CHAPTER 1
Introduction to Expert Systems

1.1 INTRODUCTION

This chapter is a broad introduction to expert systems. The fundamental principles of expert systems are introduced. The advantages and disadvantages of expert systems are discussed and the appropriate areas of application for expert systems are described. The relationship of expert systems to other methods of programming are also discussed.

1.2 WHAT IS AN EXPERT SYSTEM?

During the 20th Century, a number of definitions of **artificial intelligence** (AI) were proposed. One of the earliest popular definitions of AI was, and still is: "making computers think like people," as evident by the large number of science fiction movies that promote this view. Actually this definition has its roots in the British mathematician and computer pioneer Alan Turing's famous Turing Test in which a human would try to determine if the "person" they were talking to via a remote keyboard was a human or computer program. Passing such a test is considered to be **strong AI**. The term *strong AI* is promoted by the people who believe AI should be based on a strong logical foundation rather than what they call the **weak AI** based on artificial neural networks, genetic algorithms, and evolutionary methods. Today it is evident that no one technique of AI can successfully deal with all problems; a combination of methods works best.

The first program to pass the Turing Test was written as an experiment in psychology by Steven Weizenbaum in 1967; since then the knowledge and interaction with people has greatly increased and a $100,000 competition called the **Loebner Prize** is held to see which is best (http://www.loebner.net/Prizef/loebner-prize.html). Of course, today communication is often by speech recognition rather than the old-style teletype or keyboard. So if you are ever frustrated thinking you're talking to a person on the phone and they just don't understand what you're saying, ask if they've passed the Turing test.

Expert systems were developed as research tools in the 1960s as a special type of AI to successfully deal with complex problems in a narrow domain such as medical disease diagnosis. The classic problem of building a general-purpose AI program that can solve any problem has been too difficult without specific knowledge of the problem domain, i.e., medical disease diagnosis. Expert systems have greatly increased in popularity since their commercial introduction in the early 1980s. Today, expert systems are used in business, science, engineering, manufacturing, and many other fields in which there exists a well-defined problem domain. In fact if you are selected to get audited by the IRS or turned down for a credit card, an expert system made that decision.

The keyword mentioned in the previous sentence is "well-defined," and will be discussed in more detail later. The basic idea is that if a human expert can specify the steps of reasoning by which a problem may be solved, so too can an expert system. If a person cannot explain their reasoning, they may have better luck in Las Vegas.

As a counterexample, many people have attempted to write expert systems to predict the stock market; in fact Wall Street uses these systems all the time. However, if you look at all the ups and downs of Wall Street, no obvious trend occurs and the systems are no better than their creator. The big advantage of these systems is in real-time trading in which delays in purchasing of a millisecond are critical since a competitor's expert system may notice the same trend as yours and be placing buy or sell orders for hundreds of millions of dollars worth of stock, far faster than any human being can do. How well does it work? The infamous stock market crash of 1987 brought a host of new restrictions to trading to prevent computers selling hundreds of millions of shares to make a profit of a few hundred dollars and potentially cause a crash.

Expert systems is a very successful application of artificial intelligence technology. Many hybrid approaches exist to combine expert systems with other techniques such as genetic algorithms and artificial neural networks. The common term for a system that uses AI is **intelligent system** or automated system (Hopgood 01).

In general, the first step in solving any problem is defining the problem area or **domain** to be solved. This consideration is just as true in AI as in conventional programming. However, because of the mystique associated with AI by the general public, there is a lingering tendency to still believe the old adage: "It's an AI problem if it ain't been solved yet." Another popular definition believed by most people is that "AI is making computers act like they do in the movies." This type of mindset may have been popular in the 1970s when AI was entirely in a research stage. Today there are many real-world problems that are being solved by AI and many commercial applications, as discussed in online magazines such as *PCAI.com*, conferences such as the AAAI (http://aaii.org/conferences/conferences.htm), and books (Luger 02). For more details, see Appendix G.

Before discussing AI in more detail, it is worthwhile to step back and look at the big picture of how AI fits into the scheme of life itself. This leads us to the first question: What is life? There are many definitions of life as described in

(Adami 98) ranging from physiological, metabolic, biochemical, genetic, and thermodynamic depending on how you view it. Which definition is correct? It all depends on what aspect of life you are interested in. Perhaps the simplest is Shakespeare's, "Life is a tale told by an idiot, full of sound and fury, signifying nothing."

From a computer perspective, *life* can be represented as software. In fact there is software on the CD in the Adami book that allows the user to create artificial lifeforms and experiment with them. There is also the metaphysical definition of life as so aptly described in movies like *The Matrix* in which people "line" inside some giant computer program. Other aspects of artificial life as created in the digital computer are described in (Helmreich 98), which discusses in more detail the philosophical and even spiritual aspects of computer artificial life.

From a biological point of view, we are no longer limited to computer systems in the quest to create artificial lifeforms. Starting in the 1990s it was possible to clone mammals such as Dolly the Sheep, and companies now sell cows to business as well as cloned pets such as cats to bereaved pet owners. But cloning to make a close copy of a living creature, i.e., artificial life, is only the first step in "improving" life. For example, one group of researchers has created a bunny with an extra gene from fireflies to make it glow in the dark. Such forms of artificial life have no single natural ancestors from which to descend and are truly artificial life. At the same time, since these creatures are intelligent, they can also be considered to have artificial intelligence, although not the kind it has been customary to represent using digital computers.

Extending on artificial life is the new field of *creative evolutionary systems* in which artificial life systems are allowed to change their own programming in response to evolutionary pressure as described in (Bentley 02). Many different techniques such as genetic algorithms are described with practical applications to music, art, circuit design, architecture, and fighter aircraft maneuvering. Also included with the Bentley book is a CD that allows the user to experiment with creative evolutionary systems. Note that the book is concerned with the computer representation of these systems, not the new biological lifeforms which in the future will be purposely grown to achieve the designer's dreams—e.g., a glowing bunny (already created) to comfort children in the dark, or a monkey that is a better aircraft pilot than any human being.

In (de Silva 00) another definition of intelligence is given as: "Intelligence is the capacity to learn, the capacity to acquire, adapt, modify, and extend knowledge in order to solve problems." In this case, the desire is to build intelligent machines that interact with the real world through robotics, factories, appliances, and other hardware. The challenge is to incorporate into machines the complex human mechanisms in dealing with the real world such as ambiguity, vagueness, generality, imprecision, uncertainty, fuzziness, belief, and plausibility. Many of these topics are discussed later in this book in Chapters 4 and 5 on Uncertainty. Note that the previous sentence is itself vague. Does "this book" refer to the de Silva book or *Expert Systems Principles and Programming*? As living creatures we are used to dealing with problems like this all the time but robots and computers have a difficult time with ambiguity if only classical logic is used.

