matching information in the knowledge base, Prolog's response is simply "no (more) answers," which is why "more" is in parentheses. In this case, the answer is really "no answers."

**Conjunctive Queries**

Even the ability to retrieve specific information by fill-in-the-blanks methods is limiting. We might really want to know, for example, what *Western* country has a population of less than 100. In that situation, we use a *conjunctive* query—one that combines two or more criteria into a single query using the word "and" or the symbol &. Prolog understands both to mean that the conditions joined by them must be true for the condition to be met. (The next section discusses NOT and OR conditions which can be used similarly.)

To find out which Western country or countries have populations of less than 100, we would write this query:

```
which(x : x population LESS 100 and x Western)
  United-Kingdom
  no (more) answers
```

Another variation of the conjunctive query is the double-blank query represented by the following sentence:

```
which(x : x population y)
  United-States
  China
  United-Kingdom
  no (more) answers
```

Essentially, the query means, "Tell me which countries you have population information about." If we had deleted the United Kingdom's population record with a delete command, it would not appear in the answer list above. Although we've asked Prolog to match up any population value with the variable "y," we have not asked to have that value printed. We could have done so by coding:

```
which(x y : x population y)
  United-States 230
  China 824
  United-Kingdom 53
  no (more) answers
```

You can see, from the way Prolog responded to this query, that the question posed is slightly different from the inquiry. Here, we asked Prolog, "For every country about which you have population information, print out the name of the country and the population data."

**More Complexity: NOT and OR**

We can *negate* a condition by the use of the Prolog primitive relation NOT. This is how we get around the fact that Prolog does not include a greater-than function but only a less-than function called LESS. To extract from our knowledge base the countries whose populations are greater than 200, we use this Prolog sentence:

```
which(x : x population not LESS 200)
  United-States
  China
  no (more) answers
```

If we had a larger knowledge base we might want to list all of the countries that are either Asian or Western but not list the African and European nations. We could use this approach:

```
which(x : either x Western or x Asian)
  United-States
  China
  United-Kingdom
  no (more) answers
```

As it turns out, all the countries in our very small knowledge base fit the criteria. The form of the OR construction in Prolog is:

```
either condition1 or condition2
```

Any information in the knowledge base that fits either of the conditions will be selected and reported.

You can see how nesting these types of combinations and controls—LESS, EQ, AND, and EITHER-OR—in one Prolog query can result in a very complex, sophisticated, and focused inquiry sentence.

## Arithmetic Processing in Prolog

Prolog is not a strong language for mathematics. If you design an application that involves extensive use of calculations, you'd be better off with another language. Nonetheless, there are times when you need some arithmetic. In such situations, Prolog has basic primitives that can be used.

Which primitives are available for math is implementation-dependent. Most Prologs will at least have the following primitives: SUM, INT, TIMES, and comparison operators, including LESS and EQ. The last two, LESS and EQ, are not solely arithmetic functions since they can be applied to strings as well, but they are often used to compare two numeric values, so we discuss them here for convenience.

An interesting restriction on the use of these built-in Prolog primitive relations is that you can include only one unknown in a query using these primitive commands. This is because there really isn't a knowledge base in which Prolog looks up answers and fills in blanks with a combination matching the pattern. Instead, Prolog essentially *simulates* the knowledge bases of addition and multiplication tables. The following attempt, for example, to code an inquiry to list all pairs of numbers that when multiplied together result in the answer 18:

```
which(x y : TIMES (x y 18))
```

results in the error message, "Too many variables."

## SUM

The Prolog primitive relationship SUM describes a three-argument relationship that is true only when the third argument in its list is equal to the sum of the first two arguments. In other words,

```
SUM (x y z)
```

is a true relationship only if $x + y = z$.

We can use SUM to check a value to see if it is the sum of two numbers:

```
is (SUM (25 55 80))
  YES
```

SUM can be used both for addition and subtraction, as shown in these two Prolog query sentences:

```
which (x : SUM(31.3 14.4 x))
  45.7
which (x : SUM(x 14.4 45.7))
  31.3
```

In the subtraction example, we could have placed the variable "x" in either of the first two positions in the SUM argument list, since addition doesn't care which value is in which place.

## INT

This built-in Prolog primitive relation has two functions: it can *test* a value to determine if it is an integer (i.e., a whole number with no decimal part) or it can *convert* a number from a floating-point number (i.e., a number with a decimal portion) to an integer. Let's look at some examples:

```
is (829 INT)
  YES
is (14.3 INT)
  NO
which (x : 11.98 INT x)
11
```

## TIMES

We have discussed TIMES previously. It is used to check a product and for multiplication and division. The following examples demonstrate its use:

```
is (TIMES 8 12 96))
  YES
which (x : TIMES(8 12 x))
  96
which (x : TIMES(8 x 96))
  12
which (x : TIMES(x 8 96))
  12
```

TIMES is used in conjunction with INT to determine if a specific division produces a whole-number result. These two examples demonstrate this use.

```
is(TIMES (6 X 72) and X INT)
  YES
is(TIMES (6 Y 14) and Y INT)
  NO
```

**LESS and EQ**
The two main comparison primitives in Prolog are LESS and EQ. Both may be used only with two arguments and only to determine if two values are equal or unequal and if unequal, which is the greater. The use of these primitive relations is shown in these examples.

```
is(83 LESS 43)
  NO
is(14 LESS 14)
  NO
is(11 LESS 111)
  YES
is(9 EQ 9)
  YES
```

Most of the time our checking will involve combinations of variables and calculations rather than the simple number-for-number comparisons shown here, but the syntax is the same for all such evaluations.

We discussed earlier how combining these comparison operations with AND and OR constructions can yield complex and sophisticated Prolog query sentences.

## Creating Rules with Conditional Sentences

We have said that a Prolog knowledge base consists of a collection of relationship descriptions (facts). We also alluded to the existence of rules. Now we'll look at what rules are and how they relate to the knowledge base.

A rule is more precisely referred to in Prolog as an "implication." I use the word "rule" because it communicates more clearly what we're talking about.

**Simple Rules**
Essentially, a rule defines a relationship by means of a conditional sentence. Here's an example, stated in English, of the type of rule we are considering: "A man is tall and slender if he has a height of more than six feet and a weight of less than 200 pounds." In Prolog, this rule sentence might be coded in this way:

```
(x tall-and-slender if x height not LESS 6 and x weight LESS 200)
```

This defines a new relationship called "tall-and-slender" and we can use this rule relation in querying the data base. The following query is legal if we have made the rule sentence part of our knowledge base.

```
which(x : x tall-and-slender)
```

In response to this query, Prolog scans the knowledge base and examines each data item to see if its height is 6 or greater *and* its weight was less than 200. For each "hit" in this process, it prints the person's name on the terminal display.

Returning to our world population knowledge base, we could write a new rule:

```
(x free if not x dictatorship)
```

Now we can write this query sentence:

```
which(x : x free)
  United-States
  United-Kingdom
```

We may, of course, combine these rule statements with other query sentence constructions:

```
which(x : x free and x population not LESS 200)
  United-States
  no (more) answers
```

This would return a list of all countries that are not dictatorships and that have a population of 200 or greater; in this case, the only match with the query pattern is United-States.

**Recursive Rules**

One of the most powerful ideas in programming is that of "recursion"—programs or procedures that call on themselves for further execution. In Prolog, recursive *rule* definition adds a dimension of power and flexibility to knowledge base inquiries.

To understand this concept, we'll look at an example involving a family tree. This example is used often in Prolog texts and by instructors because it is a rich model and one in which the ideas of relationship are understood. If our family knowledge base contains a relationship called "parents-of" that defines the mother and father of a member of the tree, we need recursion to effectively and efficiently define a new relation called "ancestor-of."

To define what "ancestor-of" means in English, we might tell someone, "Your ancestors are your parents and all the ancestors of your parents." This is a recursive definition using the term "ancestors" to define the term "ancestors." (If you read this in a dictionary, you'd be justifiably angry. It's perfectly acceptable in a computer program, though—and even necessary sometimes.) Let's look at how Prolog could handle this assignment.

