PART 1
Introduction to Artificial Intelligence

1 Overview of Artificial Intelligence

Artificial Intelligence (AI), as we know it today, is a relatively new field. Even though some groundwork had been laid earlier, AI began in earnest with the emergence of the modern computer during the 1940s and 1950s. It was the ability of these new electronic machines to store large amounts of information and process it at very high speeds that gave researchers the vision of building systems which could emulate some human abilities.

During the past forty years, we have witnessed the realization of many of these early researchers' visions. We have seen computer systems shrink in size and cost by several orders of magnitude. We have seen memories increase in storage capacity to the point where they equal a significant fraction of the human brain's storage capacity. We have seen the speed and reliability of systems improve dramatically. And, we have seen the introduction of many impressive software tools.

Given these new hardware and software systems and our improved understanding of Homo sapiens, we are on the threshold of witnessing a whole new horizon of exciting innovations. During the next few decades, we will witness the introduction of many intelligent computer systems never dreamed of by the early visionaries. We will see the introduction of systems which equal or exceed human abilities, and see them become an important part of most business and government operations as well as our own daily activities.

1.1 WHAT IS AI?

What is AI exactly? As a beginning we offer the following definition:

> AI is a branch of computer science concerned with the study and creation of computer systems that exhibit some form of intelligence: systems that learn new concepts and tasks, systems that can reason and draw useful conclusions about the world around us, systems that can understand a natural language or perceive and comprehend a visual scene, and systems that perform other types of feats that require human types of intelligence.

Like other definitions of complex topics, an understanding of AI requires an understanding of related terms such as intelligence, knowledge, reasoning, thought, cognition, learning, and a number of computer-related terms. While we lack precise scientific definitions for many of these terms, we can give general definitions of them. And, of course, one of the objectives of this text is to impart special meaning to all of the terms related to AI, including their operational meanings.

Dictionaries define intelligence as the ability to acquire, understand and apply knowledge, or the ability to exercise thought and reason. Of course, intelligence is more than this. It embodies all of the knowledge and feats, both conscious and unconscious, which we have acquired through study and experience: highly refined sight and sound perception; thought; imagination; the ability to converse, read, write, drive a car, memorize and recall facts, express and feel emotions; and much more.

Intelligence is the integrated sum of those feats which gives us the ability to remember a face not seen for thirty or more years, or to build and send rockets to the moon. It is those capabilities which set Homo sapiens apart from other forms of living things. And, as we shall see, the food for this intelligence is knowledge.

Can we ever expect to build systems which exhibit these characteristics? The answer to this question is yes! Systems have already been developed to perform many types of intelligent tasks, and expectations are high for near term development of even more impressive systems. We now have systems which can learn from examples, from being told, from past related experiences, and through reasoning. We have systems which can solve complex problems in mathematics, in scheduling many diverse tasks, in finding optimal system configurations, in planning complex strategies for the military and for business, in diagnosing medical diseases and other complex systems, to name a few. We have systems which can "understand" large parts of natural languages. We have systems which can see well enough to "recognize" objects from photographs, video cameras and other sensors. We have systems which can reason with incomplete and uncertain facts. Clearly, with these developments, much has been accomplished since the advent of the digital computer.

In spite of these impressive achievements, we still have not been able to produce coordinated, autonomous systems which possess some of the basic abilities of a three-year-old child. These include the ability to recognize and remember numerous diverse objects in a scene, to learn new sounds and associate them with objects and concepts, and to adapt readily to many diverse new situations. These are the challenges now facing researchers in AI. And they are not easy ones. They will require important breakthroughs before we can expect to equal the performance of our three-year old.

To gain a better understanding of AI, it is also useful to know what AI is not. AI is not the study and creation of conventional computer systems. Even though one can argue that all programs exhibit some degree of intelligence, an AI program will go beyond this in demonstrating a high level of intelligence to a degree that equals or exceeds the intelligence required of a human in performing some task. AI is not the study of the mind, nor of the body, nor of languages, as customarily found in the fields of psychology, physiology, cognitive science, or linguistics. To be sure, there is some overlap between these fields and AI. All seek a better understanding of the human's intelligence and sensing processes. But in AI the goal is to develop working computer systems that are truly capable of performing tasks that require high levels of intelligence. The programs are not necessarily meant to imitate human senses and thought processes. Indeed, in performing some tasks differently, they may actually exceed human abilities. The important point is that the systems all be capable of performing intelligent tasks effectively and efficiently.

Finally, a better understanding of AI is gained by looking at the component areas of study that make up the whole. These include such topics as robotics, memory organization, knowledge representation, storage and recall, learning models, inference techniques, commonsense reasoning, dealing with uncertainty in reasoning and decision making, understanding natural language, pattern recognition and machine vision methods, search and matching, speech recognition and synthesis, and a variety of AI tools.

How much success have we realized in AI to date? What are the next big challenges? The answers to these questions form a large part of the material covered in this text. We shall be studying many topics which bear directly or indirectly on these questions in the following chapters. We only mention here that AI is coming of an age where practical commercial products are now available including a variety of robotic devices, vision systems that recognize shapes and objects, expert systems that perform many difficult tasks as well as or better than their human expert counterparts, intelligent instruction systems that help pace a student's learning and monitor the student's progress, "intelligent" editors that assist users in building special knowledge bases, and systems which can learn to improve their performance.

1.2 THE IMPORTANCE OF AI

Is AI important? Definitely! AI may be one of the most important developments of this century. It will affect the lives of most individuals in civilized countries by the end of the century. And countries leading in the development of AI by then will emerge as the dominant economic powers of the world.

The importance of AI became apparent to many of the world's leading countries during the late 1970s. Leaders in those countries who recognized the potential for AI were willing to seek approval for long-term commitments for the resources needed to fund intensive research programs in AI. The Japanese were the first to demonstrate their commitment. They launched a very ambitious program in AI research and development. Known as the Fifth Generation, this plan was officially announced in October 1981. It calls for the implementation of a ten-year plan to develop intelligent supercomputers. It is a cooperative effort between government and private companies having an interest in the manufacture of computer products, robotics, and related fi^lds. With a combined budget of about one billion dollars, the Japanese are determined they will realize many of their goals, namely, to produce systems that can converse in a natural language, understand speech and visual scenes, learn and refine their knowledge, make decisions, and exhibit other human traits. If they succeed, and many experts feel they will, their success as a leading economic power is assured.

Following the Japanese, other leading countries of the world have announced plans for some form of AI program. The British initiated a plan called the Alvey Project with a respectable budget. Their goals are not as ambitious as the Japanese but are set to help the British keep abreast and remain in the race. The European Common Market countries have jointly initiated a separate cooperative plan named the ESPRIT program. The French too have their own plan. Other countries including Canada, the Soviet Union, Italy, Austria, and even the Irish Republic and Singapore have made some commitments in funded research and development.

The United States, although well aware of the possible consequences, has made no formal plan. However, steps have been taken by some organizations to push forward in AI research. First, there was the formation of a consortium of private companies in 1983 to develop advanced technologies that apply AI techniques (like VLSI). The consortium is known as the Microelectronics and Computer Technology Corporation (MCC) and is headquartered in Austin, Texas. Second, the Department of Defense Advanced Research Projects Agency (DARPA) has increased its funding for research in AI, including development support in three significant programs: (1) development of an autonomous land vehicle (ALV) (a driverless military vehicle); (2) the development of a pilot's associate (an expert system which provides assistance to fighter pilots), and (3) the Strategic Computing Program (an AI based military supercomputer project). In addition, most of the larger high-tech companies such as IBM, DEC, AT&T, Hewlett Packard, Texas Instruments, and Xerox have their own research programs. A number of smaller companies also have reputable research programs.

Who will emerge as the principal leaders in this race for superiority in the production and sale of that commodity known as knowledge? If forward vision and commitment to purpose are to be the determining factors, then surely the Japanese will be among the leaders if not the leader.

Just how the United States and other leading countries of the world will fare remains to be seen. One thing is clear. The future of a country is closely tied to the commitment it is willing to make in funding research programs in AI.

1.3 EARLY WORK IN AI

As noted above, AI began to emerge as a separate field of study during the 1940s and 1950s when the computer became a commercial reality. Prior to this time, a number of important areas of research that would later help to shape early AI work were beginning to mature. These developments all began to converge during this period. First, there was the work of logicians such as Alonzo Church, Kurt Godel, Emil Post, and Alan Turing. They were carrying on earlier work in logic initiated by Whitehead and Russell, Tarski, and Kleene. This work began in earnest during the 1920s and 1930s. It helped to produce formalized methods for reasoning, the form of logic known as propositional and predicate calculus. It demonstrated that facts and ideas from a language such as English could be formally described and manipulated mechanically in meaningful ways. Turing, sometimes regarded as the father of AI, also demonstrated, as early as 1936, that a simple computer processor (later named the Turing machine) could manipulate symbols as well as numbers.

Second, the new field of cybernetics, a name coined by Norbert Wiener, brought together many parallels between human and machine. Cybernetics, the study of communication in human and machine, became an active area of research during the 1940s and 1950s. It combined concepts from information theory, feedback control systems (both biological and machine), and electronic computers.

Third came the new developments being made in formal grammars. This work was an outgrowth of logic during the early 1900s. It helped to provide new approaches to language theories in the general field of linguistics.

Finally, during the 1950s, the electronic stored program digital computer became a commercial reality. This followed several years of prototype systems including the Mark I Harvard relay computer (1944), the University of Pennsylvania Moore School of Electrical Engineering's ENIAC electronic computer (1947), and subsequent development of the Aberdeen Proving Ground's EDVAC and Sperry-Rand's UNIVAC.

Other important developments during this early period which helped to launch AI include the introduction of information theory due largely to the work of Claude Shannon, neurological theories and models of the brain which were originated by psychologists, as well as the introduction of Boolean algebra, switching theory, and even statistical decision theory.

Of course AI is not just the product of this century. Much groundwork had been laid by earlier researchers dating back several hundred years. Names like Aristotle, Leibnitz, Babbage, Hollerith, and many others also played important roles in building a foundation that eventually led to what we now know as AI.