An even more challenging problem is to develop artificial intelligence systems that are also conscious. While much is known about the brain (Cotterill

98), we still do not know where is the screen on which consciousness is played or what makes you the person you are. However with new tools such as functional magnetic resonance imaging (fMRI), the brain is being mapped dynamically to see which areas are activated during metal activities. Of course if we use the artificially cloned animal's model, then we have already succeeded because the sheep and cats are certainly conscious. Nevertheless, we still do not know how to imbue a machine with consciousness. Even more important, after seeing the *Terminator* or the *Matrix* movies, it may not be wise to even want a conscious machine intelligence. After all, no one likes to be unplugged, people or machines.

Although perfect solutions to classic AI problems such as natural language translation, speech understanding, and vision have not been found, restricting the problem domain produces a useful solution. For example, it is not difficult today to build simple natural language systems if the input is restricted to sentences of the form: noun, verb, and object. Currently, systems of this type work very well in providing a user-friendly interface to many software products such as database systems and spreadsheets. Speaker-independent voice recognition systems are also now available with a high degree of accuracy that do not require training on a particular user's voice as was the case with early systems. Coupled with expert systems, such intelligent systems will eventually replace many telephone call centers that take orders from customers once these systems pass the Turing Test (Luger 02).

A number of commercial versions of speech recognition systems that work with standard PC programs are available and quite reasonably priced. Voice recognition systems are also widely available for hands-free cell phone operation in automobiles and have excellent recognition if the problem domain is limited to digits rather than all words. In fact, the parsers associated with popular computer text-adventure games today exhibit an amazing degree of ability in understanding natural language, which is a necessity with multiplayer LAN-party games in which typing would slow down the game.

Expert systems have been combined with databases for human-like pattern recognition and automated decision systems to yield knowledge discovery through **data mining** and thus produce an **intelligent database** (Bramer 99). One important application is in airport security systems that use face recognition of suspects as a front-end to an expert system, which then determines if there is justification in proceeding with further notification of authorities.

Another exciting area of artificial intelligence has to do with artificial **discovery systems.** These are computer programs that can actually discover knowledge in certain problem domains. For example, the Automatic Mathematician (AM) program discovers new mathematical theorems and rediscovers knowledge made by humans such as the significance of prime numbers. The BACON 3 discovery system discovers new scientific knowledge such as a version of Kepler's third law of planetary motion, and a number of discovery systems are summarized in (Wagman 99).

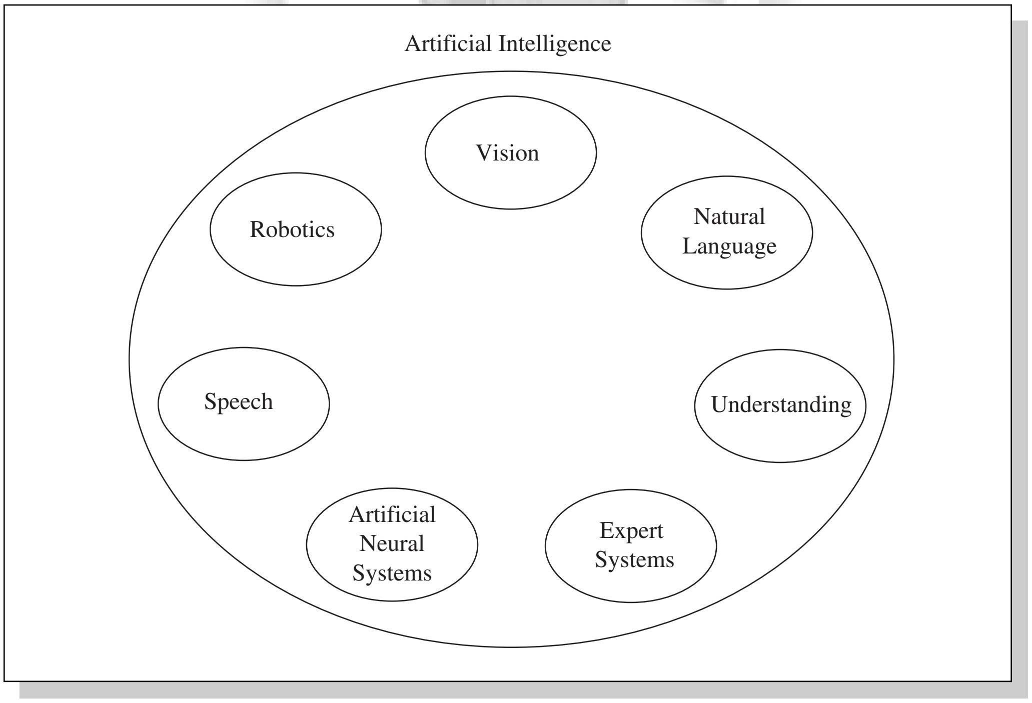
While AI was originally defined as a branch of Computer Science in the 20th Century, it is now a standalone discipline that draws on many fields such as computer science, psychology, biology, neuroscience, and many others. In fact there are a growing number of universities that offer degrees in AI.

Figure 1.1 shows some areas of interest for AI. The area of expert systems is a very successful approximate solution to the classic AI problem of programming intelligence. Professor Edward Feigenbaum of Stanford University, an early pioneer of expert systems technology, has defined an **expert system** as "... an intelligent computer program that uses knowledge and inference procedures to solve problems that are difficult enough to require significant human expertise for their solution". That is, an expert system is a computer system that emulates the decision-making ability of a human expert. The term **emulate** means that the expert system is intended to act in all respects like a human expert. An emulation is much stronger than a *simulation* which is required to act like the real thing in only some respects.

Although a general-purpose problem solver still eludes us, expert systems function very well in their restricted domains. As proof of their success, you need only observe the many applications of expert systems today in business, medicine, science, and engineering as well as all the books, journals, conferences, and products devoted to expert systems shown in Appendix G.

Expert systems makes extensive use of specialized knowledge to solve problems at the level of a human expert. An **expert** is a person who has *expertise* in a certain area. That is, the expert has knowledge or special skills that are not known or available to most people. An expert can solve problems that most people cannot solve at all or solve them much more efficiently (but not necessarily

Figure 1.1 Some Areas of Artificial Intelligence



as inexpensively.) When expert systems were first developed they contained expert knowledge exclusively. However, the term expert system is often applied today to any system that uses expert system technology. Expert system technology may include special expert system languages, programs, and hardware designed to aid in the development and execution of expert systems.

The knowledge in expert systems may be either expertise, or knowledge that is generally available from books, magazines, and knowledgeable persons. In this sense, *knowledge* is considered to be at a lower level than the more rare expertise. The terms **expert system**, **knowledge-based system,** and **knowledge-based expert system** are often used synonymously. Most people use expert system simply because it's shorter, even though there may be no expertise in their expert system, only knowledge.

Figure 1.2 illustrates the basic concept of a knowledge-based expert system. The user supplies facts or other information to the expert system and receives expert advice or expertise in response. Internally, the expert system consists of two main components. The knowledge base contains the knowledge with which the **inference engine** draws conclusions. These conclusions are the expert system's responses to the user's queries for expertise.