***Stating the Relationship*** In Prolog sentence terms, an ancestor-of relation is defined by a combination of two relationships: one's parents and one's parents' ancestors. We can write this relationship in Prolog as follows:

```
x ancestor-of y if x parent-of y
x ancestor-of y if z1 parent-of y and x ancestor-of z1
```

Now if we inquire of our family knowledge base:

```
which(x ancestor-of Danielle)
```

the program will examine the "ancestor-of" property and determine that it requires either that we be talking about one's parents or about people who qualify as ancestors of the parents. It looks up the definition of "parents-of," substituting that in the query we have made, and so on. Depending on how the knowledge base was initially defined, this query might ultimately be integrated by Prolog into this statement:

```
(which x : (x either (either mother-of y or father-of y) or ancestor-of
(either mother-of y or father-of-y))
```

Since this is a recursive rule, we never eliminate the "ancestor-of" term as we would in solving, for example, a mathematical equation where that was an unknown term.

## List-Processing Fundamentals

The list is a key data structure in Prolog. As with LISP and Logo, Prolog is often called upon to manipulate the contents of lists in AI applications.

### Using Lists in Sentences

We use lists when we have a fact that refers to a group of individual data items. For example, if Carolyn, Sheila, and Mary enjoy music, we might define the relationships as follows:

```
music enjoyed-by Carolyn
music enjoyed-by Sheila
music enjoyed-by Mary
```

(We could have written these the other way around, "Carolyn enjoys music," but the first approach will be more natural in the next example.)

Now if we ask the knowledge base via a Prolog query sentence:

```
which(x : music enjoyed-by x)
  Carolyn
  Sheila
  Mary
  no (more) answers
```

we can expect the program to find all three people in its knowledge base who enjoy music. But we could also code the original relationship as follows:

```
music enjoyed-by (Carolyn Sheila Mary)
```

In this case, however, our query produces an entirely different result:

```
which(x : music enjoyed-by x)
  (Carolyn Sheila Mary)
```

The use of a list here has the advantage of producing, in one simple answer, all the people in the knowledge base who have the relationship "enjoyed-by" with the individual called "music." *But*, the disadvantage of this approach is that if we ask the system:

```
is(music enjoyed-by Carolyn)
```

the answer will be NO because Carolyn does not match any conditional clause that attaches the enjoyed-by relationship with music.

**Extracting Information from Lists**

Obviously, we need some way of getting one individual extracted from a list. The technique for doing this depends on whether we know the length of the list in advance.

***Known-Length Lists*** For the next few examples, we will assume we have a family tree knowledge base built with lists of parents and children.

```
(Dan Carolyn) parents-of (Sheila Mary Christy Heather)
(Don Rae) parents-of (Matt Dawn Adam)
(John Cindy) parents-of (Becky Shannon)
(Corky Penny) parents-of ()
(Wally Julie) parents-of (Candace)
(Donald Daisy) parents-of (Huey Dewey Louie)
```

This list consists of a fixed-length list two elements long on the left, where the parents are stored; a description of the parents-of relationship; and a variable-length list on the right which lists all of the couple's children.

Notice that we have two situations in the right-hand list that don't seem to call for a list. Corky and Penny are childless but are described as parents of an empty list (which is how () is translated). Wally and Julie have only one child, Candace, but she is placed in a list of her own. We do this in Prolog because all sentences about one relationship must have a uniform pattern. This enables us to use predictable pattern-matching techniques to retrieve information.

Suppose we now want to retrieve the children of Don. Children of Don will be stored in a sentence with the pattern:

```
(Don y) parents-of x
```

where the length of x is unknown and, for this purpose, not important. We write the following Prolog query sentence:

```
which(x : (Don y) parents-of x)
  (Matt Dawn Adam)
  no (more) answers
```

Since we know the length of the list of parents is two, we can extract information about any one of them by the expedient of substituting a variable for the other. For example, if we want to know Don's wife's name, we inquire:

```
which(y (Don y) parents-of x)
  Rae
  no (more) answers
```

***Lists of Unknown Length*** Our first approach works for obtaining information about parents but does not permit us to find out, in a single Prolog query sentence, whether or not someone is a particular child's mother. Nor can we find out if a child is the offspring of one or two parental names. If all the families had two children, we could define two new rules like this:

```
y mother-of x1 if (x y) parents-of (x1 x2)
y mother-of x2 if (x y) parents-of (x1 x2)
```

But we need similar rules for one-child families as well as for larger families. This process is more complex than that for getting information from fixed- or known-length lists, but its implementation is important to being able to program in Prolog efficiently.

To begin with, Prolog adopts a new syntax to describe a list. It defines all lists as consisting of a *head* followed by a *tail*. Thus the list (Sheila Mary Christine Heather) has as its head its first element, "Sheila," and as its tail the list "Mary Christine Heather". This head-tail schematic is represented in Prolog by the symbol:

```
(x|y)
```

In this syntax, the vertical bar symbol ("|") is read "followed by." So (x|y) is the list comprising the element x followed by the list y.

Using this new list description syntax, Prolog defines a new primitive relation, "belongs-to." It does so by means of a pair of recursive rules:

```
X belongs-to (X|Z)
X belongs-to (Y|Z) if X belongs-to Z
```

With, this definition we cause Prolog to look at each element in a list Z to see if X is equal to that element. If it is, Prolog declares that X does have the belongs-to property with respect to the list Z. If the first element (i.e., the head) is not the element X, then the second step of the recursive rule takes over and looks at the rest of the list, *dropping the old head from consideration and making the first element in the tail the new head.* This is the key to the idea of getting information from variable-length lists. An example may clarify the situation.

If we want to know if Christine is a child of Dan and Carolyn, we would use the new belongs-to syntax as follows:

```
is((Dan Carolyn) parents-of Z and Christy belongs-to Z)
  YES
```

In other words, we first define the list Z as containing the list of children of Dan and Carolyn and then ask if the specific child belongs to the list. The answer, here, is YES.

Incidentally, the notion of a list as a head followed by a tail has many other uses in Prolog; it is an important concept in any meaningful AI application of the language.

**The Lengths to Which Lists Will Go**
There are two other commonly built-in Prolog routines dealing with lists that we should mention here. One determines the length of a list and the other finds a list or lists of a given length in a knowledge base.

***A List Has-Length*** The "has-length" primitive relation is used to find or check the length of a list. If we have a list (Matt Dawn Adam) whose length we wish to check, we use the following processing step:

```
which(x : (Matt Dawn Adam) has-length x)
  3
  no (more) answers
```

If, on the other hand, we want to confirm that a given list has a length of four, we might code:

```
is((Matt Dawn Adam) has-length 3)
  YES
```

***Find a List with Length-Of. . .*** To find a list of a given length in a knowledge base, we could use this query sentence:

```
which(x : 4 length-of x)
  (Sheila Mary Christine Heather)
  no (more) answers
```

Notice that we *don't* need to include the length of the list explicitly in the data base as we do geographic information in our one knowledge base or family data in the other. Prolog "knows" about the length-of attribute of a list; all we do is access it.

## Input/Output Primitives

In this section we'll explore the primitives for displaying information on a computer screen, for asking for and receiving data from the keyboard, and for saving and retrieving Prolog programs.

**Printing on a Screen: P and PP**
Prolog statements that result in information being displayed, retrieved, and stored are among the very few *imperatives* in this descriptive language.

To print a message on a display screen in Prolog, use either the "P" or "PP" primitive. The difference between them is that P prints the message and leaves the cursor at the end of the line of data, while PP causes a carriage return and line feed at the end of the line of output, placing the cursor at the first position on the next line.

Here are two examples (@ represents the cursor).

```
P (Dan Carolyn parents-of Mary)
  Dan Carolyn parents-of Mary@
PP (Don Rae parents-of Adam)
  Don Rae parents-of Adam
  @
```

Note that what we want printed is enclosed in parentheses, but that the parentheses do not print.

**Querying the User with "is-told"**
There are times when the information to be queried in the knowledge base is something the *user* will furnish, rather than being known when the program is written. In this case, the "is-told" primitive can be used to ask questions of the user. Its general form is:

```
(question) is-told
```

For example, to enable the system to determine merchandise prices marked down by ⅓, we could write a small Prolog routine such as this:

```
which(Price at Y : (Current price x1) is-told and which Y(TIMES 0.3333 X1 Y))
```

This query sets up a dialog between Prolog and the user in which Prolog asks "Current price x1?" and prints "Price at. . .," supplying the figure that is ⅓ of the price. Because of the rules of the language, users must precede a response either with the symbol "ans" (which means "Here's one answer; come back when you're ready for another input") or "just" (which means "This is my last (or only) answer. Quit when you've done this one.")
Here is how such a dialog might look on the terminal:

```
Current price x1? ans 21
Price at 7
Current price x1? ans 41
Price at 13.666667
Current price x1? just 15.15
Price at 5.05
no (more) answers
```

The is-told primitive is also used to get yes and no answers from the user in response to questions. The format is identical to that in our example.