Work after 1950

During the 1950s several events occurred which marked the real beginning of AI. This was a period noted for the chess playing programs which were developed by researchers like Claude Shannon at MIT (Shannon, 1952, 1955) and Allen Newell at the RAND Corporation (Newell and Simon, 1972). Other types of game playing and simulation programs were also being developed during this time. Much effort was being expended on machine translation programs, and there was much optimism for successful language translation using computers (Weaver, 1955). It was felt that the storage of large dictionaries in a computer was basically all that was needed to produce accurate translations from one language to another. Although this approach proved to be too simplistic, it took several years before such efforts were aborted.

The mid-1950s are generally recognized as the official birth date of AI when a summer workshop sponsored by IBM was held at Dartmouth College. Attendees at this June 1956 seminar included several of the early pioneers in AI including Herbert Gelernter, Trenchard More, John McCarthy, Marvin Minsky, Allen Newell, Nat Rochester, Oliver Selfridge, Claude Shannon, Herbert Simon, and Ray Solomonoff (Newell and Simon, 1972). Much of their discussion focused on the work they were involved in during this period, namely automatic theorem proving and new programming languages.

Between 1956 and 1957 the Logic Theorist, one of the first programs for automatic theorem proving, was completed by Newell, Shaw, and Simon (Newell and Simon, 1972). As part of this development, the first list-processing language called IPL (Information Processing Language) was also completed. Other important events of this period include the development of FORTRAN (begun in 1954) and Noam Chomsky's work between 1955 and 1957 on the theory of generative grammars. Chomsky's work had a strong influence on AI in the area of computational linguistics or natural language processing.

Important events of the late 1950s were centered around pattern recognition and self-adapting systems. During this period Rosenblatt's perceptrons (Rosenblatt, 1958) were receiving much attention. Perceptrons are types of pattern recognition devices that have a simple learning ability based on linear threshold logic (described in detail in Chapter 17). This same period (1958) marked the beginning of the development of LISP by John McCarthy, one of the recognized programming languages of AI. It also marked the formation of the Massachusetts Institute of Technology's AI laboratory. Several important programming projects were also begun during the late 1950s, including the General Problem Solver (GPS) developed by Newell, Shaw, and Simon (Ernst and Newell, 1969) written in IPL, Gelernter's geometry theorem-proving machine written in FORTRAN at the IBM Research Center, and the Elementary Perceiver and Memorizer (EPAM) developed by Edward Feigenbaum and Herbert Simon and written in IPL.

GPS was developed to solve a variety of problems ranging from symbolic integration to word puzzles (such as the missionary-cannibal problem). GPS used a problem-solving technique known as means-end analysis discussed later in Chapter 9. The geometry theorem-proving machine of Gelernter was developed to solve high-school level plane geometry problems. From basic axioms in geometry, the system developed a proof as a sequence of simple subgoals. EPAM was written to study rote learning by machine. The system had a learning and performance component where pairs of nonsense words, a stimulus-response pair, were first learned through repetitive presentations (in different orders). The performance component was then used to demonstrate how well responses to the stimuli were learned.

Some significant AI events of the 1960s include the following.

1961–65	A. L. Samuel developed a program which learned to play checkers at a master's level.
1965	J. A. Robinson introduced resolution as an inference method in logic.
1965	Work on DENDRAL was begun at Stanford University by J. Lederberg, Edward Feigenbaum, and Carl Djerassi. DENDRAL is an expert system which discovers molecular structures given only information of the constituents of the compound and mass spectra data. DENDRAL was the first knowledge-based expert system to be developed.
1968	Work on MACSYMA was initiated at MIT by Carl Engleman, William Martin, and Joel Moses. MACSYMA is a large interactive program which solves numerous types of mathematical problems. Written in LISP, MACSYMA was a continuation of earlier work on SIN, an indefinite integration solving program.

References on early work in AI include McCorduck's *Machines Who Think* (1979), and Newell and Simon's *Human Problem Solving* (1972).

1.4 AI AND RELATED FIELDS

Fields which are closely related to AI and overlap somewhat include engineering, particularly electrical and mechanical engineering, linguistics, psychology, cognitive science, and philosophy. Robotics is also regarded by some researchers as a branch of AI, but this view is not common. Many researchers consider robotics as a separate interdisciplinary field which combines concepts and techniques from AI, electrical, mechanical, and optical engineering.

Psychologists are concerned with the workings of the mind, the mental and emotional processes that drive human behavior. As such, we should not be suprised to learn that researchers in AI have much in common with psychologists. Durir the past 20 years AI has adopted models of thinking and learning from psychology, while psychologists in turn have patterned many of their experiments on questions first raised by AI researchers. AI has given psychologists fresh ideas and enhanced their ability to model human cognitive functions on the computer. In their book *The Cognitive Computer*, Schank and Childers (1984) estimate that ". . . AI has contributed more to psychology than any other discipline for some time."

Because they share so many common interests, it has been claimed that AI researchers think less like computer scientists than they do psychologists and philosophers. As a consequence, researchers from AI and psychology have joined together at some universities to form a separate discipline known as *cognitive science*. This name has also been adopted by a few new companies offering AI services and products. Like AI researchers, cognitive scientists are interested in the computation processes required to perform certain human functions, and in the mental and computational states related to such processes. Like computer science, cognitive science is still searching for a theory and foundation that will qualify it as a science.

AI also has much in common with engineering, particularly electrical (EE) and mechanical engineering (ME). AI and EE are both interested in computational processes and systems, and how they relate to senses of perception such as vision and speech. ME and AI share common interests in their desire to build intelligent robots. Their goals are to build robots that can see and move around, perform mechanical tasks, and understand human speech.

The field of linguistics shares an interest in the theory of grammars and languages with AI. Both fields have a desire to build a well-founded theory, and to see the development of systems that understand natural languages, that can synthesize speech, and that are capable of language translations.

Finally, AI has some overlap with almost all fields in that it offers the potential for broad applications. Applications have already been proven in such areas as medicine, law, manufacturing, economics, banking, biology, chemistry, defense, civil engineering, and aerospace to name a few. And, it is only a matter of time before applications will permeate the home.

1.5 SUMMARY

In this introductory chapter, we have defined AI and terms closely related to the field. We have shown how important AI will become in the future as it will form the foundation for a number of new consumer commodities, all based on knowledge. It was noted that countries willing to commit appropriate resources to research in this field will emerge as the world's economic leaders in the not too distant future.

We briefly reviewed early work in AI, considering first developments prior to 1950, the period during which the first commercial computers were introduced. We then looked at post-1950 developments during which AI was officially launched as a separate field of computer science. Fields which overlap with and are closely related to AI were also considered, and the areas of commonality between the two presented.

2

Knowledge: General Concepts

The important role that knowledge plays in building intelligent systems is now widely accepted by practictioners in AI. Recognition of this important fact was necessary before successful, real-world systems could be built. Because of this importance, a significant amount of coverage is devoted to knowledge in this text. We will be looking at the important roles it plays in all of the subfields of AI. In this chapter, we attempt to set the stage for what follows by gaining some familiarity with knowledge, and a better appreciation of its power. As noted in Chapter 1, the whole text is in a sense all about knowledge, and is organized to reflect the way it dominates thinking and the direction of research in the field of AI.

2.1 INTRODUCTION

Early researchers in AI believed that the best approach to solutions was through the development of general purpose problem solvers, that is, systems powerful enough to prove a theorem in geometry, to perform a complex robotics task, or to develop a plan to complete a sequence of intricate operations. To demonstrate their theories, several systems were developed including several logic theorem provers and a general problem solver system (described in Chapter 9).

All of the systems developed during this period proved to be impotent as general problem solvers. They required much hand tailoring of problem descriptions and ad hoc guidance in their solution steps. The approaches they used proved to be too general to be effective. The systems became effective only when the solution methods incorporated domain specific rules and facts. In other words, they became effective as problem solvers only when specific knowledge was brought to bear on the problems. The realization that specific knowledge was needed to solve difficult problems gradually brought about the use of domain specific knowledge as an integral part of a system. It eventually led to what we now know as knowledge-based systems. Since the acceptance of this important fact, successful problem solvers in many domains have been developed.

2.2 DEFINITION AND IMPORTANCE OF KNOWLEDGE

Knowledge can be defined as the body of facts and principles accumulated by humankind or the act, fact, or state of knowing. While this definition may be true, it is far from complete. We know that knowledge is much more than this. It is having a familiarity with language, concepts, procedures, rules, ideas, abstractions, places, customs, facts, and associations, coupled with an ability to use these notions effectively in modeling different aspects of the world. Without this ability, the facts and concepts are meaningless and, therefore, worthless. The meaning of knowledge is closely related to the meaning of intelligence. Intelligence requires the possession of and access to knowledge. And a characteristic of intelligent people is that they possess much knowledge.

In biological organisms, knowledge is likely stored as complex structures of interconnected neurons. The structures correspond to symbolic representations of the knowledge possessed by the organism, the facts, rules, and so on. The average human brain weighs about 3.3 pounds and contains an estimated number of 10^{12} neurons. The neurons and their interconnection capabilities provide about 10^{14} bits of potential storage capacity (Sagan, 1977).

In computers, knowledge is also stored as symbolic structures, but in the form of collections of magnetic spots and voltage states. State-of-the-art storage in computers is in the range of 10^{12} bits with capacities doubling about every three to four years. The gap between human and computer storage capacities is narrowing rapidly. Unfortunately, there is still a wide gap between representation schemes and efficiencies.

A common way to represent knowledge external to a computer or a human is in the form of written language. For example, some facts and relations represented in printed English are

Joe is tall.
Bill loves Sue.
Sam has learned to use recursion to manipulate linked lists in several programming languages.

The first item of knowledge above expresses a simple fact, an attribute possessed by a person. The second item expresses a complex binary relation between two persons. The third item is the most complex, expressing relations between a person and more abstract programming concepts. To truly understand and make use of this knowledge, a person needs other world knowledge and the ability to reason with it.

Knowledge may be declarative or procedural. Procedural knowledge is compiled knowledge related to the performance of some task. For example, the steps used to solve an algebraic equation are expressed as procedural knowledge. Declarative knowledge, on the other hand, is passive knowledge expressed as statements of facts about the world. Personnel data in a database is typical of declarative knowledge. Such data are explicit pieces of independent knowledge.