Useful knowledge-based systems also have been designed to act as an intelligent assistant to a human expert. These intelligent assistants are designed with expert systems technology because of the development advantages. As more knowledge is added to the intelligent assistant, it acts more like an expert. Developing an intelligent assistant may be a useful milestone in producing a complete expert system. In addition, it may free up more of the expert's time by speeding up the solution of problems. Intelligent tutors are another application of artificial intelligence. Unlike the old computer-assisted instruction systems, intelligent tutor systems can provide context-sensitive instruction (Giarratano 91a).

An expert's knowledge is specific to one **problem domain** as opposed to general problem-solving techniques. A problem domain is the special problem area such as medicine, finance, science, or engineering that an expert can solve problems in very well. Expert systems, like human experts, are generally

Figure 1.2 Basic Function of an Expert System

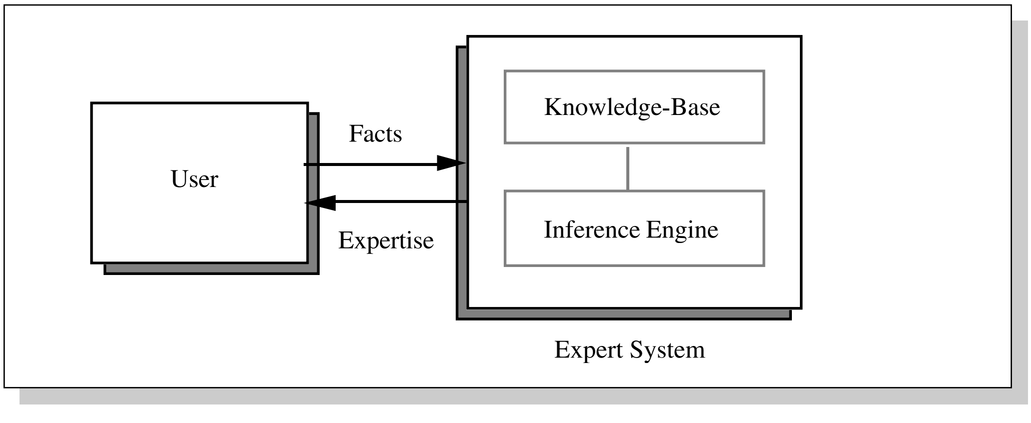
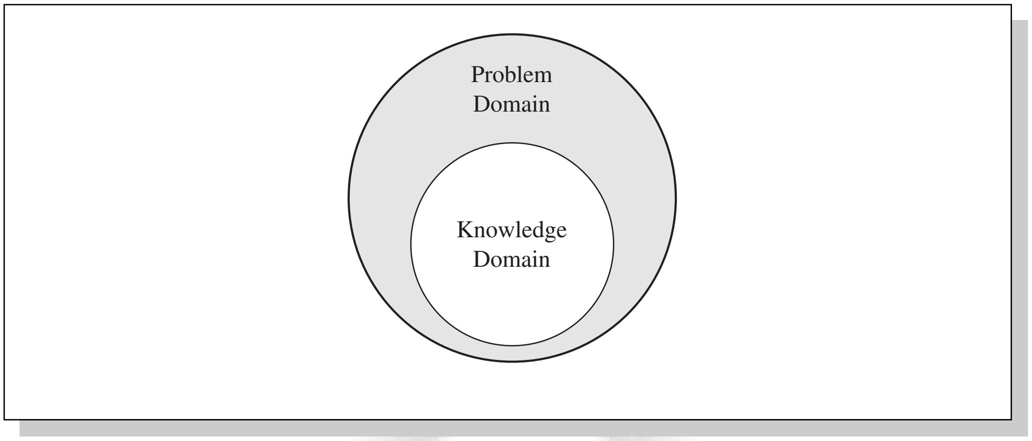


Figure 1.3 Problem and Knowledge Domain Relationship



designed to be experts in one problem domain. For example, you would not normally expect a chess expert to have expert knowledge about medicine. Expertise in one problem domain does not automatically carry over to another.

The expert's knowledge about solving specific problems is called the **knowledge domain** of the expert. For example, a medical expert system designed to diagnose infectious diseases will have a great deal of knowledge about certain symptoms caused by infectious diseases. In this case the knowledge domain is medicine and consists of knowledge about diseases, symptoms, and treatments. Figure 1.3 illustrates the relationship between the problem and knowledge domain. Notice that this knowledge domain is entirely included within the problem domain. The portion outside the knowledge domain symbolizes the area in which there is not knowledge about all the problems within the problem domain.

One expert system, such as an infectious diseases diagnostic system, usually does not have knowledge about other branches of medicine such as surgery or pediatrics. Although its knowledge of infectious disease is equivalent to or greater than a human expert, the expert system would not know anything about other knowledge domains unless it was programmed with that domain knowledge.

In the knowledge domain that it knows about, the expert system reasons or makes **inferences** in the same way that a human expert would reason or infer the solution of a problem. That is, given some facts, a logical, possible conclusion that follows is inferred by reason. For example, if your spouse hasn't spoken to you in a month, you may infer that he or she had nothing worthwhile to say. However, this is only one of several possible inferences.

As with any technology, there are many ways of viewing its utility. Table 1.1 summarizes the differing views of the participants in a technology. In this table, the technologist may be an engineer or software designer and the technology may be hardware or software. In solving any problem, there are questions that need to be answered or the technology will not be successfully used. Like any other tool, expert systems have appropriate and inappropriate applications. Chapter 6 discussed choosing appropriate applications in more detail.

Table 1.1 Differing Views of Technology

Person	Question
Manager	What can I use it for?
Technologist	How can I best implement it?
Researcher	How can I extend it?
Consumer	Will it save me time or money?
Business owner	Can I cut labor?
Stockbroker	How will it affect quarterly profits?

1.3 ADVANTAGES OF EXPERT SYSTEMS

Expert systems have a number of attractive features:

- *Increased availability*. Expertise is available on any suitable computer hardware. In a very real sense, an expert system is the mass production of expertise.
- *Reduced cost.* The cost of providing expertise per user is greatly lowered.
- *Reduced danger*. Expert systems can be used in environments that might be hazardous for a human.
- *Permanence.* The expertise is permanent. Unlike human experts who may retire, quit, or die, the expert system's knowledge will last indefinitely.
- *Multiple expertise*. The knowledge of multiple experts can be made available to work simultaneously and continuously on a problem at any time of day or night. The level of expertise combined from several experts may exceed that of a single human expert.
- *Increased reliability*. Expert systems increase confidence that the correct decision was made by providing a second opinion to a human expert or a tie-breaker in disagreements among multiple human experts. (Of course, this method probably won't work if the expert system was programmed by one of the experts.) The expert system should always agree with the expert, unless a mistake was made by the expert, which may happen if the human expert is tired or under stress.
- *Explanation.* The expert system can explain in detail the reasoning that led to a conclusion. A human may be too tired, unwilling, or unable to do this all the time. This increases the confidence that the correct decision is made.
- *Fast response.* Fast or real-time response may be necessary for some applications. Depending on the software and hardware used, an expert system may respond faster and be more available than a human expert. Some emergency situations may require responses faster than a human; in this case a real-time expert system is a good choice.
- *Steady, unemotional, and complete response at all times.* This may be very important in real-time and emergency situations when a human expert may not operate at peak efficiency because of stress or fatigue.
- *Intelligent tutor.* The expert system may act as an intelligent tutor by letting the student run sample programs and explaining the system's reasoning.