**The R (for "Read") Primitive**
The "R" primitive permits us to read information from the keyboard. It differs from the is-told primitive in some important respects. First, R will accept any number of inputs as a response until the user hits the [RETURN] key. Second, the user need not precede the answer with ans or just. Finally, responses using the R primitive need not have a question associated with them (i.e., they may be unprompted).

The following rule defines a new function that asks for a list of numbers and then furnishes the sum of those numbers. It uses another description called "list-sum," which is not shown here but is assumed to exist elsewhere in the program.

```
x sum-of-entered-list if P(Enter a list of numbers) and R(y) and x list-sum y
```

If we now query this relation by coding:

```
which (x : x sum-of-entered list)
```

the following dialog will occur:

```
which (x : x sum-of-entered list)
  Enter a list of numbers (8 12 411 3)
  434
  no (more) answers
```

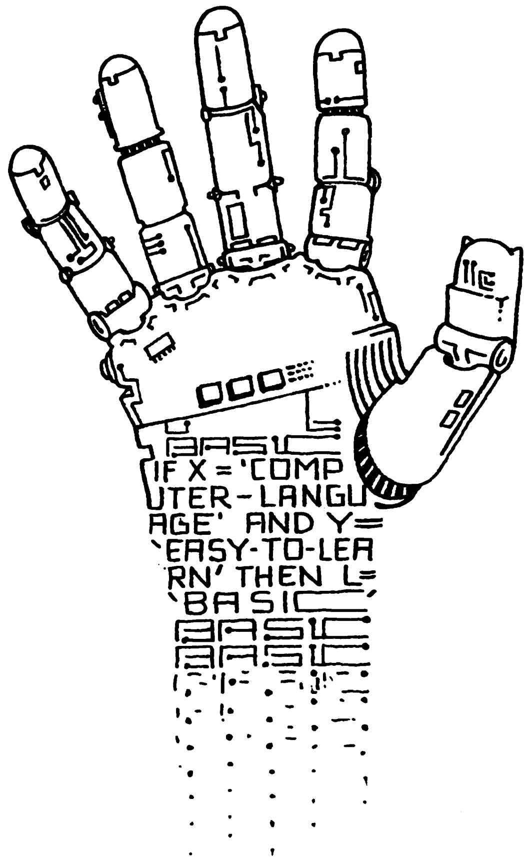
## Concluding Remarks

Prolog is one of the most flexible languages available. This chapter has provided the basic elements of Prolog, a language capable of extension by the programmer who wishes to define new functions and operators. To learn more about this language, I strongly recommend Clark and McCabe's book, referenced in Appendix C.

CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI

13

# For BASIC Programmers



- Differences Between BASIC and Other Languages
- Specific Things to Think About in Creating AI in BASIC
- Property Lists and Their Simulation in BASIC

If you are a BASIC programmer who would like to implement some of the programs in this book in that language, you may wonder how difficult such a translation will be. In this chapter, we'll explore some of the major tasks involved in the translation. If you want to undertake such a conversion, you should know something about both BASIC and Logo. The fact that you are even *reading* this chapter indicates you have at least an acquaintance with BASIC. Reading Chapter 10 will give you enough information about Logo to get by this chapter. With that background, you're ready to see what's involved in the translation of these programs.

## Fundamental Differences

In addition to the specific, detailed differences discussed in the remainder of this chapter, there are a few fundamental, stylistic differences between BASIC and Logo programs.

These comments are more applicable to versions of BASIC available on most other microcomputers prior to the introduction of Microsoft's powerful and structured MSBASIC Version 2.00 for the Mac in early 1985. From an AI programming standpoint, the Microsoft® program overcomes many of the shortcomings of earlier implementations of the language. Apart from the cumbersome way even this version of BASIC deals with lists, meaningful AI programs could be written in MSBASIC Version 2.00 or later.

### Structured vs. Unstructured Programs

BASIC programs have tended to be unstructured, primarily because of BASIC's use of line numbers rather than procedure names to transfer control. Such a design encourages programs that are collections of sequential commands with GOTO and GOSUB branching implemented to deviate from the otherwise straight-line execution of code.

With good commenting (almost never done by programmers in any language!), an unstructured program is not necessarily any *worse* than a structured one. But it has the distinct disadvantage—in the world of commercial programming—of being difficult for a team of programmers to work on. Structured languages, on the other hand, facilitate and even encourage piecemeal, teamwork approaches to system design and programming.

### Recursion Impossible

There are frequent needs in AI applications to implement recursive procedures—procedures that repeatedly invoke themselves with different arguments. (The classic example of recursion is a program that calculates the factorial of any number.)

Implementing such a recursive procedure typically requires two things lacking in most BASICs: local variables and separate procedures. Again, MSBASIC 2.00 corrects these deficiencies, so this discussion does not apply if you are using it. But other microcomputer versions of BASIC typically ignore these issues.

### Intuitive Program Understanding

A final essential difference between BASIC and Logo is that Logo uses names where BASIC uses line numbers. When we see a Logo statement that calls into effect a procedure, named for example, SETUP_MAZE, we know intuitively what that procedure will do (provided the programmer has not used misleading names for procedures, of course). But in a BASIC program, the command GOSUB 20000 doesn't tell us anything at all about what the subroutine at line 20000 does. We have to look at that line of code and, if the programmer has commented the code liberally, we may get an understanding of the program's flow and operation.

In a Logo program, then, we can look at the main driver routine and those procedures it directly invokes, and we can figure out a great deal about how the program works. That's why we used box diagrams throughout the first part of this book to depict the operation of our programs.

## Specific Implementation Concerns

So much for generalities. Now let's look at some of the very specific implementation problems that arise when we try to write AI-type programs in BASIC.

### List Manipulation

Perhaps the most important problem in converting a program written in Logo or LISP (or some other list-rich language) is the absence of the list as a data structure in BASIC. No BASIC we know about for a microcomputer—including the otherwise quite powerful MSBASIC 2.00—implements lists. As you have seen, if you have worked through any of the programs in Part I, lists are critically important data structures in AI programs. That is why Logo, LISP, and other languages with strong list-handling capabilities tend to be selected for the creation of AI systems.

Even though cumbersome, it is possible to implement listlike structures in BASIC. We'll describe how a list can be simulated in BASIC and demonstrate how some of the key list-manipulation routines can be handled.

***A List Is an Array. . . Sort Of*** From the standpoint of a BASIC program, a list is a 1-dimensional array. To define a list that will contain five elements, we would set up an array in BASIC using a command such as:

```
DIM LIST1$[5]
```

(Throughout this chapter, we'll use generic BASIC commands wherever we can. Where an MSBASIC 2.00 instruction is needed, we'll point out its use.)

Two important limits about the way BASIC handles lists are apparent. First, the program must be told *in advance* what kind of data the list will contain; mixing data types, which can be done in Logo and LISP lists with no problem, is taboo in BASIC. Our DIM statement defines this list to contain only strings. (Since BASIC offers a facility for converting from strings to numeric values and back, that drawback may, with difficulty, be overcome.) Second, we must predetermine the maximum number of elements the list will have; the example specifies 5. Thus an array is a *static* data structure. Lists are dynamic structures in Logo and LISP, to which we can add data as necessary.

***Creating Lists*** Creating a pseudo-list in BASIC requires two steps: dimensioning the 1-dimension array (as just described) and initializing data in the array. This latter is generally handled with a loop something like the one in this very short program segment:

```
10 DIM LIST1$[5]
20 FOR N = 1 TO 5
30 READ VALUE$
40 LIST1$[N] = VALUE$
50 VALUE$ = ""
60 NEXT N
70 DATA THIS,IS,A,DIFFICULT,TEST
80 END
```

This program sets up the array called LIST1$ to have five elements and then puts THIS, IS, A, DIFFICULT, TEST into the array. (Some versions of BASIC call the first element in an array element zero instead of element one, resulting in another level of confusion for the programmer.)

***Simulating FIRST and BUTFIRST*** In Logo and LISP, when dealing with lists, we often want to do something to the first element of the list or, alternatively, to all of the elements except the first. Because of the frequent need for such manipulation, those languages have specific string manipulators for these extractions. In Logo, they are FIRST and BUTFIRST (abbreviated BF), respectively. Invoking those functions in BASIC is more complex than in Logo or LISP.