Frequently, we will be interested in the use of heuristic knowledge, a special type of knowledge used by humans to solve complex problems. Heuristics are the knowledge used to make good judgments, or the strategies, tricks, or "rules of thumb" used to simplify the solution of problems. Heuristics are usually acquired with much experience. For example, in locating a fault in a TV set, an experienced technician will not start by making numerous voltage checks when it is clear that the sound is present out the picture is not, but instead will immediately reason that the high voltage flyback transformer or related component is the culprit. This type of reasoning may not always be correct, but it frequently is, and then it leads to a quick solution.

Knowledge should not be confused with data. Feigenbaum and McCorduck (1983) emphasize this difference with the following example. A physican treating a patient uses both knowledge and data. The data is the patient's record, including patient history, measurements of vital signs, drugs given, response to drugs, and so on, whereas the knowledge is what the physician has learned in medical school and in the years of internship, residency, specialization, and practice. Knowledge is what the physician now learns in journals. It consists of facts, prejudices, beliefs, and most importantly, heuristic knowledge.

Thus, we can say that knowledge includes and requires the use of data and information. But it is more. It combines relationships, correlations, dependencies, and the notion of gestalt with data and information.

Even with the above distinction, we have been using knowledge in its broader sense up to this point. At times, however, it will be useful or even necessary to distinguish between knowledge and other concepts such as belief and hypotheses. For such cases we make the following distinctions. We define *belief* as essentially any meaningful and coherent expression that can be represented. Thus, a belief may be true or false. We define a *hypothesis* as a justified belief that is not known to be true. Thus, a hypothesis is a belief which is backed up with some supporting evidence, but it may still be false. Finally, we define *knowledge* as true justified belief. Since these distinctions will be made more formal in later chapters, we need not attempt to give any further definitions of truth or justification at this time.

Two other knowledge terms which we shall occasionally use are epistemology and metaknowledge. *Epistemology* is the study of the nature of knowledge, whereas *metaknowldge* is knowledge about knowledge, that is, knowledge about what we know.

In this section we have tried to give a broader definition of knowledge than that commonly found in dictionaries. Clearly, we have not offered a scientific definition, nor will we in this text. That will have to wait. But, without a scientific definition, we are not able to measure knowledge. How then will we know when a system has enough knowledge to perform a specified task? Can we expect to build intelligent systems without having a more precise definition of either knowledge or intelligence? In spite of our ignorance about knowledge, the answer is definitely yes. We can and have built intelligent systems as we shall see in the following chapters.

As it happened, in 1950 Turing proposed a way to demonstrate if a machine can think and, therefore, exhibit intelligence. Known as the Turing test, it involves isolating a person in a room with only a computer teletype. If the person cannot distinguish between a man imitating a woman and a computer imitating a man imitating a woman on the teletype, the computer succeeded in passing the test. (Turing's test is often mistakenly understood to be simply a test of whether or not a person can distinguish between some other hidden person and a computer impersonating a person.) To date no one has developed a system able to pass the Turing test. Of course, it may be that no one has tried. Even so, systems need not pass such a test to be useful, and many systems have already been built that exhibit a high level of intelligence.

Finally, our overall picture of knowledge cannot be complete without also knowing the meaning of closely related concepts such as understanding, learning, thinking, remembering, and reasoning. These concepts all depend on the use of knowledge. But then just what is learning, or reasoning, or understanding? Here too we will find dictionary definitions lacking. And, as in the case of knowledge and intelligence, we cannot give scientific definitions for any of these terms either. But, we will gain a deeper understanding and appreciation for the concepts through our study of AI. In particular, we will see the difficulties encountered in attempting to implement such concepts in computer programs. For in programs, one must be precise.

The Importance of Knowledge

AI has given new meaning and importance to knowledge. Now, for the first time, it is possible to "package" specialized knowledge and sell it with a system that can use it to reason and draw conclusions. The potential of this important development is only now beginning to be realized. Imagine being able to purchase an untiring, reliable advisor that gives high level professional advice in specialized areas, such as manufacturing techniques, sound financial strategies, ways to improve one's health, top marketing sectors and strategies, optimal farming plans, and many other important matters. We are not far from the practical realization of this, and those who create and market such systems will have more than just an economic advantage over the rest of the world.

As noted in Chapter 1, the Japanese recognized the potential offered with these knowledge systems. They were the first to formally proceed with a plan to commit substantial resources toward an accelerated program of development for super-computers and knowledge-based systems. In their excellent book on the Fifth Generation, Feigenbaum and McCorduck (1983) present convincing arguments for the importance that should be ascribed to such programs. They argue that the time is right for the exploitation of AI and that the leaders in this field will become the leaders in world trade. By forging ahead in research and the development of powerful knowledge-based systems, the Japanese are assuring themselves of a leading role in the control and dissemination of packaged knowledge. Feigenbaum and McCorduck laud the Japanese for their boldness and farsightedness in moving ahead with this ambitious program.

2.3 KNOWLEDGE-BASED SYSTEMS

One of the important lessons learned in AI during the 1960s was that general purpose problem solvers which used a limited number of laws or axioms were too weak to be effective in solving problems of any complexity. This realization eventually led to the design of what is now known as knowledge-based systems, systems that depend on a rich base of knowledge to perform difficult tasks.

Edward Feigenbaum summarized this new thinking in a paper at the International Joint Conference on Artificial Intelligence (IJCAI) in 1977. He emphasized the fact that the real power of an expert system comes from the knowledge it possesses rather than the particular inference schemes and other formalisms it employs. This new view of AI systems marked the turning point in the development of more powerful problem solvers. It formed the basis for some of the new emerging expert systems being developed during the 1970s including MYCIN, an expert system developed to diagnose infectious blood diseases.

Since this realization, much of the work done in AI has been related to so-called knowledge-based systems, including work in vision, learning, general problem solving, and natural language understanding. This in turn has led to more emphasis being placed on research related to knowledge representation, memory organization, and the use and manipulation of knowledge.

Knowledge-based systems get their power from the expert knowledge that has been coded into facts, rules, heuristics, and procedures. The knowledge is stored in a knowledge base separate from the control and inferencing components (Figure 2.1). This makes it possible to add new knowledge or refine existing knowledge without recompiling the control and inferencing programs. This greatly simplifies the construction and maintenance of knowledge-based systems.

In the knowledge lies the power! This was the message learned by a few farsighted researchers at Stanford University during the late 1960s and early 1970s.

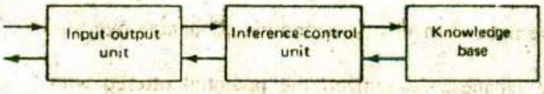


Figure 2.1 Components of a knowledge-based system.

The proof of their message was provided in the first knowledge-based expert systems which were shown to be more than toy problem solvers. These first systems were real world problem solvers, tackling such tasks as determining complex chemical structures given only the atomic constituents and mass spectra data from samples of the compounds, and later performing medical diagnoses of infectious blood diseases.

2.4 REPRESENTATION OF KNOWLEDGE

Given the fact that knowledge is important and in fact essential for intelligent behavior, the representation of knowledge has become one of AI's top research priorities. What exactly is meant by knowledge representation? As defined above, knowledge consists of facts, concepts, rules, and so forth. It can be represented in different forms, as mental images in one's thoughts, as spoken or written words in some language, as graphical or other pictures, and as character strings or collections of magnetic spots stored in a computer (Figure 2.2). The representations we shall be concerned with in our study of AI are the written ones (character strings, graphs, pictures) and the corresponding data structures used for their internal storage.

Any choice of representation will depend on the type of problem to be solved and the inference methods available. For example, suppose we wish to write a program to play a simple card game using the standard deck of 52 playing cards. We will need some way to represent the cards dealt to each player and a way to express the rules. We can represent cards in different ways. The most straightforward way is to record the suit (clubs, diamonds, hearts, spades) and face values (ace, 2, 3, . . . , 10, jack, queen, king) as a symbolic pair. So the queen of hearts might

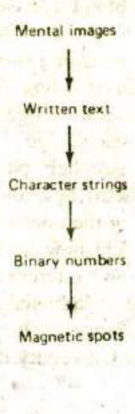


Figure 2.2 Different levels of knowledge representation.

be represented as <queen, hearts>. Alternatively, we could assign abbreviated codes (c6 for the 6 of clubs), numeric values which ignore suit (1, 2, . . . ,13), or some other scheme. If the game we wish to play is bridge, suit as well as value will be important. On the other hand, if the game is black jack, only face values are important and a simpler program will result if only numeric values are used.

To see how important a good representation is, one only needs to try solving a few simple problems using different representations. Consider the problem of discovering a pattern in the sequence of numbers 1 1 2 3 4 7. A change of base in the number from 10 to 2 transforms the number to

0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1.

Clearly a representation in the proper base greatly simplifies finding the pattern solution.

Sometimes, a state diagram representation will simplify solutions. For example, the Towers of Hanoi problem requires that *n* discs (say *n* = 3), each a different size, be moved from one of three pegs to a third peg without violating the rule a disc may only be stacked on top of a larger disc. Here, the states are all the possible disc-peg configurations, and a valid solution path can easily be traced from the initial state through other connected states to the goal state.

Later we will study several representation schemes that have become popular among AI practitioners. Perhaps the most important of these is first order predicate logic. It has become important because it is one of the few methods that has a well-developed theory, has reasonable expressive power, and uses valid forms of inferring. Its greatest weakness is its limitation as a model for commonsense reasoning. A typical statement in this logic might express the family relationship of fatherhood as FATHER(john, jim) where the predicate father is used to express the fact that John is the father of Jim.

Other representation schemes include frames and associative networks (also called semantic and conceptual networks), fuzzy logic, modal logics, and object-oriented methods. Frames are flexible structures that permit the grouping of closely related knowledge. For example, an object such as a ball and its properties (size, color, function) and its relationship to other objects (to the left of, on top of, and so on) are grouped together into a single structure for easy access. Networks also permit easy access to groups of related items. They associate objects with their attributes, and linkages show their relationship to other objects.