- *Intelligent database.* Expert systems can be used to access a database in an intelligent manner. Data mining is an example.

The process of developing an expert system has an indirect benefit also since the knowledge of human experts must be put into an explicit form for entering in the computer. Because the knowledge is then explicitly known instead of being implicit in the expert's mind, it can be examined for correctness, consistency, and completeness. The knowledge may then have to be adjusted (which is not appreciated by the expert!)

1.4 GENERAL CONCEPTS OF EXPERT SYSTEMS

The knowledge of an expert system may be represented in a number of ways. One common method of representing knowledge is in the form of IF THEN type **rules**, such as:

```
IF the light is red THEN stop
```

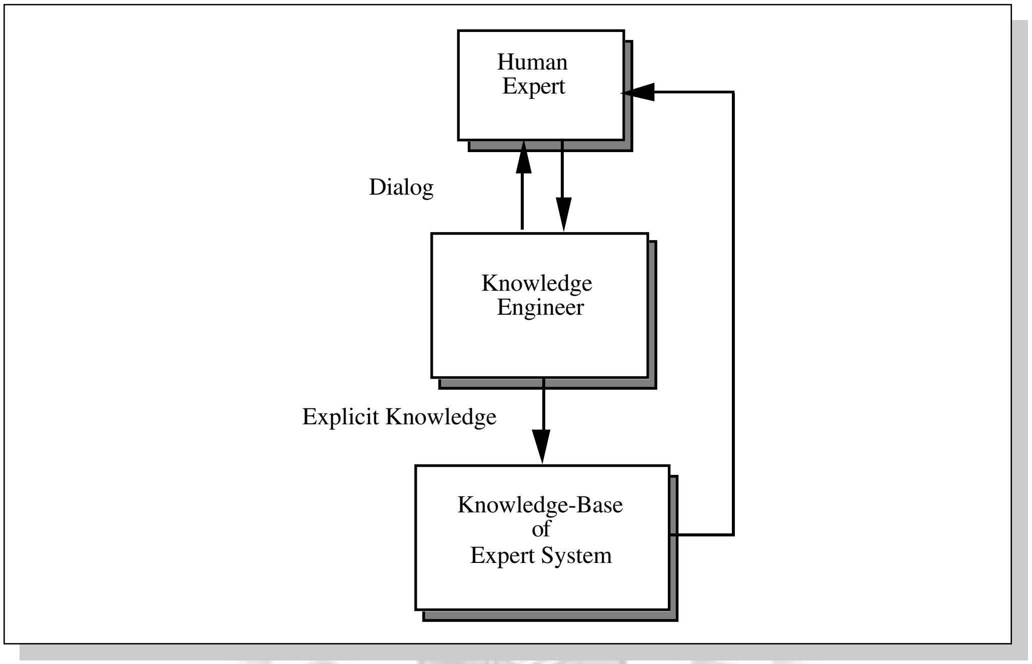
If a fact exists that the light is red, this matches the pattern "the light is red." The rule is satisfied and performs its action of "stop." Although this is a very simple example, many significant expert systems have been built by expressing the knowledge of experts in rules. In fact, the knowledge-based approach to developing expert systems has completely supplanted the early AI approach of the 1950s and 1960s which tried to use sophisticated reasoning techniques with no reliance on knowledge. Some types of expert systems tools such as CLIPS allow **objects** as well as rules. Knowledge can be encapsulated in rules and objects. Rules can pattern match on objects as well as facts. Alternatively, objects can operate independently of the rules.

Since their first successful commercial use in the XCON/R1 system of Digital Equipment Corporation, which knew much more than any single human expert on how to configure computer systems, expert systems have demonstrated their value and usefulness over and over again. Many small systems for specialized tasks have been constructed with several hundred rules. These small systems may not operate at the level of an expert but are designed to take advantage of expert systems technology to perform knowledge-intensive tasks. For these small systems, the knowledge may be in books, journals, or other publicly available documentation.

In contrast, a classic expert system embodies unwritten knowledge that must be extracted from an expert by extensive interviews with a **knowledge engineer** over a long period of time. The process of building an expert system is called **knowledge engineering** and is done by a knowledge engineer. Knowledge engineering refers to the acquisition of knowledge from a human expert or other source and its coding in the expert system.

The general stages in the development of an expert system are illustrated in Figure 1.4. The knowledge engineer first establishes a dialog with the human expert in order to elicit the expert's knowledge. This stage is analogous to a system designer in conventional programming discussing the system requirements with

Figure 1.4 Development of an Expert System



a client for whom the program will be constructed. The knowledge engineer then codes the knowledge explicitly in the knowledge base. The expert then evaluates the expert system and gives a critique to the knowledge engineer. This process iterates until the system's performance is judged to be satisfactory by the expert.

The expression **knowledge-based system** is a better term for the application of knowledge-based technology because it may be used for the creation of either expert systems or knowledge-based systems. However, like the term *artificial intelligence*, it is common practice today to use the term *expert systems* when referring to both expert systems and knowledge-based systems, even when the knowledge is not at the level of a human expert.

Expert systems are generally designed very differently from conventional programs because the problems have no satisfactory algorithmic solution and rely on inferences to achieve a reasonable solution. An algorithm is the ideal solution to a problem because it is guaranteed to yield an answer in finite time (Berlinski 00). However an algorithm may not be satisfactory and the problem scales up in size, and that is why AI is used. Note that a reasonable solution is about the best we can expect if no algorithm is available to help us achieve the optimum solution. Since the expert system relies on inference, it must be able to explain its reasoning so that its reasoning can be checked. An **explanation facility** is an integral part of sophisticated expert systems. In fact, elaborate explanation facilities may be designed to allow the user to explore multiple lines of "What if" type questions, called **hypothetical reasoning.**

Some expert systems even allow the system to learn rules by example, through **rule induction**, in which the system creates rules from data. Formalizing the knowledge of experts into rules is not simple, especially when the expert's knowledge has never been systematically explored. There may be inconsistencies, ambiguities, duplications, or other problems with the expert's knowledge that are not apparent until attempts are made to formally represent the knowledge in an expert system.

Human experts should also know the extent of their knowledge and qualify their advice as the problem reaches their **limits of ignorance**. A human expert also knows when to "break the rules." Unless expert systems are explicitly designed to deal with uncertainty, they will make recommendations with the same confidence even if the data they are dealing with is inaccurate or incomplete. An expert system's advice, like a human expert's, should degrade gracefully at the boundaries of ignorance rather than abruptly.