FIRST can be implemented quite easily. We define a function or use a subroutine or a subprogram (in MSBASIC 2.00 or later) to return the first element of an array.

```
FIRST$ = LIST1$[1]
```

If a program will frequently need this simulation, you pass the name of the list to the procedure or function rather than code it explicitly into the definition of the operation.

BUTFIRST is more complex to implement. It requires a loop similar to that used to initialize the array in the first place. Assuming we want to assign the BUTFIRST of the list called LIST1$ to a new variable called BFLIST1$, the following code could be used (assuming the list is set up):

```
100 FOR N = 1 TO 4
110 BFLIST1$[N] = LIST1$[N + 1]
120 NEXT N
```

How to know that the original list had five elements and, therefore, the new BF of that list will contain four, is problematic. In Logo, if we had to carry out such a function, we simply use the command COUNT to find out how many elements were in the list. No such command exists in BASIC. We will probably want to store explicitly in some variable, say LENLIST1, the number of elements in LIST1$. Then the earlier code could be generalized as follows:

```
100 FOR N = 1 TO LENLIST1 – 1
110 BFLIST1$[N] = LIST1$[N + 1]
120 NEXT N
```

***Adding Elements to a List*** Another frequent occurrence in Logo programs is the need to add an element to a list, usually with some concern about adding it to the front or the end of the list. This is a tricky and cumbersome process in BASIC. Assuming we haven't dimensioned the array too small initially to handle the additional element, adding one to the end of the list is straightforward. We use a variable to keep track of the last position in the list and then assign the new item to the end of the list.

Assume, for the sake of example, that we want to add the word "indeed" to the end of LIST1$ and have dimensioned that array to be six elements long rather than five. Further assume that the last position used in the array—the fifth—stored in the variable called LASTUSED. Then the following routine would add INDEED to the end of the list called LIST1$:

```
100 LIST1$[LASTUSED + 1] = "INDEED"
110 LASTUSED = LASTUSED + 1
```

This second line will only be necessary if we expect to add more than one element to the end of a list.

Adding an element at the *front* of a list is really tricky in BASIC. Rather than reproduce the code, we'll explain the *steps* required and leave it to you to figure out how to implement that in BASIC code.

1. Begin with two arrays, one dimensioned at least one element larger than the other. We'll call them LIST1$ and LIST2$, with the latter being the larger of the two.
2. To add an element to the front of LIST1$, put that element into the first position of LIST2$.
3. Next, copy the elements of LIST1$ into LIST2$, with their position in LIST2$ incremented by one over their position in LIST1$ (LIST1$[1] becomes LIST2$[2], etc.).
4. Now, if you want the new list to be called LIST1$, you'll have to write the entire new list back into the variable LIST1$. If your BASIC permits you to release arrays when they are no longer needed, remove LIST1$ from the arrays the program knows about and rename LIST2$ as LIST1$. (If that's not possible, using lists is going to get even messier!)

***REVERSE a List*** Though we haven't used REVERSE operations on lists in this book, AI programs often need such a facility, particularly in natural language processing. Logo includes a command to REVERSE a list; BASIC does not. To simulate the REVERSE function in BASIC, you could try such a routine as this:

```
100 FOR N = 1 TO LENLIST1
110 LIST2$[N] = LIST1$[LENLIST1 + 1 - N]
120 NEXT N
```

This routine takes advantage of the fact that when we reverse a list, the numbers of the positions of the corresponding elements add up to the length of the list plus 1. In other words, if we are working with a five-element list, the value in position one of the new list corresponds to the value in position five of the old list and 5 + 1 = 6. The second and fourth positions correspond (2 + 4 = 6) and so on.

## Property Lists

Another major data structure in Logo not available in BASIC is the *property list.* These lists are composed of a series of elements associated in pairs. The first item in each pair names the attribute involved and the second the value of that attribute for this particular list. Thus if we have a collection of lists about birds and one is called ROBIN, we might have an attribute called COLOR_OF_BREAST and, for ROBIN, the value associated with that attribute would be RED.

Implementing property lists in Logo is very clean; to know the color of the robin, we use a GPROP (Get PROPerty) command. In BASIC, things are far stickier.

### Property Lists as Two-Dimensional Arrays

A property list, from BASIC's point of view, would probably be implemented as an array of two dimensions. Each row of the table corresponds to a pair in the property list, with the first column representing the attribute and the second the value. (See Figure 13-1.)

| *Row* | *Column 0* | *Column 1* |
|---|---|---|
| 0 | attribute-name 0 | value 0 |
| 1 | attribute-name 1 | value 1 |
| 2 | attribute-name 2 | value 2 |
| 3 | attribute-name 3 | value 3 |
| 4 | attribute-name 4 | value 4 |
| . | . | |
| . | . | |
| . | . | |
| n | attribute-name n | value n |

**Figure 13-1. Property list structure in BASIC**

Naming the array to correspond with the object being described (e.g., a list called ROBIN) facilitates retrieval of information. But, this also makes it difficult to keep a list of property lists all under one name, like BIRDS, which is a trivial task for Logo. This entails a trade-off for the BASIC programmer using a two-dimensional array to replace the more flexible property list. If we are keeping track of birds, we have a choice of one large array named BIRDS, with the first entry in each row of the array being the name of the particular bird (see Figure 13-2) or a collection of smaller individual arrays, one for each type of bird (as in Figure 13-3). Neither approach permits the dynamic sizing of the array, but that is simply a BASIC limitation over which we have no control.

An additional complication occurs because an array in BASIC is indexed by numeric position while a property list in Logo is accessed by the attribute name. We have two choices in BASIC: we can keep track as we program that, for example, the second entry of each property array corresponds to the value for the color, *or* we can use the process of looping through the array to find the cor-

*ARRAY BIRDS (4 × 4)*

| **Row** | ***(name)* Column 0** | ***(size)* Column 1** | ***(color)* Column 2** | ***(domesticated)* Column 3** |
|---|---|---|---|---|
| 0 | ROBIN | MEDIUM | RED | NO |
| 1 | SPARROW | SMALL | BROWN | NO |
| 2 | PARAKEET | VERY-SMALL | YELLOW | YES |
| 3 | HAWK | LARGE | BROWN | NO |

**Figure 13-2. Single Large Array Approach to Property Lists**

*ARRAY ROBIN (2 × 3)*

| | |
|---|---|
| SIZE | MEDIUM |
| COLOR | RED |
| DOMESTICATED | NO |

*ARRAY PARAKEET (2 × 3)*

| | |
|---|---|
| SIZE | VERY-SMALL |
| COLOR | YELLOW |
| DOMESTICATED | YES |

*ARRAY SPARROW (2 × 3)*

| | |
|---|---|
| SIZE | SMALL |
| COLOR | BROWN |
| DOMESTICATED | NO |

*ARRAY HAWK (2 × 3)*

| | |
|---|---|
| SIZE | LARGE |
| COLOR | BROWN |
| DOMESTICATION | NO |

**Figure 13-3. Collection of Smaller Arrays as Property Lists**

responding pair. If we choose the first method, we could design the property list as a one-dimensional array (i.e., something that more closely resembles a list) as in Figure 13-4. We would then keep track of the position of each kind of attribute separately from the array, designing the program so that if we want to retrieve the color of a bird, we simply automatically index the second position of the array. Let's look at an example.

*ARRAY ROBIN (1 × 3)*

| | | |
|---|---|---|
| SMALL | RED | NO |

**Figure 13-4. A one-dimensional array as a property list**

**Setting Up the Property Array in BASIC**

Let's assume we want to keep track of a little information about birds. Specifically, we want to record their size (in small-medium-large terms), the color of their breasts, and whether they are domesticated. An array for the robin (called ROBIN$), will contain the following pairs of data: SIZE-MEDIUM, BREAST.COLOR-RED, DOMESTIC-NO. The same entry for a parakeet will look like this: SIZE-VERY-SMALL, BREAST.COLOR-YELLOW, DOMESTIC-YES. These arrays will look like those in Figure 13-3.