Fuzzy logic is a generalization of predicate logic, developed to permit varying degrees of some property such as tall. In classical two-valued logic, TALL(john) is either true or false, but in fuzzy logic this statement may be partially true. Modal logic is an extension of classical logic. It was also developed to better represent commonsense reasoning by permitting conditions such as likely or possible. Object oriented representations package an object together with its attributes and functions, therefore, hiding these facts. Operations are performed by sending messages between the objects.

Another representation topic covered more fully later is uncertainty. Not all knowledge is known with certainty. Knowledge may be vague, contradictory, or incomplete. Yet we would still like to be able to reason and make decisions. Humans do remarkably well with fuzzy, incomplete knowledge. We would also like our AI programs to demonstrate this versatility.

2.5 KNOWLEDGE ORGANIZATION

The organization of knowledge in memory is key to efficient processing. Knowledge-based systems may require tens of thousands of facts and rules to perform their intended tasks. It is essential then that the appropiate facts and rules be easy to locate and retrieve. Otherwise, much time will be wasted in searching and testing large numbers of items in memory.

Knowledge can be organized in memory for easy access by a method known as indexing. It amounts to grouping the knowledge in a way that key words can be used to access the group. The key words "point" to the knowledge groups. As a result, the search for some specific chunk of knowledge is limited to the group only, a fraction of the knowledge base rather than the whole memory.

The choice of representation can simplify the organization and access operations. For example, frames linked together in a network represent a versatile organization structure. Each frame will contain all closely associated information about an object and pointers to related object frames making it possible to quickly gain access to this information. Subsequent processing then typically involves only a few related frames.

2.6 KNOWLEDGE MANIPULATION

Decisions and actions in knowledge-based systems come from manipulation of the knowledge in specified ways. Typically, some form of input (from a user keyboard or sensors) will initiate a search for a goal or decision. This requires that known facts in the knowledge-base be located, compared (matched), and possibly altered in some way. This process may set up other subgoals and require further inputs, and so on until a final solution is found. The manipulations are the computational equivalent of reasoning. This requires a form of inference or deduction, using the knowledge and inferring rules.

All forms of reasoning require a certain amount of searching and matching. In fact, these two operations by far consume the greatest amount of computation time in AI systems. For this reason it is important to have techniques available that limit the amount of search and matching required to complete any given task. Much research has been done in these areas to find better methods. The research has paid off with methods which help to make many otherwise intractable problems solvable. They help to limit or avoid the so-called combinatorial explosion in problems which are so common in search.

2.7 ACQUISITON OF KNOWLEDGE

One of the greatest bottlenecks in building knowledge-rich systems is the acquisition and validation of the knowledge. Knowledge can come from various sources, such as experts, textbooks, reports, technical articles, and the like. To be useful, the knowledge must be accurate, presented at the right level for encoding, complete in the sense that all essential facts and rules are included, free of inconsistencies, and so on. Eliciting facts, heuristics, procedures, and rules from an expert is a slow, tedious process. Experience in building dozens of expert systems and other knowledge-based systems over the past fifteen years has shown this to be the single most time-consuming and costly part of the building process. This has led to the development of some sophisticated acquisition tools, including a variety of intelligent editors, editors which provide much assistance to the knowledge engineers and system users.

The acquisition problem has also stimulated much research in machine learning systems, that is, systems which can learn new knowledge autonomously without the aid of humans. Since knowledge-based systems depend on large quantities of high quality knowledge for their success, it is essential that better methods of acquisition, refinement, and validation be developed. The ultimate goal is to develop techniques that permit systems to learn new knowledge autonomously and continually improve the quality of the knowledge they possess.

2.8 SUMMARY

In this chapter we have defined and described the importance of knowledge in building intelligent AI computer systems. A definition of knowledge was given, and the differences between knowledge, belief, and hypotheses were described. The difference between knowledge and data was also clarified.

The recognition of the important role that knowledge plays in AI systems has led several countries to commit substantial resources to long-range research programs in AI. In particular, the Japanese government has undertaken a cooperative program with several industrial companies to develop intelligent supercomputers within a ten-year period.

Also in this chapter, we considered some of the basic research priorities related to knowledge-based systems: knowledge representation, knowledge organization, knowledge manipulation, and knowledge acquisition. These topics form the main theme of the remaining chapters.

EXERCISES

2.1 Define and describe the difference between knowledge, belief, hypotheses, and data.

2.2 What is the difference between declarative and procedural knowledge?

2.3 Look up the meaning of epistemology in a good encyclopedia and prepare a definition.

2.4 The Turing test has often been incorrectly interpreted as being a test of whether or not a person could distinguish between responses from a computer and responses from a person. How does this differ from the real Turing test? Are the two tests equivalent? If not, explain why they are not?

2.5 What important knowledge products are currently being marketed like other commodities? What are some new knowledge products likely to be sold within the next ten years?

2.6 Briefly describe the meaning of knowledge representation and knowledge acquisition.

2.7 Give four different ways to represent the fact that John is Bill's father.

3

LISP and Other AI Programming Languages

LISP is one of the oldest computer programming languages. It was invented by John McCarthy during the late 1950s, shortly after the development of FORTRAN. LISP (for *LIS*t *P*rocessing) is particularly suited for AI programs because of its ability to process symbolic information effectively. It is a language with a simple syntax, with little or no data typing and dynamic memory management. There are several dialects of LISP including FRANZLISP, INTERLISP, MACLISP, QLISP, SCHEME, and COMMON LISP. The COMMON LISP version is a recent attempt to standardize the language to make it more portable and easier to maintain.

LISP has become the language of choice for most AI practitioners. It was practically unheard of outside the research community until AI began to gain some popularity ten to fifteen years ago. Since then, special LISP processing machines have been built and its popularity has spread to many new sectors of business and government. In this chapter we give a summary of the important features of LISP and briefly introduce PROLOG and other AI languages.

3.1 INTRODUCTION TO LISP: SYNTAX AND NUMERIC FUNCTIONS

The basic building blocks of LISP are the atom, list, and the string. An atom is a number or string of contiguous characters, including numbers and special characters. A list is a sequence of atoms and/or other lists enclosed within parentheses. A string is a group of characters enclosed in double quotation marks. Examples of atoms, lists and strings are

VALID ATOMS	INVALID ATOMS[1]
this-is-a-symbolic-atom	(abc
bill	123abc
100004352	abcd'ef
mountain_top	(a b)
var	ab cde
block#6	
a12345	
VALID LISTS	INVALID LISTS
(this is a list)	this is not a list
(a (a b) c def)	(abcdef ghij
(father sam (joe bill sue))	)abck efg(
(mon tue wed thur fri sat sun)	(a b c (d e)
( )	((a b) (
VALID STRINGS	INVALID STRINGS
"this is a string"	this is not a string
"a b c d e fgh #$%@!"	"neither is this
"please enter your name"	nor" this"

Since a list may contain atoms as well as other lists, we will call the basic unit members top elements. Thus, the top elements of the list (a b (c d) e (f)) are a, b, (c d), e, and (f). The elements c and d are top elements of the sublist (c d).

Atoms, lists, and strings are the only valid objects in LISP. They are called symbolic-expressions or s-expressions. Any s-expression is potentially a valid program. And those, believe it or not, are essentially the basic syntax rules for LISP. Of course, to be meaningful, a program must obey certain rules of semantics, that is, a program must have meaning.

LISP programs run either on an interpreter or as compiled code. The interpreter examines source programs in a repeated loop, called the read-evaluate-print loop. This loop reads the program code, evaluates it, and prints the values returned by the program. The interpreter signals its readiness to accept code for execution by printing a prompt such as the -> symbol. For example, to find the sum of the three numbers 5, 6, and 9 we type after the prompt the following function call:

```
-> (+ 5 6 9)
20
->
```

[1] Some dialects require that symbolic atoms begin with a letter and do not include parens or single quotes.

Note that LISP uses *prefix* notation, and the + symbol is the function name for the sum of the arguments that follow. The function name and its arguments are enclosed in parentheses to signify that it is to be evaluated as a function. The read-evaluate-print loop reads this expression, evaluates it, and prints the value returned (20). The interpreter then prints the prompt to signal its readiness to accept the next input. More complicated computations can be written as a single embedded expression. For example, to compute the centigrade equivalent of the Fahrenheit temperature 50, for the mathematical expression (50 * 9 / 5) + 32 we would write the corresponding LISP function

```
-> (+ (* (/ 9 5) 50) 32)
122
->
```

Each function call is performed in the order in which it occurs within the parentheses. But, in order to compute the sum, the argument (* (/ 9 5) 50) must first be evaluated. This requires that the product of 50 and 9/5 be computed, which in turn requires that the quotient 9/5 be evaluated. The embedded function (/ 9 5) returns the quotient 1.8 to the multiply function to give (* 1.8 50). This is then evaluated and the value 90 is returned to the top (sum) function to give (+ 90 32). The final result is the sum 122 returned to the read-evaluate-print loop for printing.

The basic numeric operations are +, -, *, and /. Arguments may be integers or real values (floating point), and the number of arguments a function takes will, of course, differ. For example, + and * normally take zero or more arguments, while - and / take two. These and a number of other basic functions are predefined in LISP. Examples of function calls and the results returned are given in Table 3.1. In addition to these basic calls, some LISP implementations include mnemonic names for arithmetic operations such as plus and times.

LISP tries to evaluate everything, including the arguments of a function. But, three types of elements are special in that they are constant and always evaluate to themselves, returning their own value: numbers, the letter *t* (for logical true), and

TABLE 3.1 PREDEFINED NUMERIC FUNCTIONS

Function call	Value returned	Remarks
(+ 3 5 8 4)	20	+ takes zero or more arguments. The sum of zero arguments is 0.
(− 10 12)	− 2	− takes two arguments.
(* 2 3 4)	24	* takes zero or more arguments. The product of no arguments is 1, and the product of 1 argument is the value of the argument.
(/ 25 2)	12.5	/ takes two arguments.

nil (for logical false). Nil is also the same as the empty list (). It is the only object in LISP that is both an atom and a list. Since these elements return their own value, the following are valid expressions.

```
->6
6
->t
T
->NIL
NIL
```

3.2 BASIC LIST MANIPULATION FUNCTIONS IN LISP

Sometimes we wish to take atoms or lists literally and not have them evaluated or treated as function calls as, for example, when the list represents data. To accomplish this, we precede the atom or the list with a single quotation mark, as in 'man or as in '(a b c d). The quotation mark informs the interpreter that the atom or list should not be evaluated, but should be taken literally as an atom or list.