A practical limitation of many expert systems today is lack of **causal knowledge**. That is, the expert systems do not really have an understanding of the underlying causes and effects in a system. It is much easier to program expert systems with **shallow knowledge** based on empirical and heuristic knowledge than **deep knowledge** based on the basic structure, function, and behavior of objects. For example, it is much easier to program an expert system to prescribe an aspirin for a person's headache than program all the underlying biochemical, physiological, anatomical, and neurological knowledge about the human body. The programming of a causal model of the human body would be an enormous task and even if successful, the response time of the system would probably be extremely slow because of all the information that the system would have to process.

One type of shallow knowledge is **heuristic knowledge;** the term *heuristic* comes from the Greek and means *to discover*. Heuristic solutions are not guaranteed to succeed in the same way that an algorithm is a guaranteed solution to a problem. Instead, heuristics are rules of thumb or empirical knowledge gained from experience that may aid in the solution but are not guaranteed to work. However, in many fields such as medicine and engineering, heuristics play an essential role in some types of problem solving. Even if an exact solution is known, it may be impractical to use because of cost or time constraints. Heuristics can provide valuable shortcuts that can reduce time and cost.

Another problem with expert systems today is that their expertise is limited to the knowledge domain contained in the systems. Typical expert systems cannot generalize their knowledge by using **analogy** to reason about new situations the way people can. Although rule induction helps, only limited types of knowledge can be put into an expert system this way. The customary way of building an expert system by having the knowledge engineer repeat the cycle of interviewing the expert, constructing a prototype, testing, interviewing, and so on is a very time-consuming and labor intensive task. In fact, this problem of transferring human knowledge into an expert system is so major that it is called the **knowledge acquisition bottleneck**. This is a descriptive term because the knowledge acquisition bottleneck constricts the building of an expert system like an ordinary bottleneck constricts fluid flow from a bottle.

In spite of their present limitations, expert systems have been very successful in dealing with real-world problems that conventional programming methodologies have been unable to solve, especially those dealing with uncertain or incomplete information. The important point is to be aware of the advantages and limitations of any technology so that it can be appropriately utilized.

1.5 CHARACTERISTICS OF AN EXPERT SYSTEM

An expert system is usually designed to have the following general characteristics:

- *High performance*. The system must be capable of responding at a level of competency equal to or better than an expert in the field. That is, the quality of the advice given by the system must have high integrity.
- *Adequate response time*. The system must perform in a reasonable time, comparable to or better than the time required by an expert to reach a decision. An expert system that takes a year to reach a decision compared to a human expert's time of one hour would not be useful. The **time constraints** placed on the performance of an expert system may be especially severe in the case of real-time systems, when a response must be made within a certain time interval such as landing an aircraft in fog.
- *Good reliability*. The expert system must be reliable and not prone to crashes or else it will not be used.
- *Understandable*. The system should be able to explain the steps of its reasoning while executing so that it is understandable. Rather than being just a "black box" that produces a miraculous answer, the system should have an explanation capability in the same way that human experts can explain their reasoning. This feature is important for several reasons:

One reason is that human life and property may depend on the answers of the expert system. Because of the great potential for harm, an expert system must be able to justify its conclusions in the same way a human expert can explain why a certain conclusion was reached. Thus, an explanation facility provides a sanity check of the reasoning for humans.

A second reason for having an explanation facility occurs in the development phase of an expert system to confirm that the knowledge has been correctly acquired and is being correctly used by the system. This is critical in debugging since the knowledge may be incorrectly entered due to typos or be incorrect due to misunderstandings between the knowledge engineer and the expert. A good explanation facility allows the expert and knowledge engineer to validate the accuracy of the knowledge. Also, because of the way that typical expert systems are constructed, it is very difficult to read a long program listing and understand its operation.

An additional source of error may be unforeseen interactions in the expert system, which may be detected by running test cases with known reasoning that the system should follow. As we will discuss in more detail later, multiple rules may apply to a given situation about which the system is reasoning. The flow of execution is not sequential in an expert system; i.e., you cannot just read its code line by line and understand how the system operates. The order in which rules have been entered in the system is not necessarily the order in which they will be executed. The expert system acts much like a parallel program in which the rules are independent knowledge processors.

- *Flexibility*. Because of the large amount of knowledge that an expert system may have, it is important to have an efficient mechanism for adding, changing, and deleting knowledge. One reason for the popularity of rule-based systems is the efficient and modular storage capability of rules.

Depending on the system, an explanation facility may be simple or elaborate. A simple explanation facility in a rule-based system may simply list all the facts that caused the latest rule to execute. More elaborate systems may do the following:

- *List all the reasons for and against a particular hypothesis*. A **hypothesis** is a **goal** which is to be proved; e.g., in a medical diagnostic expert system the hypothesis might be "The patient has a tetanus infection." In a real problem, there may be multiple hypotheses, or the patient may really have several diseases at once. A hypothesis is an assertion whose truth is in doubt and must be proved. Once a goal has been assumed, **subgoals** are generated that are simpler, and so on until simple enough subgoals may be solved. This method of solving a problem is the classic **divide-and-conquer** method that is the basis of backward chaining discussed in a later chapter.
- List all the hypotheses that may explain the observed evidence.
- Explain all the consequences of a hypothesis. For example, assuming that the patient does have tetanus, there should also be evidence of fever as the infection runs its course. If this symptom is then observed, it adds credibility that the hypothesis is true. If the symptom is not observed, it reduces the credibility of the hypothesis.
- Give a **prognosis** or prediction of what will occur if the hypothesis is true.
- Justify the questions that the program asks of the user for further information. These questions may be used to direct the line of reasoning to likely diagnostic paths. In most real problems, it is too expensive or time consuming to explore all possibilities, and some way must be provided to guide the search for the correct solution. For example, consider the cost, time, and effect of administering all possible medical tests to a patient complaining of a sore throat (very profitable to the medical business, very bad for the patient).
- *Justify the knowledge of the program.* For example, if the program claims that the hypothesis "The patient has a tetanus infection" is true, the user could ask for an explanation. The program might justify this conclusion on the basis of a rule that says that if the patient has a positive blood test for tetanus, the patient has tetanus. Now the user could ask the program to justify this rule. The program could respond by stating that a positive blood test for a disease is proof of the disease. Unfortunately this ignores false positives, as discussed in a later chapter.

In this case, the program is actually quoting a **metarule**, which is knowledge about rules. The prefix *meta* means "above or beyond." Programs have been explicitly created to infer new rules using the process of **machine learning**. A hypothesis is justified by knowledge and the knowledge is justified by a **warrant** that it is correct. A warrant is essentially a meta-explanation that explains the expert system's explanation of its reasoning.

Knowledge can easily grow **incrementally** in a rule-based system, which is one of the reasons why these systems have proven so successful. That is, the knowledge base can grow little by little as rules are added so that the performance and validity of the system can be continually verified. This is analogous to the way a child learns new knowledge every day and checks that the new knowledge is correct. If the rules are properly designed, the interactions between rules will be minimized or eliminated to protect against unforeseen effects. The incremental growth of knowledge facilitates **rapid prototyping** so that the knowledge engineer can quickly show a working prototype of the expert system. This is an important feature because it maintains the expert's and management's interest in the project. Rapid prototyping also quickly exposes any gaps, inconsistencies, or errors in the expert's knowledge or the system so that corrections can be made immediately.