The following routine would set up these two property arrays in generic BASIC.

```
10  DIM ROBIN$[2,3],PARAKEET$[2,3]
20  ROBIN$[1,1]="SIZE"
30  PARAKEET$[1,1]="SIZE"
40  ROBIN$[1,2]="BREAST.COLOR"
50  PARAKEET$[1,2]="BREAST.COLOR"
60  ROBIN$[1,3]="DOMESTIC"
70  PARAKEET$[1,3]="DOMESTIC"
80  ROBIN$[2,1]="MEDIUM"
90  PARAKEET$[2,1]="SMALL"
100 ROBIN$[2,2]="RED"
110 PARAKEET$[2,2]="YELLOW"
120 ROBIN$[2,3]="NO"
130 PARAKEET$[2,3]="YES"$
```

There are many other ways to approach this routine, some of which may be more efficient, but our example is useful for clarity.

**Querying the Property Array**
Now let's assume that the user wants to retrieve the breast color of the robin. One way of approaching that problem is as follows:

```
100 FOR N=1 TO LENLIST1
110 IF ROBIN$[1,N]="COLOR" THEN 130
120 NEXT N
130 ANS$=ROBIN$[2,N]
140 PRINT "The robin's breast is ", ANS$;"."
```

This approach searches through the first column of a list, where the names of the attributes are stored, and finds the corresponding list entry for the value of that attribute. The alternative—keeping two arrays, one that contains the names of the attributes and the other containing the values for the array—may *seem* more efficient in terms of memory usage, since the words "SIZE" and "BREAST.COLOR," for example, would only be stored once. But we would then have to search through *two* arrays, one to find the location of the index that points to the value we want to recover. This would further slow down our already slow BASIC program.

## Other Considerations

There are many other considerations involved in converting from LIST and Logo programs to their BASIC equivalents. Most of these should be apparent if you know both languages and study a listing in one of them.

# *Appendixes*

CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL IN
L INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
L INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
L INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI

A

# A Sample LISP Listing: PiL

This appendix presents a listing of the Prologo interpreter of Chapters 7 and 8 written in ExperLISP for the Macintosh®. The program is popularly referred to in the computer press as "PiL," an acronym of sorts meaning Prolog in LISP.

## Why This Listing Is Included

I have included this LISP listing in a book on Logo because I thought that it would be helpful for the reader who wants to see what a real LISP program looks like. It will also be helpful to you to use this listing after you have looked at Chapter 11, which is an introduction to this strange-looking but widely used language.

PiL serves the secondary purpose of presenting information about Prolog in a different form for the person who is really interested in the nonprocedural languages of which Prolog is the best current example.

## Where PiL Originated

PiL grew out of a paper entitled "The World's Shortest Prolog Interpreter?" written by Professor M. Nilsson of Uppsala University. This paper appeared, among other places, in J. A. Campbell's compilation of highly technical material on Prolog entitled *Implementations of Prolog*; a full bibliographic citation appearing in Appendix C.

Although it was upon this program that the ExperLogo® interpreter in this book was based, the similarities between the Logo and LISP versions are not as great as one might hope or expect. For one thing, PiL implements the widely debated Prolog CUT function that permits some facility in recursion but results in less "Prologlike" code in the minds of some purists. (A discussion of this function is beyond the scope of this book; see Appendix C's recommended references for more study.)

### Executing PiL

The interpreter in PiL is invoked by coding:

```
(PROVE (goal1)(goal2)...(goal N))
```

Goals are stated the same way as in Prologo except that the question mark is used in place of the underscore to indicate the variables whose values are being sought. Thus we might write this line of LISP code:

```
(PROVE (GRANDFATHER-OF ?X ?Y))
```

to get a list of all the grandfathers in the knowledge base.

A knowledge base in LISP is set up with the SETQ primitive rather than the MAKE command of Logo. Other than that, the establishment of a knowledge base is quite similar between the two implementations.

### Nonstandard LISP Syntax

There is only one nonstandard LISP syntax in the PiL program and it was originally introduced by Nilsson. The use of the LOOP construct is not generally permitted in LISP and, in fact, most LISP interpreters don't allow for it at all. But it is a useful construction, particularly in situations like those in ExperLisp where an iteration is required. Most LISPs provide for some form of iteration.

If you are working with XLISP (a public-domain LISP available from a number of sources for the Macintosh®) or a non-Macintosh® LISP, you may find it necessary to modify PiL to accommodate your interpreter's definition of an interactive process. You can use a PROG or a DO construction if you're using a version of MACLISP.

Essentially, the LOOP construction can be understood as follows: Set the local variables *var1* and *var2* to *expr1* and *expr2*, respectively. Then for all elements in (list) set (var3) to the head of the element and (var4) to the tail and evaluate (expr3). Continue to evaluate (expr3) until the list is exhausted, and then return "nil" unless RETURN gets called with an argument during the evaluation, in which case that value should be returned immediately and the iteration terminated.

```
(defun prolog (database)
  (print "Welcome to Prolog")
  (prolog-1 database)
  (print "Back to Lisp")
  t
)
(defun prolog-1 (database) ;;a top-level loop for Prolog
  ;;reads a form, proves it, and then iterates
  (prove (list (rename-variables (read) '(0)))
    '((bottom-of-environment))database 1)
  (print "no (more) answers")
  (if (y-or-n-p "Another assertion? (y or n)")
    (prolog-1 database)))

(defun prove (list-of-goals environment database level)
  ;;proves the conjunction of the list-of-goals
  ;;in the current environment
  (cond
    ((null list-of-goals)
      ;;succeeded since there are no goals
      (print-bindings environment environment)
      ;;ask user if another possibility is wanted
      (not (y-or-n-p "More? (y or n)")))
    (t (try-each database database
         (rest list-of-goals)
         (first list-of-goals)
         environment level))))

(defun try-each (database-left database goals-left goal environment level)
  (cond
    ((null database-left)
      ()) ;;fail if nothing left in database
    (t (let ((assertion
           (rename-variables (first database-left)
             (list level))))
      (let ((new-environment (unify goal (first assertion) environment)))
    (cond
      ((null new-environment) ;;failed to unify
        (try-each (rest database-left) database
          goals-left goal environment level))
      ((prove (append (rest assertion) goals-left)
        new-environment database (add1 level)))
      (t (try-each (rest database-left) database
          goals-left goal environment level)))))))

(defun unify (x y environment)
  (let ((x (value x environment))
    (y (value y environment)))
  (cond
    ((variable-p x) (cons (list x y) environment))
    ((variable-p y) (cons (list y x) environment))
    ((or (atom x) (atom y)) (and (equal x y) environment))
    (t (let ((new-environment (unify (first x) (first y) environment)))
      (and new-environment (unify (rest x) (rest y) new-environment)))))))
(defun value (x environment)
  (cond
    ((variable-p x)
      (let ((binding (assoc x environment)))
        (cond
          ((null binding) x)
          (t (value (second binding) environment)))))
    (t x)))

(defun variable-p (x)      ;;a variable is a list beginning with "?"
(and (listp x) (eq (first x) '?)))

(defun rename-variables (term level)
  (cond
    ((variable-p term)
      (append term level))
    ((atom term) term)
    (t (cons (rename-variables (first term) level)
        (rename-variables (rest term) level)))))

(defun print-bindings (environment-left environment)
  (cond
    ((rest environment-left)
      (let ((variable (first (first environment-left))))
      (cond
        ((zerop (third variable))  ;;variable level
          (princ (second variable))  ;;variable name
```

```
            (princ " = ")
            (print (value variable environment)))))
      (print-bindings (rest environment-left) environment))))

(defun y-or-n-p (message)
   (print message)
   (let ((response (read)))
   (cond ((eq response 'y) t)
      ((eq response 'n) nil)
      (t (y-or-n-p message)))))

(setq rest cdr)

(defun expand-assertion (assertion environment)
   (cond
      ((atom assertion) assertion)
      ((variable-p assertion)
         (let ((val (value assertion environment)))
            (if (eq val assertion)
                  val
                  (expand-assertion val environment))))
      (t (cons  (expand-assertion (car assertion) environment)
                (expand-assertion (cdr assertion environment)))))
```

CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELL
ELLIGENCE ARTIFICIAL IN
L INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELL
ELLIGENCE ARTIFICIAL IN
L INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELL
ELLIGENCE ARTIFICIAL IN
L INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELL

# B

# Converting Between Mac Logos

Two important versions of Logo available for the Macintosh®: ExperLogo® by ExperTelligence (the one we chose to use for our programs) and Microsoft's LCSI Logo®. This appendix discusses significant differences between these two published versions of Logo and points out what to watch for in future implementations. For people who don't have ExperLogo®, this appendix should facilitate translating the programs in this book to another form of Logo.