Variables in LISP are symbolic (nonnumeric) atoms. They may be assigned values, that is, bound to values with the function setq. Setq takes two arguments, the first of which must be a variable. It is never evaluated and should not be in quotation marks. The second argument is evaluated (unless in quotation marks) and the result is bound to the first argument. The variable retains this value until a new assignment is made. When variables are evaluated, they return the last value bound to them. Trying to evaluate an undefined variable (one not previously bound to a value) results in an error. Some examples of the use of setq are as follows (note that comments in LISP code may be placed anywhere after a semicolon).

```
->(setq x 10)          ;the number 10 evaluates to itself
10                     ;is bound to x and 10 is returned
->x                    ;the variable x is evaluated to
10                     ;return the value it is bound to
->(setq x (+ 3 5))     ;x is reset to the value (+ 3 5)
8                      ;and that value returned
->(setq x '(+ 3 5))    ;x is reset to the literal value
(+ 3 5)                ;(+ 3 5), quote inhibits evaluation
->y                    ;the variable y was not previously
Unbound variable:Y     ;bound to a value, causing an error
```

Some basic symbol processing functions are car, cdr, cons, and list. Examples of these functions are given in Table 3.2. Car takes one argument, which must be a list. It returns the first top element of the list. Cdr also takes a list as its argument, and it returns a list consisting of all elements except the first. Cons takes two

TABLE 3.2 BASIC LIST MANIPULATION FUNCTIONS

Function call	Value returned	Remarks
(car '(a b c))	a	Car takes one argument, a list, and returns the first element.
(cdr '(a b c))	(b c)	Cdr takes one argument, a list, and returns a list with the first element removed.
(cons 'a '(b c))	(a b c)	Cons takes two arguments, an element, and a list and returns a list with the element inserted at the beginning.
(list 'a '(b c))	(a (b c))	List take any number of arguments and returns a list with the arguments as elements.

arguments, an element and a list. It constructs a new list by making the element the first member of the list. List takes any number of arguments, and makes them into a list, with each argument a top member.

Note the quotation marks preceding the arguments in the function calls of Table 3.2. As pointed out above, an error will result if they are not there because the interpreter will try to evaluate each argument before evaluating the function. Notice the difference in results with and without the quotation marks.

```
->(cons '(* 2 3) '(1))       ;the literal list (* 2 3) is
((* 2 3) 1)                  ;"consed" to the list (1)
->(cons (* 2 3) '(1))        ;the evaluated list (* 2 3) is
(6 1)                        ;consed to the list (1)
->(setq x '(a b c))          ;x is bound to the literal list
(A B C)                      ;(a b c)
->'x                         ;the quote ' inhibits evaluation
X                            ;of x
->x                          ;but unquoted x evaluates to its
(A B C)                      ;previously bound value
```

The syntax for a function call is

```
(function-name arg1 arg2 ...)
```

where any number of arguments may be used. When a function is called, the arguments are first evaluated from left to right (unless within quotation marks) and then the function is executed using the evaluated argument values. Complete the list manipulation examples below.

```
->(car (cdr '(a b c d)))           ;extracts the second element
B
->(cdr car '((a b) c d))           ;extracts the list (b)
(B)
->(cons 'one '(two three))         ;inserts the element one in
(ONE TWO THREE)                    ;the list (two three)
->(cons (car '(a b c)              ;lists may continue on
   (cdr '(a b c)))                 ;several lines, but parens
(A B C)                            ;must always balance!
->(list '(a b) 'c 'd))             ;makes a list of the top
((A B) C D)                        ;elements
->
```

Sequences of car and cdr functions may be abbreviated by concatenating the letter *a* for car and *d* for cdr within the letters *c*, and *r*. For example, to extract c from the list

$$(a\ (b\ c)\ d)$$

we write cadadr to abbreviate the sequence car cdr car cdr. Thus

```
->(cadadr '(a (b c) d))
C
->
```

Other useful list manipulation functions are append, last, member, and reverse. Append merges arguments of one or more lists into a single list. Last takes one argument, a list, and returns a list containing the last element. Member takes two arguments, the second of which must be a list. If the first argument is a member of the second one, the remainder of the second list is returned beginning with the member element. Reverse takes a list as its argument and returns a list with the top elements in reverse order from the input list. Table 3.3 summarizes these operations.

TABLE 3.3 ADDITIONAL LIST MANIPULATION FUNCTIONS

Function call	Value returned	Remarks
(append '(a) '(b c))	(a b c)	merges two or more lists into a single list.
(last '(a b c d))	(d)	returns a list containing the last element.
(member 'b '(a b d))	(b d)	returns remainder of second argument list starting with element matching first argument.
(reverse '(a (b c) d))	(d (b c) a)	returns list with top elements in reverse order.

Complete the practice examples below.

```
->(append '(a (b c)) '(d e))        ;returns a single list of
(A (B C) D E)                       ;top element input lists
->(append '(a) '(b c) '(d))
(A B C D)
->(last '(a b (c d) (e)))           ;returns the last top
((E))                               ;element as a list
->(member '(d) '(a (d) e f))        ;returns the tail of
((D) E F)                           ;list from member element
->(reverse '(a b (c d) e))          ;returns the list with top
(E (C D) B A)                       ;elements in reverse order
```

3.3 DEFINING FUNCTIONS, PREDICATES, AND CONDITIONALS

Defining Functions

Now that we know how to call functions, we should learn how to define our own. The function named defun is used to define functions. It requires three arguments: (1) the new function name, (2) the parameters for the function, and (3) the function body or LISP code which performs the desired function operations. The format is

```
(defun name (parm1 parm2 ...) body).
```

Defun does not evaluate its arguments. It simply builds a function which may be called like any other function we have seen. As an example, we define a function named averagethree to compute the average of three numbers.

```
->(defun averagethree (n1 n2 n3)
     (/ (+ n1 n2 n3) 3))
AVERAGETHREE
->
```

Note that defun returned the name of the function. To call averagethree, we give the function name followed by the actual arguments

```
->(averagethree 10 20 30)
20
->
```

When a function is called, the arguments supplied in the call are evaluated unless they are in quotation marks and bound to (assigned to) the function parameters. The argument values are bound to the parameters in the same order they were given in the definition. The parameters are actually dummy variables (n_1, n_2, n_3 in averagethree) used to make it possible to give a function a general definition.

Predicate Functions

Predicates are functions that test their arguments for some specific condition. Except for the predicate "member" (defined above), predicates return true (t) or false (nil), depending on the arguments. The most common predicates are

atom	>=
equal	listp
evenp	null
greaterp (or >)	numberp
<=	oddp
lessp (or <)	zerop

The predicate "atom" takes one argument. It returns t if the argument is an atom and nil otherwise. Equal takes two arguments and returns t if they evaluate to the same value, and nil otherwise. Evenp, numberp, oddp, and zerop are tests on a single numeric argument. They return t if their argument evaluates to an even number, a number, an odd number, or zero respectively. Otherwise they each return nil.

Greaterp and lessp each take one or more arguments. If there is only one argument, each returns t. If more than one argument is used, greaterp returns t if the arguments, from left to right, are successively larger; otherwise nil is returned. Lessp requires that the arguments be successively smaller from left to right to return t. Otherwise it returns nil. The predicates >= and <= have the same meaning as greaterp and lessp respectively, except they return t if successive elements are also equal. Finally listp and null both take a single argument. Listp returns t if its argument evaluates to a list, and nil otherwise. Null returns t if its argument evaluates to nil; otherwise it returns nil. Examples for calls using each of these predicates are given in Table 3.4.

TABLE 3.4 THE MOST COMMON PREDICATE CALLS

Function call	Value returned	Remarks
(atom 'aabb)	t	aabb is a valid atom
(equal 'a (car '(a b))	t	a equals a, but note that (equal 1 1.0) returns nil
(evenp 3)	nil	3 is not an even number
(numberp 10ab)	nil	10ab is not a number
(oddp 3)	t	3 is an odd number
(zerop .000001)	nil	argument is not zero
(greaterp 2 4 27)	t	arguments are succeedingly larger, from left to right
(lessp 5 3 1 2)	nil	arguments are not successively smaller, left to right
(listp '(a))	t	(a) is a valid list
(null nil)	t	nil is an empty list

The Conditional Cond

Predicates are one way to make tests in programs and take different actions based on the outcome of the test. However, to make use of the predicates, we need some construct to permit branching. Cond (for conditonal) is like the if..then..else construct.

The syntax for cond is

$$
\begin{array}{ll}
\texttt{(cond} & \texttt{(<test}_1\texttt{> <action}_1\texttt{>)} \\
 & \texttt{(<test}_2\texttt{> <action}_2\texttt{>)} \\
 & \quad . \quad . \\
 & \quad . \quad . \\
 & \quad . \quad . \\
 & \texttt{(<test}_k\texttt{> <action}_k\texttt{>))}
\end{array}
$$

Each (<test$_i$> <action$_i$>), i=1, ... k, is called a clause. Each clause consists of a test portion and an action or result portion. The first clause following the cond is executed by evaluating <test$_1$>. If this evaluates to non nil, the <action$_1$> portion is evaluated, its value is returned, and the remaining clauses are skipped over. If <test$_1$> evaluates to nil, control passes to the second clause without evaluating <action$_1$> and the procedure is repeated. If all tests evaluate to nil, cond returns nil.

We illustrate the use of cond in the following function maximum2 which returns the maximum of two numbers.

```
->(defun maximum2 (a b)
    (cond ((> a b) a)
          (t b)))
MAXIMUM2
->
```

When maximum2 is executed, it starts with the first clause following the cond. The test sequence is as follows: if (the argument bound to) a is greater than (that bound to) b, return a, else return b. Note the t in the second clause preceding b. This forces the last clause to be evaluated when the first clause is not.

```
->(maximum2 234 320)
320
->
```

A slightly more challenging use of cond finds the maximum of three numbers in the function maximum3.

```
->(defun maximum3 (a b c)
    (cond ((> a b) (cond ((> a c) a)
                         (t c)))
          ((> b c) b)
          (t c)))
->MAXIMUM3
```

Trying maximum3 we have

```
->(maximum3 20 30 25)
30
->
```

Common LISP also provides a form of the more conventional if..then..else conditional. It has the form

```
(if test <then-action> <else-action>)
```

For this form, test is first evaluated. If it evaluates to non nil, the <then-action> is evaluated and the result returned; otherwise the <else-action> is evaluated and its value returned. The <else-action> is optional. If omitted, then when test evaluates to nil, the if function returns nil.