1.6 THE DEVELOPMENT OF EXPERT SYSTEMS TECHNOLOGY

The roots of expert systems lie in many disciplines; in particular, the area of psychology that is concerned with human information processing, **cognitive science**. Cognition is the study of how humans know or process information. In other words, cognition is the study of how people think, especially when solving problems. A variety of cognitive tools have been developed to provide better teaching (Lajoie 00).

Another important concept that AI machines must have is the ability to recognize signs, which forms the basis of the field named **semiotics** (Fetzer 01). In semiotics, we do not mean simple written signs such as a Stop sign, but the whole concept of signs in general. A sign is something that represents something else. For example, if you watch a movie that also has music playing, a common sign that something exciting is going to happen is a rise in pitch and faster rhythm of the music. Likewise when something suspenseful is about to happen, the music becomes slower and deeper. Many different nonverbal signs exist in music, movies, television, and ordinary life. For example, if a person is lying or uncomfortable about a question, they will usually drop their eyes. This has vast bearing on intelligent machines designed to exist in the real world. Simple programming to understand words is not enough; the machines must also recognize the underlying meaning of signs.

The study of cognition is very important if we want to make computers emulate human experts. Often, experts can't explain how they solve problems—the solution just comes to them. If the expert can't explain how a problem is solved, it's not possible to encode the knowledge in an expert system based on explicit knowledge. In this case, the only possibility is programs that learn by themselves to emulate the expert. These are programs based on induction, artificial neural systems, and other soft computing methods to be discussed later.

Human Problem Solving and Productions

The development of expert systems technology draws on a wide background. Table 1.2 is a brief summary of some important developments that have led to CLIPS. Whenever possible, the starting dates of projects are used. Many projects extend over many years. These developments are covered in more detail in this chapter and others. A number of expert systems tools and modern expert systems are described in Appendix G.

Table 1.2 Some Important Events Leading to the First Release of CLIPS

Year	Events
1943	Post production rules; McCulloch and Pitts Neuron Model
1954	Markov Algorithm for controlling rule execution
1956	Dartmouth Conference; Logic Theorist; Heuristic Search; term "AI" coined
1957	Perceptron invented by Rosenblatt; GPS (General Problem Solver) started (Newell, Shaw, and Simon)
1958	LISP AI language (McCarthy)
1962	Rosenblatt's *Principles of Neurodynamics* on Perceptions
1965	Resolution Method of automatic theorem proving (Robinson) Fuzzy Logic for reasoning about fuzzy objects (Zadeh) Work begun on DENDRAL, the first expert system (Feigenbaum, Buchanan)
1968	Semantic nets, associative memory model (Quillian)
1969	MACSYMA math expert system (Martin and Moses)
1970	Work begins on PROLOG (Colmerauer, Roussell)
1971	HEARSAY I for speech recognition *Human Problem Solving* popularizes rules (Newell and Simon)
1973	MYCIN expert system for medical diagnosis (Shortliffe) leading to GUIDON, intelligent tutoring (Clancey) and TEIRESIAS, explanation facility concept (Davis) and EMYCIN, first shell (Van Melle, Shortliffe, and Buchanan) HEARSAY II, blackboard model of multiple cooperating experts
1975	Frames, knowledge representation (Minsky)
1976	AM (Artificial Mathematician) creative discovery of math concepts (Lenat); Dempster-Shafer Theory of Evidence for reasoning under uncertainty; Work begun on PROSPECTOR expert system for mineral exploration (Duda, Hart)
1977	OPS expert system shell (Forgy), an ancestor of CLIPS
1978	Work started on XCON/R1 (McDermott, DEC) to configure DEC computer systems Meta-DENDRAL, metarules, and rule induction (Buchanan)
1979	Rete algorithm for fast pattern matching (Forgy) Commercialization of AI begins Inference Corp. formed (releases ART expert system tool in 1985)
1980	Symbolics, LMI founded to manufacture LISP machines
1982	SMP math expert system; Hopfield Neural Net; Japanese Fifth Generation Project to develop intelligent computers
1983	KEE expert system tool (IntelliCorp)
1985	CLIPS version 1 expert system tool (NASA). Freely available on all computers, not just special purpose and expensive LISP machines.

In the late 1950s and 1960s, a number of programs were written with the goal of general problem solving. The most famous of these was the **General Problem Solver** created by Newell and Simon. This caused a huge sensation because the press termed the big computers then that filled a whole room, "Giant Brains." People became afraid that they would lose their jobs until companies such as IBM claimed the machines would only do the work that thousands of mathematicians would take a million years to accomplish. In those days when the machines cost a million dollars and CPU time was charged at $1 a microsecond, it never seemed possible that people would someday have inexpensive personal computers in their homes or at work.

One of the most significant results demonstrated by Newell and Simon was that much of human understanding or **cognition** could be expressed by IF-THEN **production rules**. For example, IF it looks like it's going to rain THEN carry an umbrella, or IF your spouse is in a bad mood THEN don't appear happy. A rule corresponds to a small, modular collection of knowledge called a **chunk**. The chunks are organized in a loose arrangement with links to related chunks of knowledge. One theory is that all human memory is organized in chunks. Here is an example of a rule representing a chunk of knowledge:

```
IF the car doesn't run and
   the fuel gauge reads empty
THEN fill the gas tank
```

Newell and Simon popularized the use of rules to represent human knowledge and showed how standard reasoning could be done with rules. Cognitive psychologists have used rules as a model to explain human information processing. The basic idea is that sensory input provides stimuli to the brain. The stimuli trigger the appropriate rules of **long-term memory,** which produces the appropriate response. Long-term memory is where our permanent knowledge is stored. For example, we all have rules such as:

```
IF there is flame THEN there is a fire
IF there is smoke THEN there may be a fire
IF there is a siren THEN there may be a fire
```

Notice that the last two rules are not expressed with complete certainty. The fire may be out, but there may still be smoke in the air. Likewise, a siren does not prove that there is a fire since it may be a false alarm. The stimuli of seeing flames, smelling smoke, and hearing a siren will trigger these and similar types of rules.

Long-term memory consists of many rules having the simple IF THEN structure. In fact, a Grand Master chess expert may know 50,000 or more chunks of knowledge about chess patterns. In contrast to the long-term memory, the **short-term memory** is used for the temporary storage of knowledge during problem solving. Although long-term memory can hold hundreds of thousands or more chunks, the capacity of working memory is surprisingly small—four to seven chunks. As a simple example of this, try visualizing some numbers in your mind. Most people can only see four to seven numbers at once. Of course we can memorize many more than four to seven numbers. However, those numbers are kept in long-term memory and it takes a while to make them permanent.