Throughout the text, where differences exist between ExperLogo® and LCSI Logo®, we have noted them with comments contained in parentheses; those differences are not duplicated here.

## Speed of Execution

One of the biggest differences between ExperLogo® and any other Logo we've seen for any system is its speed of execution. Because ExperLogo® is compiled rather than interpreted, it can run programs and procedures much faster than one would expect from this traditionally slower language.

We can't do anything about that. But if you find yourself with a traditional interpreted Logo, and some of the programs—particularly such longer ones as Blocks World and Prologo—run unacceptably slowly, you'll understand the reason.

## Array-Handling

The ability to manipulate arrays is available on both of the currently published versions of Logo. However, such a capability has not been part of Logo traditionally, and it is conceivable that Logos developed for the Mac in the future may not include them.

This would not be a serious drawback, however, since the primary advantage of arrays over the more flexible and popular list structure is one of processing speed. Because an array has known dimensions, and is accessed by numerical position rather than by list-manipulating instructions, extracting data from an array is quicker than drawing information from a list. Other than that, substituting a list for an array has no effect on the program.

There is another array distinction between ExperLogo® and Microsoft Logo®. To create an array in ExperLogo®, the MAKE_ARRAY function is combined with the well-known MAKE function, as shown here:

```
MAKE POSITION MAKE_ARRAY [4 5]
```

(This line is from Micro Blocks World in Chapter 5.)

In Microsoft Logo®, the equivalent of MAKE_ARRAY is ARRAY, so the above example would be coded as follows:

```
MAKE POSITION ARRAY [4 5]
```

## Extracting Multiple Elements from a List

ExperLogo® implements a very powerful ELEMS command that extracts several consecutive items from a list. It takes three arguments. The first provides the *starting* element, the second defines the *number of elements* to extract, and the third is the name of the string from which the elements are to be extracted. The following examples (with computer output indented for ease of reading) clarify this command.

```
MAKE TESTLIST [A B C D E F G H I J K]
  [A B C D E F G H I J K]
ELEMS 2 5 :TESTLIST
  [B C D E F]
ELEMS 6 1 :TESTLIST
  [F]
ELEMS 1 9 :TESTLIST
  [A B C D E F G H I]
```

There is no direct equivalent of the ELEMS function in LCSI Logo (or in most other Logos, for that matter). The same result can be achieved, however, by defining a procedure called ELEMS that takes the same information as arguments. There are a number of ways of implementing such a procedure. Here is one:

```
TO ELEMS :START :LENGTH :OBJECT
  IF :START = 1[ELEMS_2 :LENGTH :OBJECT]
  [ELEMS :START - 1 :LENGTH BF :OBJECT]
END

TO ELEMS_2 :LENGTH :OBJECT
  IF :LENGTH = 1[ITEM 1 :OBJECT]
  [SE ITEM 1 :OBJECT ELEMS_2 :LENGTH - 1 BF :OBJECT]
END
```

(This solution is adapted from the ExperLogo® user manual, page 164.)

## Local Variable Declaration

LCSI Logo permits a list of names to be given to the LOCAL command, which can compress the code where such variables are defined. For example, in Prologo's TRY_EACH procedure, there are two LOCAL variables: ASSERTION and NEW_ENVIRONMENT. In ExperLogo®, declaring these requires two statements; in LCSI Logo®, one statement suffices.

## TEST, IFTRUE and IFFALSE

Microsoft's Logo® has no equivalent of the ExperLogo® TEST function. (For an example of its use, see the final procedure, FIND_PROBABLE, in the program in Chapter 6.) The alternative implementation is more cumbersome, but only slightly so. Substitute an IF construct as shown here.

In ExperLogo®, the operation is performed as follows:

```
TEST (ITEM 1 :LIST) = (ITEM 2 :LIST)
  IFTRUE [MAKE NEWCOUNT :NEWCOUNT + 1]
```

```
IFFALSE [IF AND (:NEWCOUNT ≥ :OLDCOUNT)
  (MEMBERP :CHARACTER :POSSIBLE_DIRECTIONS)
  [MAKE MOST_PROBABLE :CHARACTER
  MAKE OLDCOUNT :NEWCOUNT
  MAKE NEWCOUNT 1][MAKE NEWCOUNT 1]]
```

In Microsoft Logo®, the same construct would be handled as a straightforward IF sequence:

```
IF ITEM 1 :LIST = ITEM 2 :LIST
  [MAKE NEWCOUNT :NEWCOUNT + 1]
  [IF AND (OR :NEWCOUNT > :OLDCOUNT :NEWCOUNT = :OLDCOUNT)
  [MEMBERP :CHARACTER :POSSIBLE_DIRECTIONS]
    [MAKE MOST_PROBABLE :CHARACTER
    MAKE OLDCOUNT :NEWCOUNT
    MAKE NEWCOUNT 1][MAKE NEWCOUNT 1]]
```

(Note that the third line in this example is necessary because Microsoft Logo® lacks a "greater than or equal to" function. So we must do both tests joined with an OR.)

## APPLYing Procedures

The ExperLogo® APPLY function is a handy addition to Logo but is not usually implemented in that language. It is similar to a LISP construct, applying a procedure or function to a list of arguments. It is a powerful shorthand for carrying out certain AI programming needs.

In more standard Logos, the same result can be achieved by defining your own APPLY procedure:

```
TO APPLY :FUNCTION :ARGUMENTS
  IF EMPTYP :ARGUMENTS [STOP]
  RUN LIST :FUNCTION FIRST :ARGUMENTS
  MAKE ARGUMENTS BF :ARGUMENTS
  APPLY :FUNCTION :ARGUMENTS
END
```

## Variable Binding

ExperLogo® uses some LISPlike structures that don't have equivalents in LCSI Logo. In the Prologo TRY_EACH procedure, one of these structures, BOUNDP, appears. In fact, this expression is undocumented in ExperLogo® and is a carryover from the language's alter ego, ExperLisp. It is used in that procedure simply to determine whether tracing of the Prologo execution should be carried out or not. If you are not using ExperLogo®, we recommend eliminating the IF construct that uses that relationship. If you wish to be able to trace the execution, you can do so with a menu option or user request.

Another LISP construct, ATOM, appears in this book infrequently. In LISP, an atom is a value that is not part of a list. (This is something of an oversimplification; see Chapter 12 for more information.) We can achieve essentially the same result in more standard Logo by using the function WORDP.

The APPEND function also appears only in Prologo. It is roughly equivalent to the more standard Logo command SE. As it turns out, there are a very small number of situations where SE will not work quite as well as Prologo needs, so we chose to use the APPEND function. But you are not likely to encounter many, if any, such situations, so feel free to substitute SE where the Prologo listing includes the APPEND function.

Finally, LISP offers two types of equality checking. For the most part, we use the standard EQUALP processing in this book. But in some instances, we have chosen the slightly different LISP construct of EQ. For all practical purposes in this book, it is identical to the Logo EQUALP. Its advantage is that it is far faster in its execution, so we have chosen to use it in places where extensive looping is going on and speed could be a significant factor.

## Logical Comparators

ExperLogo® takes full advantage of the Macintosh® keyboard's inclusion of ≥, ≤, and ≠ functions, using these symbols, respectively, for greater-than-or-equal-to, less-than-or-equal-to, and not-equal-to. Microsoft's Logo does not implement these functions directly, requiring instead that we use OR and AND constructs.

Thus the greater-than-or-equal-to function is implemented in Microsoft Logo® by using an OR to connect tests for greater-than and equality. Not-equal-to is simulated by using an AND between less-than and greater-than symbols.

## QuickDraw Graphic Commands

The names by which QuickDraw graphic commands are called varies between the two published Logos for the Mac. Table B-1 provides the Microsoft Logo® equivalents of those QuickDraw and mouse commands that appear in programs in this book.

**Table B-1. Microsoft Logo® Graphic and Mouse Commands**

| *ExperLogo® Command* | *Microsoft Logo® Equivalent* |
|---|---|
| PENPAT | SetPPattern |
| SHOWPEN | PenDown |
| PAINTRECT | FillSh |
| MOVETO | SetX,SetY |
| GETMOUSE | MousePos |

One QuickDraw command in ExperLogo® is implemented indirectly in Microsoft Logo®. The DRAWSTRING function in ExperLogo® writes text into the Graphics Window. In Microsoft Logo®, it is necessary to make the Graphics Window the currently active window and then to use PRINT to display the information:

```
SETWRITE "GRAPHICS
PRINT [Here is a string.]
```

The set of graphic commands used in ExperLogo® to manipulate polygons—used in this book only in the Blocks World program of Chapter 5—are not available in Microsoft Logo® and there is no equivalent way of dealing with their function. Since these operations are used only in conjunction with the pyramids, you can skirt the problem by treating the pyramids as individual shapes rather than as polygon regions. This will make the program run even more slowly, but it will permit it to run.