Logical Functions

Like predicates, logical functions may also be used for flow of control. The basic logical operations are and, or, and not. Not is the simplest. It takes one argument and returns t if the argument evaluates to nil; it returns nil if its argument evaluates to non-nil. The functions and and or both take any number of arguments. For both, the arguments are evaluated from left to right. In the case of and, if all arguments evaluate to non-nil, the value of the last argument is returned; otherwise nil is returned. The arguments of or are evaluated until one evaluates to non-nil, in which case it returns the argument value; otherwise it returns nil.

Some examples of the operators and, or and not in expressions are

```
->(setq x '(a b c))
(A B C)
->(not (atom x))
T
->(not (listp x))
NIL
 -(or (member 'e x) (member 'b x))
(B C)
->(or (equal 'c (car x)) (equal 'b (car x)))
NIL
->(and (listp x) (equal 'c (caddr x)))
C
->(or (and (atom x) (equal 'a x))
     (and (not (atom x)) (atom (car x))))
T
->
```

3.4 INPUT, OUTPUT, AND LOCAL VARIABLES

Without knowing how to instruct our programs to call for inputs and print messages or text on the monitor or a printer, our programs will be severely limited. The operations we need for this are performed with the input-output (I/O) functions. The most commonly used I/O functions are read, print, prinl, princ, terpri, and format.

Read takes no arguments. When read appears in a procedure, processing halts until a single s-expression is entered from the keyboard. The s-expression is then returned as the value of read and processing continues. For example, if we include a read in an arithmetic expression, an appropriate value should be entered when the interpreter halts.

```
->(+ 5 (read))
6
11
->
```

When the interpreter looked for the second argument for +, it found the read statement which caused it to halt and wait for an input from the keyboard. If we enter 6 as indicated, read returns this value, processing continues, and the sum 11 is then returned.

Print takes one argument. It prints the argument as it is received, and then returns the argument. This makes it possible to print something and also pass the same thing on to another function as an argument. When print is used to print an expression, its argument is preceded by the carriage-return and line-feed characters (to start a new line) and is followed by a space. In the following example, note the double printing. This occurs because print first prints its argument and then returns it, causing it to be printed by the read-evaluate-print loop.

```
->(print '(a b c))
(A B C)
(A B C)
->(print "hello there")
"hello there"
"hello there"
```

Notice that print even prints the double quotation marks defining the string.

Prinl is the same as print except that the new-line characters and space are not provided (this is not true for all implementations of Common LISP).

```
->((prin1 '(hello)) (prin1 '(hello)))
(HELLO)(HELLO)
->
```

We can avoid the double quotation marks in the output by using the printing function princ. It is the same as prinl except it does not print the unwanted quotation marks. For example, we use princ to print the following without the marks.

```
->(princ "hello there")
hello there "hello there"
->
```

Princ eliminated the quotes, but the echo still remains. Again, that is because princ returned its argument (in a form that LISP could read), and, since that was the last thing returned, it was printed by the read-evaluate-print loop. In a typical program, the returned value would not be printed on the screen as it would be absorbed (used) by another function.

The primitive function terpri takes no arguments. It introduces a new-line (carriage return and line feed) wherever it appears and then returns nil. Below is a program to compute the area of a circle which uses several I/O functions, including user prompts.

```
->(defun circle-area ()
     (terpri)
     (princ "Please enter the radius: ")
     (setq radius (read))
     (princ "The area of the circle is: ")
     (princ (* 3.1416 radius radius))
     (terpri))
CIRCLE-AREA
->(circle-area)
Please enter the radius: 4
The area of the circle is: 50.2656
->
```

Notice that princ permits us to print multiple items on the same line and to introduce a new-line sequence we use terpri.

The format function permits us to create cleaner output than is possible with just the basic printing functions. It has the form (format <destination> <string> arg1 arg2 . . .): Destination specifies where the output is to be directed, like to the monitor or some other external file. For our purposes, destination will always be *t* to signify the default output, the monitor. String is the desired output string, but intermixed with format directives which specify how each argument is to be represented. Directives appear in the string in the same order the arguments are to be printed. Each directive is preceded with a tilde character (~) to identify it as a directive. We list only the most common directives below.

~A The argument is printed as though princ were used.

~S The argument is printed as though prinl were used.

~D The argument which must be an integer is printed as a decimal number.

~F The argument which must be a floating-point number is printed as a decimal floating-point number.

~C The argument is printed as character output.

~% A new-line is printed.

The field widths for appropriate argument values are specified with an integer immediatly following the tilde symbol; for example, ~5D specifies an integer field of width 5. As an example of format, suppose *x* and *y* have been bound to floating-point numbers 3.0 and 9.42 respectively. Using format, these numbers can be embedded within a string of text.

```
->(format t "Circle radius = ~2F~%Circle area = ~3F" x y)
"Circle radius = 3.0
Circle area = 9.42"
->
```

Constructs for Local Variables

Frequently, it is desirable to designate local variables (variables which are defined only within a given procedure) rather than the global assignments which result from setq. Of course parameters named as arguments in a function definition are local. The values they are assigned within the function are accessible only within that function. For example, consider the variables *x* and *y* in the following:

```
->(setq y '(a b c))
(A B C)
->(setq x '(d e f))
(D E F)
->(defun local-var (x)
     (setq y (cons x y)))
LOCAL-VAR
->(local-var 6)
(6 A B C)
->x
(D E F)
->y
(A B C)
->
```

The variable *x* in the defun is local in scope. It reverts back to its previous value after the function local-var is exited. The variable *y*, on the other hand, is global. It is accessible from any procedure and retains its value unless reset with setq.

The let and prog constructs also permit the creation of local variables. The syntax for the let function is

(let ((var_1 val_1) (var_2 val_2)...) <s-expressions>)

where each var_i is a different variable name and val_i is an initial value assigned to each var_i, respectively. When let is executed, each val_i is evaluated and assigned to the corresponding var_i, and the s-expressions which follow are then evaluated in order. The value of the last expression evaluated is then returned. If an initial value is not included with a var_i, it is assigned nil, and the parentheses enclosing it may be omitted.

```
-> (let ((x 'a)
        (y 'b)
        (z 'c))
     (cons x (cons y (list z))))
(A B C)
->
```

The prog function is similar to let in that the first arguments following it are a list of local variables where each element is either a variable name or a list containing a variable name and its initial value. This is followed by the body of the prog, and any number of s-expressions.

Prog executes list s-expressions in sequence and returns nil unless it encounters a function call named return. In this case the single argument of return is evaluated and returned. Prog also permits the use of unconditional go statements and labels (atom labels) to identify the go-to transfer locations. With the go and label statements, prog permits the writing of unstructured programs and, therefore, is not recommended for general use. An example of a function like memb (member) which uses iteration, will illustrate this use. The main function memb requires two arguments, an element and a list.

```
-> (defun memb (el 1st)
     (prog ()
       start
       (cond ((equal el (car 1st)) (return 1st)))
       (setq 1st (cdr 1st))
       (go start)))
MEMB
->
```

Note that prog used here requires no local variables. Also note the label start, the transfer (loop back) point for the go statement. The second clause of the cond executes when the first clause is skipped because setq is non-nil!

3.5 ITERATION AND RECURSION

Iteration Constructs

We saw one way to perform iteration using the prog construct in the previous section. In this section, we introduce a structured form of iteration with the do construct, which is somewhat like the while loop in Pascal.

The do statement has the form

```
(do(<var1 val1> <var-update1>)
   (<var2 val2> <var-update2>)
       .
       .
       .
   (<test> <return-value>)
   (<s-expressions>))
```

The val$_i$ are initial values which are all evaluated and then bound to the corresponding variables var$_i$ in parallel. Following each such statement are optional update statements which define how the var$_i$ are to be updated with each iteration. After the variables are updated during an iteration, the test is evaluated, and if it returns non-nil (true), the return-value is evaluated and returned. The s-expressions forming the body of the construct are optional. If present, they are executed each iteration until an exit test condition is encountered. An example of the factorial function will illustrate the do construct.

```
->(defun factorial (n)
    (do ((count n (- count 1))
         (product n (* product (- count 1))
         ((equal 0 count) product)))
  FACTORIAL
```

In this definition there is no need to include a body. All operations required to compute the factorial of n are contained in the initial values and the update procedures.

There is also a do* construct which is the same as do, except the assignment of initial values is made to each var$_i$ sequentially before the next form is evaluated.

In addition to the do and prog constructs for iteration, one may use a loop function. Loop has the simple form

```
(loop <s-expressions>)
```

where the s-expressions are evaluated repeatedly until a call to a return is encountered.

Of course, let and other functions can be embedded within the loop construct if local variables are needed.

Recursion

For many problems, recursion is the natural method of solution. Such problems occur frequently in mathematical logic, and the use of recursion will often result in programs which are both elegant and simple. A recursive function is one which calls itself successively to reduce a problem to a sequence of simpler steps. Recursion requires a stopping condition and a recursive step.

We illustrate with a recursive version of factorial. The recursive step in factorial is the product of *n* and factorial(*n*-1). The stopping condition is reached when *n* = 0.

```
->(defun factorial (n)
    (cond ((zerop n) 1)
          (t (* n (factorial (- n 1))))))
FACTORIAL
->(factorial 6)
720
```

Note the stopping condition on the second line of the function definition, and the recursive step on the last line.