One theory proposes that short-term memory represents the number of chunks that simultaneously can be active and considers human problem solving as a spreading of these activated chunks in the mind. Eventually, a chunk may be activated with such intensity that a conscious thought is generated and you say to yourself, "Hmm ... something's burning. Wonder if my pants are on fire?"

The other element necessary for human problem solving is a **cognitive processor**. The cognitive processor tries to find the rules that will be **activated** by the appropriate stimuli. Not just any rule will do. For example, you wouldn't want to fill your gas tank every time you heard a siren. Only a rule that matched the stimuli would be activated. If there are multiple rules that are activated at once, the cognitive processor must perform a **conflict resolution** to decide which rule has highest priority. The rule with highest priority will be executed. For example, if both of the following rules are activated:

```
IF there is a fire THEN leave
IF my clothes are burning THEN put out the fire
```

then the actions of one rule will be executed before the other. The **inference engine** of modern expert systems corresponds to the cognitive processor.

The Newell and Simon model of human problem solving in terms of long-term memory (rules), short-term memory (working memory), and a cognitive processor (inference engine) is the basis of modern rule-based expert systems.

Rules like these are a type of **production system**. Rule-based production systems are a popular method of implementing expert systems today. The individual rules that compose a production system are the **production rules**. In designing an expert system, an important factor is the amount of knowledge or **granularity** of the rules. Too little granularity makes it difficult to understand a rule without reference to other rules. Too much granularity makes the expert system difficult to modify since several chunks of knowledge are intermingled in one rule. The ability of CLIPS to use objects as well as rules is a major asset.

Until the mid-1960s, a major quest of AI was to produce intelligent systems that relied little on domain knowledge and greatly on powerful methods of reasoning. Even the name *General Problem Solver* illustrates the concentration on machines that were not designed for one specific domain but were intended to solve many types of problems. Although the methods of reasoning used by general problem solvers were very powerful, the machines were eternal beginners. When presented with a new domain, they had to discover everything from first principles and were not as good as human experts who relied on domain knowledge for high performance.

An example of the power of knowledge is playing chess. Although computers now rival humans, people play well despite the fact that computers can do calculations millions of times faster. Studies have shown that human expert chess players do not have super powers of reasoning but instead rely on knowledge of chesspiece patterns built up over years of play. As mentioned previously, one estimate places an expert chess player's knowledge at about 50,000 patterns.

Humans are very good at recognizing patterns such as pieces on a chessboard. Instead of trying to reason ahead 50, 100 or more possible moves for every piece, the best human chessplayers analyze the game in terms of patterns that reveal long-term threats while remaining alert for short-term surprise moves. This strategy rather than the bruteforce look-ahead method of anticipating moves is what has enabled computer chess programs like IBM's Big Blue to beat all the human chess champions.

While domain knowledge is very powerful, it is generally limited to the particular domain. For example, a person who becomes an expert chess player does not automatically become an expert at solving math problems or even a checkers expert. While some knowledge may carry over to another domain, such as careful planning of moves, this is a skill rather than genuine expertise.

By the early 1970s, it became apparent that domain knowledge was the key to building machine problem solvers that could function at the level of human experts. Although methods of reasoning are important, studies have shown that experts do not primarily rely on reasoning for problem solving. In fact, reasoning may play a minor role in an expert's problem solving. Instead, experts rely on a vast knowledge of heuristics and experience that they have built up over the years. If an expert cannot solve a problem based on expertise, then it is necessary for the expert to reason from first principles and theory (or more likely ask another expert.) The reasoning ability of an expert is generally no better than that of an average person in dealing with a totally unfamiliar situation. The early attempts at building powerful problem solvers based only on reasoning have shown that a problem solver is crippled if it must rely solely on reasoning.

The insight that domain knowledge was the key to building real-world problem solvers led to the success of expert systems. The successful expert systems today are thus knowledge-based expert systems rather than general problem solvers. The same technology that led to the development of expert systems led to the development of knowledge-based systems that do not necessarily contain human expertise.

While expertise is considered knowledge that is specialized and known to only a few, knowledge is generally found in books, the Web, periodicals, and other widely available resources. For example, the knowledge of how to solve a quadratic equation or perform integration and differentiation is widely available. Knowledge-based computer programs such as Mathematica and MATLAB are available to automatically perform these and many other mathematical operations on either numeric or symbolic operands. Other knowledge-based programs may perform process control of manufacturing plants. Today the terms *knowledge-based systems* and *expert systems* are often used synonymously. In fact, expert systems is now considered an alternative programming model or **paradigm** to conventional algorithmic programming.

The Rise of Knowledge-Based Systems

With the acceptance of the knowledge-based paradigm in the 1970s, a number of successful expert systems were created. These systems are described here in more detail since they are all well documented with many papers and books devoted to their workings and knowledge base. The general problem with modern expert systems that you will read about is that their knowledge is proprietary and secret. Companies use expert systems to encapsulate the knowledge of human experts who work for them. They do not want this knowledge available to competitors and especially not to lawyers. Consider a medical expert system that gives a diagnosis from which the patient dies or claims injury. In fact the plaintiff need not even have suffered personal injury but simply be in a class action suit with others who claim injury. Both the software as well as the knowledge base in an expert system will be subject to inspection by other experts. As shown in many trials, you can always find one expert to refute another.

These classic expert systems could interpret mass spectrograms to identify chemical constituents (DENDRAL), diagnose illness (MYCIN), analyze geologic data for oil (DIPMETER) and minerals (PROSPECTOR), and configure computer systems (XCON/R1). The news that PROSPECTOR had discovered a mineral deposit worth $100 million dollars and that XCON/R1 was saving Digital Equipment Corporation (DEC) millions of dollars a year triggered a sensational interest in the new expert systems technology by 1980. The branch of AI that had started off in the 1950s as a study of human information processing had now grown to achieve commercial success by the development of practical programs for real-world use.

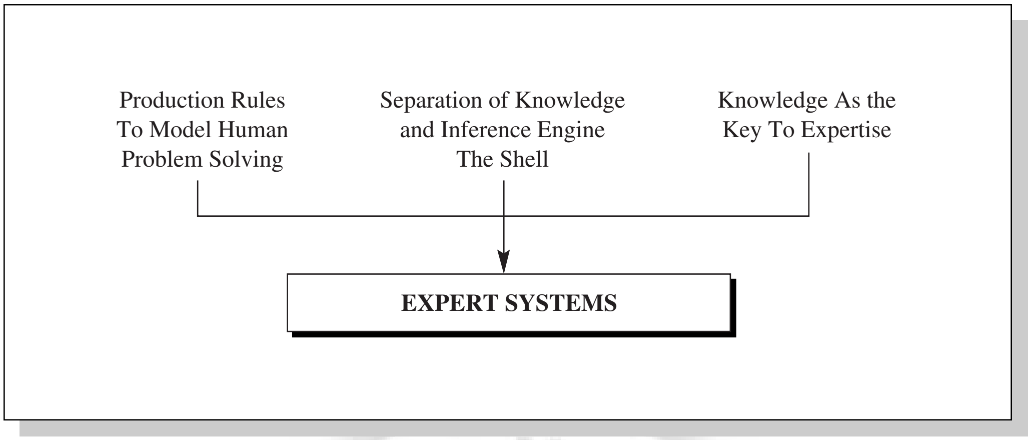
The MYCIN expert system was important for several reasons. First, it demonstrated that AI could be used for medical diagnosis. Second, MYCIN was the testbed of new concepts such as the explanation facility, automatic acquisition of knowledge, and intelligent tutoring that are found in a number of expert systems today. The third reason that MYCIN was important is that it demonstrated the feasibility of the expert system **shell** in which the program software is separate from the data. In other words, the data or knowledge is not hard-coded in with the software program. In fact a shell allows easy replacement of one knowledge-base domain with another.