## Closing Remarks

There may be other differences among Logos implemented on the Mac. We can't anticipate how Logo will be developed by others marketing the language for the Macintosh® system. In most cases, though, noting these differences between ExperLogo® and more standard Logo, carefully reviewing the manual for your version, and doing some experimenting, should help you deal with any incompatibilities you encounter.

CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
L INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
L INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI
ELLIGENCE ARTIFICIAL INT
L INTELLIGENCE ARTIFICIA
CIAL INTELLIGENCE ARTIF
RTIFICIAL INTELLIGENCE A
CE ARTIFICIAL INTELLIGEN
GENCE ARTIFICIAL INTELLI

C

# Suggestions for Further Reading

## Books on Artificial Intelligence

### For the General Reader

Boden, Margaret. *Artificial Intelligence and Natural Man*. New York: Basic Books, 1977. 473 pages, with extensive bibliography and comprehensive index. One of the more readable books on the subject which, though somewhat dated, remains an intelligent reader's excellent choice as a place to start learning about AI. Surveys much of what was valuable at the time the book was written. Avoids programming examples but contains many examples of how programs run.

Hofstadter, Douglas. *Godel, Escher, Bach: An Eternal Golden Braid*. New York: Basic Books, 1979. 800 pages. A well-deserved Pulitzer Prize was awarded to Dr. Hofstadter for this work. It is difficult reading in some places and isn't the kind of book you sit down to relax with for an evening. But for a highly readable, intelligent, perceptive, and thorough introduction to the panorama and issues of the rapidly growing field of cognitive science, of which AI is a part, this book is difficult to beat. Borders on "must" reading for everyone; crosses that border for serious students of the subject.

Hofstadter, Douglas and Daniel C. Dennett, eds. *The Mind's I: Fantasies and Reflections on Self & Soul*. New York: Basic Books, 1981. 464 pages, with annotated bibliography and extensive index. Something of a sequel to *Godel, Escher, Bach*. The coeditors bill themselves as "composers and arrangers" and the metaphor is apt. A collection of essays and fiction on cognitive science, interconnected by witty and incisive comments by Hofstadter and Dennett. More readable than its predecessor, this book makes no less significant a contribution to the field.

Hunt, Morton. *The Universe Within: A New Science Explores the Human Mind*. New York: Simon and Schuster, 1982. 364 pages, plus solutions to problems, end notes, bibliography (cited references only), and index. This is a highly readable and immensely enjoyable overview of the emerging field of cognitive science. The author understands what most of his readers will and will not know before they read this book. As a result, he amplifies where amplification is needed and doesn't talk down to the reader who has already done some reading or thinking in the field. Of particular interest here is his entertaining Chapter 9, "Mind and Supermind," in which he explores what the human mind and the computer mind have in common and, more important to Hunt, what they don't have in common. Highly recommended reading for the curious.

Schank, Roger C. *The Cognitive Computer: On Language, Learning and Artificial Intelligence*. Reading, MA: Addison-Wesley, 1984. 264 pages, plus index. Easily one of the most accessible books on the subject, this widely read work is written in lay language. It represents an attempt on the part of one of the premier researchers in the field to popularize the subject and his views on it, which are considerable and somewhat controversial. There is much of value here for the intelligent general reader and no knowledge of programming is needed to understand it. On the other hand, there is a certain lack of depth and an overabundance of opinion. Dr. Schank has a perception of the field of AI as it relates to cognition and he doesn't hesitate to use this book as a vehicle for spreading that view. Nonetheless, a very useful work and one well worth reading, but not as a primary source of information.

Simon, Geoff. *Are Computers Alive?* Boston: Birkhauser, 1983. 195 pages, with bibliography and index. An unusual book that addresses the issue of whether computers—or more appropriately, their future offspring, robots and androids—should be properly viewed as a new life form entitled to rights, protection, and so on. No basic data here, but an entertaining diversion nonetheless, and one that may provoke more thinking about what computers are *not* than any other book on our list.

Weizenbaum, Joseph. *Computer Power and Human Reason: From Judgment to Calculation*. San Francisco: Freeman, 1976. 300 pages. Dr. Weizenbaum is one of the best-known critics of AI and its proponents. This book is his classic work on the subject of the limits of computing. He argues against the possibility that computers *can* do certain things and adds his view that they should not be *allowed* to intrude into other areas. If you're interested in the critics' side of the underlying issues in AI, this is the most important book of its kind.

**More Technical Materials**

Barr, Avron and Edward A. Feigenbaum, eds. *The Handbook of Artificial Intelligence*. Los Altos, CA: William Kaufmann, Inc., 1981. 3 vols. Total of 1302 pages of text, plus extensive indexes and a comprehensive bibliography. The current bible of AI research, this set of works collects much of the important writing on the subject from the early 1970s to its publication date. As with any collection, readability varies dramatically. Not for the technically unsophisticated, but on the whole well assembled and certainly a seminal work. Particularly strong in the areas of theoretical work and experimentation, less so in terms of real-world uses and applications.

Krutch, John. *Experiments in Artificial Intelligence for Small Computers*. Indianapolis: Howard W. Sams, 1981. 106 pages, plus index. This book collects several programs, written in BASIC, that demonstrate AI concepts and that can be run on microcomputers. If you have an interest in languages and how they can be applied to AI ideas, this book makes the tacit argument that BASIC can be used for AI.

Nilsson, Nils J. *Principles of Artificial Intelligence*. Palo Alto, CA: Tioga Publishing, 1980. 476 pages. A classic work aimed at the technical reader interested in the theoretical and linguistic framework of AI. Quite readable, but somewhat out of date. Does a good job of explaining some of the more difficult ideas behind AI programs, especially the concept of unification.

Winston, Patrick Henry. *Artificial Intelligence*, 2nd Edition. Reading, MA: Addison-Wesley, 1984. 527 pages. The most current and useful book on the field for the technical reader. Winston is one of the most widely read authors of texts and technical materials in the field of AI. The best comprehensive overview of the technical, programming, and conceptual issues involved in bringing intelligence to machines. Not for the faint-at-heart.

## Books on Logo

(Author's Note: There is a dearth of Logo programming texts and books that go beyond a simplistic use of Turtle graphics. One book, Dr. Thornburg's, is known to be in the works and rumors persist that others will be forthcoming. I hope so.)

Martin, Donald, Marijane Paulsen, and Stephen Prata. *Apple Logo Programming Primer*. Indianapolis: Howard W. Sams & Co., 1984. 448 pages, plus index. This is the best introduction to the language available. It focuses on structured programming approaches, which I have adopted with some minor modifications in the programs in this book. Going beyond graphics, it does a fine job of explaining property lists and data management concepts. Many examples. If you don't have access to an Apple II series, the Logo is still standard enough to be used with minor modifications on other popular micros. A good book if you really want to learn Logo. A companion book for the IBM PC is also available from the same publisher.

Thornburg, David. *Beyond Turtle Graphics*. Dr. Thornburg, one of the most prolific authors in the field of Logo and easily one of its most visible and vocal defenders, indicates that this book will transcend the usual introductory work on Logo and explore advanced concepts. Portions of the book were used by Thornburg in a class he taught at Stanford University in the spring of 1985 on AI programming ideas in Logo on the Macintosh®. Watch for this one; it could be one of the best Logo buys when it emerges, given Thornburg's sterling reputation in the field. In press, 1985.

Waite, Mitchell, Don Martin, and Jennifer Martin. *88 Apple Logo Programs*. Indianapolis: Howard W. Sams, 1984. 422 pages, no index. This collection of programs is noteworthy for the number of interesting and instructive programs it crams into a limited space. A companion book for the IBM PC is also available.

Watt, Daniel. *Learning With Logo*. New York: McGraw-Hill, 1983. 358 pages, plus index. A generic Logo teaching book, particularly useful for a young person interested in learning Logo on his or her own or for a parent interested in teaching a child Logo. It offers lots of hints along the way for how to teach ideas in Logo. Focuses on Terrapin, Apple, and TI Logo, but among them there is enough commonality that adapting the book to the Mac and other computers ought not be too difficult. Profusely illustrated.