We present another example of recursion which defines the member function called newmember.

```
->(defun newmember (el 1st)
    (cond ((null 1st) nil)
          ((equal el (car 1st)) 1st)
          ((t (newmember el (cdr 1st)))))
NEWMEMBER
->
```

If the atom *c* and list (*a b c d*) are given as the arguments in the call to newmember, *c* gets bound to el and (*a b c d*) is bound to *lst*. With these bindings, the first cond test fails, since *lst* is not null. Consequently, the second test is executed. This also fails since *el*, bound to *c*, does not equal the car of *lst* which is *a*. The last test of the cond construct is forced to succeed because of the t test. This initiates a recursive call to newmember with the new arguments *el* (still bound to *c*) and the cdr of *lst* which is (*b c d*). Again, a match fails during the cond tests; so another recursive call is made, this time with arguments *el* (still bound to *c*) and *lst* now bound to (*c d*). When this call is executed, a match is found in the second cond test so the value of *lst* (*c d*) is returned.

3.6 PROPERTY LISTS AND ARRAYS

Property Lists

One of the unique and most useful features of LISP as an AI language is the ability to assign properties to atoms. For example, any object, say an atom which represents a person, can be given a number of properties which in some way characterize the person, such as height, weight, sex, color of eyes and hair, address, profession, family members, and so on. Property list functions permit one to assign such properties to an atom, and to retrieve, replace, or remove them as required.

The function putprop assigns properties to an atom. It takes three arguments: an object name (an atom), a property or attribute name, and property or attribute value. For example, to assign properties to a car, we can assign properties such as make, year, color, and style with the following statements:

```
->(putprop 'car 'ford 'make)
FORD
->(putprop 'car 1988 'year)
1988
->(putprop 'car 'red 'color)
RED
->(putprop 'car 'four-door 'style)
FOUR-DOOR
->
```

As you can see, the form of putprop is

```
(putprop object value attribute)
```

where value is returned. The object, car, will retain these properties until they are replaced with new ones or until removed with the remprop function which takes two arguments, the object and its attribute. In other words, properties are global assignments. To retrieve a property value, such as the color of car, we use the function get, which also takes the two arguments object and attribute.

```
->(get 'car 'color)
RED
->(get 'car 'make)
FORD
->(putprop 'car 'blue 'color)
BLUE
->(get 'car 'color)
BLUE
->(remprop 'car 'color)
BLUE
->(get 'car 'color)
NIL
->
```

The property value may be an atom or a list. For example, if Danny has pets named Schultz, Penny, and Etoile, they can be assigned as

```
->(putprop 'danny '(schultz penny etoile) 'pets)
(SCHULTZ PENNY ETOILE)
->(get 'danny 'pets)
(SCHULTZ PENNY ETOILE)
->
```

To add a new pet named Heidi without knowing the existing pets one can do the following:

```
->(putprop 'danny (cons 'heidi (get 'danny 'pets)) 'pets)
(HEIDI SCHULTZ PENNY ETOILE)
->
```

Items can be removed from a list of values in a similar manner.

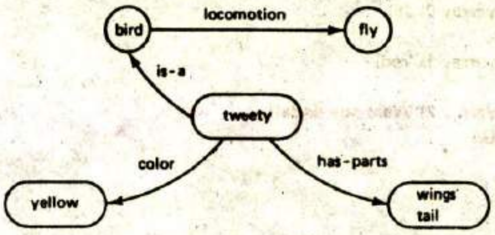
Since some versions of Common LISP do not provide the putprop function, it may be necessary to define your own. This can be done with the following code.

```
->(defun putprop (object value property)
    (setf (get object property) value))
PUTPROP
->
```

The new function setf used in the above definition is like setq except it is more general. It is an assignment function which also takes two arguments, the first of which may be either an atom or an access function (like car, cdr, and get) and the second, the value to be assigned. When the first argument is an atom, setf behaves the same as setq. It simply binds the evaluated second argument to the first. When the first argument is an access function, setf places the second argument, the value, at the location accessed by the access function. For example, if (*a b c*) has been bound to *x*, the expression (setf (car x) 'd) will replace the *a* in (*a b c*) with *d*. Likewise, setf can be used directly to assign or replace a property value.

```
->(setf (get 'car 'color) 'pink)
PINK
->(get 'car 'color)
PINK
->
```

As we shall see in later chapters, property lists provide us with a convenient mechanism with which to represent knowledge. One such representation is the conceptual network where objects, their attributes, and relations to other objects are easily



```
(putprop 'bird 'fly 'locomotion)
(putprop 'tweety 'bird 'is-a)
(putprop 'tweety '(wings tail) 'has-parts)
(putprop 'tweety 'yellow 'color)
```

Figure 3.1 Representation of facts and relations as a network.

expressed. In Figure 3.1 some facts about Tweety, the famous AI bird, have been represented as a network using property lists.

Arrays

Single- or multiple-dimension arrays may be defined in LISP using the make-array function. The items stored in the array may be any LISP object. For example, to create an array with ten cells named myarray, we bind the unquoted name to an array using setf (or setq) with the make-array function and a specification of the number of cells.

```
->(setf myarray (make-array '(10)))
#A(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)
->
```

Note that the function returns the pound sign (#) followed by an A and the array representation with its cells initially set to nil.

To access the contents of cells, we use the function aref which takes two arguments, the name of the array and the index value. Since the cells are indexed starting at zero, an index value of 9 must be used to retrieve the contents of the tenth cell.

```
->(aref myarray 9)
NIL
->
```

To store items in the array, we use the function setf as we did above to store properties on a property list. So, to store the items 25, red, and (sam sue linda) in the first, second, and third cells of myarray, we write

```
->(setf (aref myarray 0) 25)
25
->(setf (aref myarray 1) 'red)
RED
->(setf (aref myarray 2) '(sam sue linda))
(SAM SUE LINDA)
->
```

3.7 MISCELLANEOUS TOPICS

We complete our presentation of LISP in this section with a few additional topics, including the functions mapcar, eval, lambda, trace and untrace, and a brief description of the internal representation of atoms and lists.

Mapping Functions

Mapcar is one of several mapping functions provided in LISP to apply some function successively to one or more lists of elements. The first argument of mapcar is a function, and the remaining argument(s) are lists of elements to which the named function is applied. The results of applying the function to successive members of the lists are placed in a new list which is returned. For example, suppose we wish to add 1 to each element of the list (5 10 15 20 25). We can do this quite simply with mapcar and the function 1+.

```
->(mapcar '1+ '(5 10 15 20 25))
(6 11 16 21 26)
->
```

If we wish to add the corresponding elements of two lists (even of unequal length), say (1 2 3 4 5 6) and (1 2 3 4), we use the + function with the lists to obtain the sum of the first four elements.

```
->(mapcar '+ '(1 2 3 4 5 6) '(1 2 3 4))
(2 4 6 8)
->
```

It should be clear that mapcar can be used in a variety of ways in lieu of iterative functions. And, the function being applied to the lists may be either user defined or built-in.

Lambda Functions

When a function is defined with defun, its name and address must be stored in a symbol table for retrieval whenever it is called in a program. Sometimes, however, it is desirable to use a function only once in a program. This will be the case when it is used in a mapping operation such as with mapcar, which must take a procedure as its first argument. LISP provides a method of writing unnamed or anonymous functions that are evaluated only when they are encountered in a program. Such functions are called lambda functions. They have the following form

```
(lambda (arguments) <function-body>)
```

We illustrate the use of a lambda function to compute the cubed value of a list of numbers. This will be accomplished by using mapcar to apply a lambda cube function to a list of numbers. When a function is called by another function, it should be preceded by the characters #' to indicate that the following item is a function. This is equivalent to preceding the function with a single quotation mark (') or the function named function in some LISP dialects. The lambda function we need to find the cube of a single number is just (lambda (x) (* x x x)). We use this with mapcar now to find the cubes of the numbers (1 2 3 4)

```
->(defun cube-list (lst)
     (mapcar #'(lambda (x) (* x x x)) lst))
CUBE-LIST
->(cubelist (1 2 3 4))
(1 8 27 64)
->
```

Internal Storage

As we have seen, lists are flexible data structures that can shrink or grow almost without limit. This is made possible through the use of linked cell structures in memory to represent lists. These can be visualized as storage boxes having two components which correspond to the car and cdr of a list. The cells are called cons-cells, because they are constructed with the cons function, where the left component points to the first element of a list (the car of the list) and the right component points to the remainder of the list (the cdr of the list). An example of the representation for the list (a (b c (d)) e f) is given in Figure 3.2.

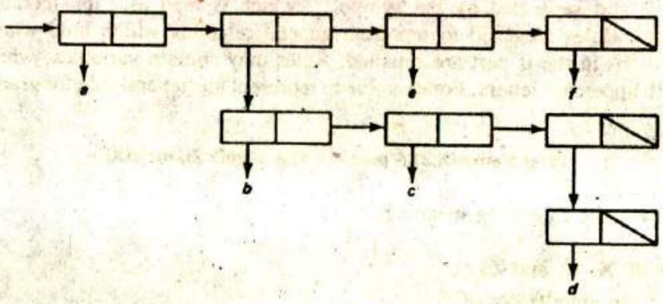


Figure 3.2 Representation for the list (*a* (*b c* (*d*)) *e f*).

The boxes with the slash in the figure represent nil. When cons is used to construct a list, the cons-cells are created with pointers to the appropriate elements as depicted in Figure 3.2. The use of such structures permits lists to be easily extended or modified.

3.8 PROLOG AND OTHER AI PROGRAMMING LANGUAGES

PROLOG (for *PRO*gramming in *LOG*ic) was invented by Alain Colmerauer and his associates at the University of Marseilles during the early 1970s. PROLOG uses the syntax of predicate logic to perform symbolic, logical computations. It has a number of built-in features (particularly control features) that limit its flexibility but simplify many aspects of programming.

Programming in PROLOG is accomplished by creating a data base of facts and rules about objects, their properties, and their relationships to other objects. Queries can then be posed about the objects and valid conclusions will be determined and returned by the program. Responses to user queries are determined through a form of inferencing control known as resolution. This process is described in the next chapter.

Facts in PROLOG are declared with predicates and constants written in lowercase letters. The arguments of predicates are enclosed in parentheses and separated with commas. For example, some facts about family relationships could be written as

```
sister(sue,bill)
parent(ann,sam)
parent(joe,ann)
male(joe)
female(ann)
```

The first fact is the predicate sister with arguments sue and bill. This predicate has the intended meaning that Sue is the sister of Bill. Likewise, the next predicate has the meaning that Ann is the parent of Sam, and so on.