## Books on LISP

Cherniak, Eugene, Christopher K. Riesback, and Drew V. McDermott. *Artificial Intelligence Programming*. Hillsdale, NJ: Lawrence Erlbaum Associates, 1980. 310 pages, plus bibliography and indexes. An advanced LISP programming text that is definitely not for the uninitiated. After reading a more basic LISP text, however, this book can be quite useful to the reader who wonders what to do with this language. It offers abundant examples and suggests many ways for the student reader to explore the language.

Schank, Roger C. and Christopher K. Riesback. *Inside Computer Understanding: Five Programs Plus Miniatures*. Hillsdale, NJ: Lawrence Erlbaum Associates, 1981. 372 pages, plus bibliography and indexes. For the advanced LISPer, this book provides two useful things: insights into how some of the better-known AI programs in the research literature work and scaled-down versions for implementation on microcomputers. All of the programs are generally in the field of natural language processing. Not for the beginner, this is more of an intermediate book.

Touretzky, David S. *LISP: A Gentle Introduction to Symbolic Computation*. New York: Harper & Row, 1984. 298 pages, plus answers to exercises and index. Easily the least intimidating introductory text on the language. Using a plethora of hands-on examples, this book leads the reader step-by-step down the path to learning LISP. It is as "gentle" as its title promises. If it has one failing, it is that it stops far short of providing the reader with anything resembling a comprehensive understanding of the language. Even the "Advanced Topics" in each chapter fall short of this. But that shouldn't stop you. It is one of the two best books to use to learn LISP. The other is the next one in our bibliography.

Winston, Patrick Henry and Berthold K. P. Horn. *LISP*. Reading, MA: Addison-Wesley, 1981. 314 pages, plus bibliography, index, and four helpful appendices. A classic one-year textbook in LISP programming for the serious student. Dry in its writing approach (contrasted with Touretzky's work), this book takes a no-nonsense approach to teaching the language. Its problems are challenging and solutions are provided. If you're interested in learning LISP for practical programming use rather than for casual experimentation, this book is preferable to Touretzky's. A good idea would be to get both, go through Touretzky's and then tackle Winston's.

## Books on Prolog

(Books on Prolog are not yet widely available in the United States. These two offer a beginning and an advanced approach. Others are being published frequently. Check your bookstore for latest releases.)

Campbell, J.A., ed. *Implementations of Prolog*. Chichester, UK: Ellis Harwood Ltd., 1984. 388 pages, plus minimal index. This book is highly technical, consisting of a collection of papers and articles by various Prolog researchers on the inner workings of the language. Readers seriously interested in Prolog should find in this book a great many in-depth pointers to future research on specific Prolog issues.

Clark, K. L. and F. G. McCabe. *micro-PROLOG: Programming in Logic*. Englewood Cliffs, NJ: Prentice-Hall, 1984. 364 pages, with answers to exercises and index. This is the best introductory book on this language. While it becomes hard to follow at times and condenses some of its explanations, it is nonetheless possible to become a competent Prolog programmer with this book and some practice.

## Books on Expert Systems

### For the General Reader

Hayes, J.E., and D. Michie, eds. *Intelligent Systems: The Unprecedented Opportunity*. Chichester, UK: Ellis Harwood, Ltd., 1984. 199 pages, plus index. Though there is some technically challenging material here, the book also contains some very thought-provoking and insightful writing on the role of expert systems and their future. Of particular interest is Dr. Edward Feigenbaum's excellent piece, "Knowledge Engineering: The Applied Side."

Hayes-Roth, Frederick, Donald A. Waterman, and D. Lenat. *Building Expert Systems*. Reading, MA: Addison-Wesley, 1983. Although technical, this book is still readable. It provides a comprehensive overview of the field: technical considerations, impact of the technology, and problem areas. May tell you more than you need or want to know on the subject.

**Technical Materials**

Brownston, Lee, Robert Farrell, Elaine Kant, and Nancy Martin. *Programming Expert Systems in OPS5*. Reading, MA: Addison-Wesley, 1985. 411 pages, plus bibliography, answers to exercises, index, and excellent glossary. This is the first comprehensive book to be published on the popular OPS5 programming language, a language designed specifically for expert systems development work. It contains a great many hands-on examples, and it is an excellent introduction to the idea of production rule systems as well. If you have access to ExperLisp and ExperOPS5, this book could make you at least comfortable with expert systems programming.

Naylor, Chris. *Build Your Own Expert System*. Cheshire, UK: Sigma Technical Press, 1983. 246 pages, including annotated bibliography and minimal index. This book uses Apple and Spectrum BASIC listings and examples to explain the ideas involved in expert system design. The book is entertaining, brightly written, and enjoyable. Beyond that, the programs work and although they are clearly not intended to be full-blown expert systems, they provide some clear insights into how such systems are put together.

## Books on Natural Language Processing

Dyer, Michael G. *In-Depth Understanding: A Computer Model of Integrated Processing for Narrative Comprehension*. For the reader willing to work a little, Dyer has managed to make accessible a very complex concept and approach to natural language processing. Though the technical vocabulary can be troublesome, if you persist, you can emerge from this book with a deeper appreciation for the incredible array of problems involved in this field of AI.

Winograd, Terry. *Understanding Natural Language*. New York: Academic Press, 1972. 608 pages. This is the landmark work in this field. If you are seriously interested in research in this area, this is the starting point. While much of what Dr. Winograd wrote in this book has since been clarified or corrected, much of what has gone since was built upon his basic ideas.

# Index

D. A. Richmond

SAMS

The Waite Group

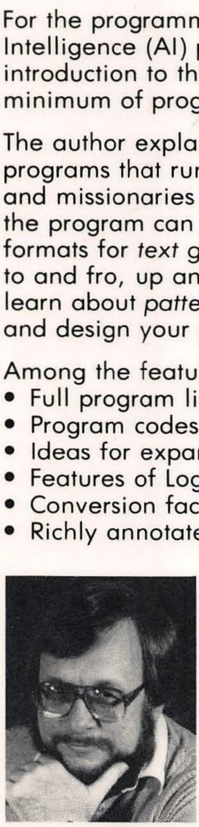
# Artificial Intelligence Programming on the Macintosh

For the programming student and hobbyist eager to learn the fundamentals of Artificial Intelligence (AI) programming theory and technique, this book provides a step-by-step introduction to this next frontier in computer usage. No prior knowledge of AI and only a minimum of programming experience is assumed.

The author explains basic AI concepts and procedures by means of fascinating hands-on programs that run on your Macintosh. Discover *search techniques* as you move cannibals and missionaries from bank to bank in this classic problem; build *property lists* from which the program can make inferences in the Micro-Logician; construct a vocabulary and formats for *text generation* in the form of Haiku poetry in the Poetry Maker; move blocks to and fro, up and down, using *natural language processing* in the Micro Blocks World; learn about *pattern-matching* as you try to beat the system in the Intelligent Maze Game; and design your own *expert systems* based on models given here.

Among the features of this reader-friendly book:

- Full program listings are in the easy-to-learn Logo language
- Program codes and their relationship to AI concepts are fully explained
- Ideas for expansion are given for each of the programs
- Features of Logo, LISP, and Prolog languages are summarized
- Conversion factors are explained for BASIC programmers
- Richly annotated bibliography aids further study



**Dan Shafer** is an independent product consultant and freelance writer in California's Silicon Valley. He is also the coordinator of symposia for attorneys on the impact of Artificial Intelligence and expert systems on their profession. Dozens of his feature articles and product reviews have appeared in computer magazines. As publisher of the monthly newsletter, *Visions: The Shafer Report*, he analyzes future technological trends and impacts. Dan and his wife and four daughters live in Sunnyvale, California.

**The Waite Group** is a San Francisco-based producer of books on personal computing. Acknowledged as a leader in the field, The Waite Group has produced over 50 titles, including such best sellers as *Unix Primer Plus, C Primer Plus, CP/M Primer*, and *Assembly Language Primer for the IBM PC & XT*. Mitchell Waite, president of The Waite Group, has been involved in the computer industry since 1976, when he bought one of the first Apple I computers from Steven Jobs. Besides writing and producing books, Mr. Waite is also a columnist and lecturer on computer-related topics.

**Howard W. Sams & Co.**
A Division of Macmillan, Inc.
4300 West 62nd Street, Indianapolis, IN 46268 USA

0 81262 22447 2

$24.95/22447

ISBN: 0-672-22447-X