Rules in PROLOG are composed of a condition or "if" part, and a conclusion or "then" part separated by the symbol :- which is read as "if" (conclusion if conditions). Rules are used to represent general relations which hold when all of the conditions in the if part are satisfied. Rules may contain variables, which must begin with uppercase letters. For example to represent the general rule for grandfather, we write

```
grandfather(X,Z) :- parent(X,Y), parent(Y,Z), male(X)
```

This rule has the following meaning:

For all X, Y, and Z,
X is the grandfather of Z

IF X is the parent of Y, and
Y is the parent of Z, and
X is a male

Note that separate conditions in the rule are separated by commas. The commas act as conjunctions, that is, like and statements where all conditions in the right-hand side must be satisfied for the rule to be true.

Given a data base of facts and rules such as that above, we may make queries by typing after the query symbol -? statements such as

```
?- parent(X,sam)
X=ann
?- male(joe)
yes
?-grandfather(X,Y)
X=joe, Y=sam
?-female(joe)
no
```

Note that responses to the queries are given by returning the value a variable can take to satisfy the query or simply with yes (true) or no (false).

Queries such as these set up a sequence of one or more goals that are to be satisfied. A goal is satisfied if it can be shown to logically follow from the facts and rules in the data base. This means that a proper match must be found between predicates in the query and the database and that all subgoals must be satisfied through consistent substitutions of constants and variables for variable arguments. To determine if a consistent match is possible, PROLOG searches the data base and tries to make substitutions until a permissible match is found or failure occurs. For example, when the query

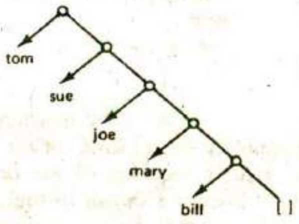
```
?-grandfather(X,sam)
```

is given, a search is made until the grandfather predicate is found in the data base. In this case, it is the head of the above grandfather rule. The constant Sam is substituted for Z, and an attempt is then made to satisfy the body of the rule. This requires that the three conditions (subgoals) in the body are satisfied. Attempts to satisfy these conditions are made from left to right in the body by searching and finding matching predicates and making consistent variable substitutions by (1) substituting Joe for X in the first subgoal parent(X,Y) and later in the third subgoal male(X), (2) substituting Sam for Z in the second subgoal parent(Y,Z), and (3) substituting Ann for Y in the first two subgoals parent(X,Y) and parent(Y,Z). Since consistent variable substitutions can be made in this case, PROLOG returns X = Joe, in response to the unknown X in the query. PROLOG will continue searching through the database until a consistent set of substitutions is found. If all substitutions cannot be found, failure is reported with the printout no.

Lists in PROLOG are similar to list data structures in LISP. A PROLOG list is written as a sequence of items separated by commas, and enclosed in square brackets. For example, a list of the students Tom, Sue, Joe, Mary, and Bill is written as

```
[tom,sue,joe,mary,bill]
```

A list is either empty [] or nonempty. If nonempty, a list is considered as an object with a head and a tail, corresponding to the car and cdr of a list in LISP. Thus, the head of the above list is Tom, and the tail is the remaining sublist [sue,joe,mary,bill]. Since the tail is a list, it too is either empty or it has a head and tail. This suggests a binary tree representation similar to linked LISP cons cells.



PROLOG provides a notation to separate the head and tail, the vertical bar |, as in [Head|Tail]. This permits one to define the Head of a list as any number of items followed by a | and the list of remaining items. Thus, the list [a,b,c,d] may be written as $[a,b,c,d]=[a|[b,c,d]]=[a,b|[c,d]]=[a,b,c,d|[]]$.

Matching with lists is accomplished as follows:

```
?-[Head|Tail] = [tom,sue,joe,mary].
Head = tom
Tail = [sue,joe,mary]
```

A number of list manipulation predicates are available in most PROLOG implementations, including append, member, conc (concatenate), add, delete, and so on. One can also define special predicates as needed from the basic definitions included. For example, a definition of the member function member(*X*,*L*), where *X* is an item and *L* is a list, can be written as a fact and a rule.

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

member(X,[Head|Tail]) :-
    member(X,Tail).
```

The first condition states that *X* is a member of the list *L* if *X* is the head of *L*. Otherwise, the rule states that *X* is a member of *L* if *X* is a member of the tail of *L*. Thus,

```
?- member(c,[a,b,c,d])
yes
?- member(b,[a,[b,c],d])
no
```

PROLOG has numeric functions and relations, as well as list handling capabilities, which give it some similarity to LISP. In subsequent chapters, we will see examples of some PROLOG programs. For more details on the syntax, predicates, and other features of PROLOG the reader is referred to the two texts, Bratko (1986), and Clocksin and Mellish (1981).

Other programming languages used in AI include C, object oriented extensions to LISP such as Flavors, and languages like Smalltalk. The language C has been used by some practictioners in AI because of its popularity and its portability. Object oriented languages, although they have been introduced only recently, have been gaining much popularity. We discuss these languages in Chapter 8.

3.9 SUMMARY

In this chapter we introduced LISP, the programming language of AI. We defined LISP syntax and examined a number of built-in functions, including the numeric functions (+, -, *, /), and the list manipulation functions (car, cdr, cons, list, append, last, member, and reverse). We also defined a number of predicate functions such as equal, null, numberp, atom, listp, and so on. We then described how to define our own functions using defun. We saw how to use the conditional cond and the logical functions and, or, and not.

The input-output functions print, prinl, princ, read, terpri, and format were defined and example programs presented. We saw how to write iterative programs with the do, loop, and prog constructs. We saw how to define and use local variables with the let and prog functions. We discussed recursion, and saw examples of its use. We then looked at property lists, a valuable method with which to assign multiple values or properties to variable atoms. Array definitions and examples were briefly introduced, and a few miscellaneous topics such as mapping functions (mapcar), the lambda form, and internal list representations concluded our treatment of LISP.

Finally, we introduced the logic programming language PROLOG, and gave some simple examples of its use. We concluded the chapter with a few comments regarding other AI languages and noted that object oriented languages are described in detail in Chapter 8.

EXERCISES

3.1 Write a LISP program to convert centigrade temperatures to Fahrenheit.

3.2 Define a function called first-element that takes a list as its argument and returns the first top element of the list.

3.3 Define a function called number-of-elements that takes a list as its only argument and returns the number of top elements in the list.

3.4 Define a function called rotate that takes a list and rotates the elements by one position as in

```
(rotate '(a b c d)) returns (D A B C).
```

3.5 Define a function newlist that takes one argument and returns it as a list. If the argument is already a list, including the empty list, newlist returns it without change. If the argument is an atom, it returns it as a list.

3.6 Define a function named addlist that takes two arguments, an item and a list. If the item is in the list, the function returns the list unaltered. If the item is not in the list, the function returns the list with the item entered as the first element. For example,

```
(addlist 'c '(a b c d)) returns (a b c d) but
(addlist 'e '(a b c)) returns (e a b c).
```

3.7 Define a function construct-sentence which takes two lists as arguments. The lists are simple sentences such as

```
(the dog barked) or
(the dog chased the car).
```

The function should check to see if the subject of the sentence is the same in both sentences. If so, the function should return the compound sentence

```
(the dog barked and chased the car).
```

If the two sentences do not have the same subject, the function should return nil.

3.8 Write a function called word-member which takes one list as argument. The function should print a prompt to the user

```
(please type any word)
```

If the word typed is a member of the list the function should return *t*, otherwise nil. For example,

```
> (word-member '(the brown fox ran))
(please type any word) fox
T
```

3.9 Write a function talk that takes no arguments. It prints the prompt (without quotation marks or parentheses)

```
What is your name?
```

The function should then read an input from the same line two spaces following the question mark (e.g., Susan), issue a line feed and carriage return, and then print

```
Hello, Susan. What is your best friend's name?
```

As before, the user should type a name (e.g., Joe). The program should then respond with

```
Very interesting; Joe is my best friend too!
```

3.10 Write an iterative function named nth-item that takes two arguments, a positive integer and a list. The function returns the item in the nth position in the list. If the integer exceeds the number of elements in the list, it returns nil. For example,

```
(nth-item 3 '(a b c d e f)) returns c.
```

3.11 Write a recursive function named power that takes two numeric arguments, n and m. The function computes the nth power of m (m^n). Be sure to account for the the case where $n = 0$, that is $m^0 = 1$. For example,

(power 4 3) returns $4^3 = 64$.

3.12 Define a function called intersection which takes two lists as arguments. The function should return a list containing single occurrences of all elements which appear in both input lists. For example,

```
(intersection '(a b e g k l) '(a c e g x y ))
```

should return (a e g).

3.13 Write a new function called new-reverse that takes a list as argument. The function should return the list in reversed order. Do not use the LISP function reverse to define new-reverse. For example,

```
(new-reverse '(a b (f e) l)) should return
(l (f e) b a).
```

3.14 Write an iterative function named sum-all using do that takes an integer n as argument and returns the sum of the integers from 1 to n. For example,

```
(sum-all 5) should return 15.
```

3.15 Write a function called sum-squares which uses mapcar to find the sum of the squares of a list of integers. The function takes a single list as its argument. Write a lambda function which mapcar uses to find the square of the integers in the list. For example,

```
(sum-squares (2 3 1 4)) should return 30.
```

3.16 Write a PROLOG program that answers questions about famlly members and relationships. Include predicates and rules which define sister, brother, father, mother, grandchild, grandfather, and uncle. The program should be able to answer queries such as the following:

```
?- father(X, bob).
?- grandson(X, Y).
?- uncle(bill, sue).
?- mother(mary, X).
```

3.17 Trace the search sequence PROLOG follows in satisfying the following goal:

```
?- member(c,[a,b,c,d]).
```

3.18 Write a function called match that takes two arguments, a pattern and a clause. If the first argument, the pattern, is a variable identified by a question mark followed by a lowercase letter (like ?x or ?y), the function should return a list giving the variable and the corresponding clause, since a variable matches anything. If the pattern is not a variable, it should return *t* only if the pattern and the clause are identical. Otherwise, the function should return nil.