Previous expert systems such as DENDRAL were one-of-a-kind systems in which the knowledge base was intermingled with the software that applied the knowledge, the inference engine. MYCIN explicitly separated the knowledge base from the inference engine. This was extremely important to the development of expert system technology since it meant that the essential core of the expert system could be reused. That is, a new expert system could be built much more rapidly than a DENDRAL-type system by emptying out the old knowledge and putting in knowledge about the new domain. The part of MYCIN that dealt with inference and explanation, the shell, could then be refilled with knowledge about the new system. The shell produced by removing the medical knowledge of MYCIN was called EMYCIN (essential or empty MYCIN).

By the late 1970s, the three concepts that are basic to most expert systems today had converged, as shown in Figure 1.5. These concepts are rules, the shell, and knowledge.

By 1980 new companies started to bring expert systems out of the university laboratory and produce commercial products. Powerful new software and specialized software written directly in LISP for expert system development were introduced. Unfortunately the high cost and significant training time ultimately

Figure 1.5 Convergence of Three Important Factors to Create the Modern Rule-Based Expert Systems


Production Rules To Model Human Problem Solving
Separation of Knowledge and Inference Engine The Shell
Knowledge As the Key To Expertise
EXPERT SYSTEMS

spelled the demise of these systems as PCs grew more powerful and expert systems tools such as CLIPS were introduced. In LISP machines, the native assembly language, operating system, and all other fundamental code were done in LISP so a great deal of maintenance was required.

CLIPS was originally written in C for speed and portability, and uses powerful pattern matching called the Rete Algorithm. Also unlike any other expert system tool, CLIPS is not only free, but the source code is well documented and comes with it. CLIPS can be installed on any C compiler that supports the standard Kernigan and Richie C language and has been installed on all makes of computers. A number of CLIPS-descended languages have been developed with advanced features such as fuzzy logic, and backward chaining, they have even been coded in Java, called Jess, as mentioned in the Preface.

1.7 EXPERT SYSTEMS APPLICATIONS AND DOMAINS

Conventional computer programs are used to solve many types of problems. Generally, these problems have algorithmic solutions that lend themselves well to conventional programs and programming languages such as C, C++, Java, C#, and so on. In many application areas such as industry and engineering, numeric calculations are of primary importance. By contrast, expert systems are primarily designed for symbolic reasoning.

While classic AI languages such as LISP and PROLOG are also used for symbolic manipulation, they are more general purpose than expert system shells. This does not mean that it is not possible to build expert systems in LISP and PROLOG. The term **logic programming** is generally applied today to programming that is done in PROLOG although a number of other languages have also been developed for this purpose. Many expert systems have been built with PROLOG and LISP. PROLOG especially has a number of advantages for diagnostic systems because of its built-in backward chaining. It is more convenient and efficient to build large expert systems with shells and utility programs specifically designed for expert system building. Instead of "reinventing the wheel" every time a new expert system is to be built, it is more efficient to use specialized tools designed for expert system building than general-purpose tools.

Applications of Expert Systems

Expert systems have been applied to virtually every field of knowledge. Some have been designed as research tools while others fulfill important business and industrial functions. The first major commercial success of an expert system in routine business use in the 1970s was the XCON system of DEC. The XCON system (originally called R1) was developed in conjunction with John McDermott of Carnegie-Mellon University. XCON was an expert configuring system for DEC computer systems.

The **configuration** of a system means that when a customer places an order, all the right parts—software, hardware, and documentation—are supplied. This is still a topic of major importance especially when people order computers, cars, and other products over the Internet by filling in forms. It would be extremely inefficient for a computer maker or automobile manufacturing plant to check each order for availability of parts and determine whether a special order can be delivered. Today, for any system that must be configured and delivered in a timely fashion, an expert system is a logical and financial necessity.

Thousands of expert systems have been built and reported in computer journals, the Internet, books, and conferences. This undoubtedly represents only the tip of the iceberg since many companies and government will not report details of their systems because of proprietary or secret knowledge contained in the systems. Based on the systems described in the open literature, certain broad classes of expert systems applications can be discerned, as shown in Table 1.3. Tables 1.4–1.9 list well known classic systems because they are well-documented, and modern systems have similar applications.

Table 1.3 Broad Classes of Expert Systems

Class	General Area
Configuration	Assemble proper components of a system in the proper way.
Diagnosis	Infer underlying problems based on observed evidence.
Instruction	Intelligent teaching so that a student can ask *why*, *how*, and *what if* questions just as if a human were teaching.
Interpretation	Explain observed data.
Monitoring	Compares observed data to expected data to judge performance.
Planning	Devise actions to yield a desired outcome.
Prognosis	Predict the outcome of a given situation.
Remedy	Prescribe treatment for a problem.
Control	Regulate a process. May require interpretation, diagnosis, monitoring, planning, prognosis, and remedies.

Table 1.4 Classic Chemistry Expert Systems

Name	Chemistry
CRYSALIS	Interpret a protein's 3-D structure.
DENDRAL	Interpret molecular structure.
TQMSTUNE	Remedy Triple Quadruple Mass Spectrometer (keep it tuned).
CLONER	Design new biological molecules.
MOLGEN	Design gene-cloning experiments.
SECS	Design complex organic molecules.
SPEX	Plan molecular biology experiments.

Table 1.5 Classic Electronics Expert Systems

Name	Electronics
ACE	Diagnose telephone network faults.
IN-ATE	Diagnose oscilloscope faults.
NDS	Diagnose national communication net.
EURISKO	Design 3-D microelectronics.
PALLADIO	Design and test new VLSI circuits.
REDESIGN	Redesign digital circuits to new.
CADHELP	Instruct for computer-aided design.
SOPHIE	Instruct circuit fault diagnosis.

Table 1.6 Classic Medical Expert Systems

Name	Medicine
PUFF	Diagnose lung disease.
VM	Monitors intensive-care patients.
ABEL	Diagnose acid-base/electrolytes.
AI/COAG	Diagnose blood disease.
AI/RHEUM	Diagnose rheumatoid disease.
CADUCEUS	Diagnose internal medicine disease.
ANNA	Monitor digitalis therapy.
BLUE BOX	Diagnose/remedy depression.
MYCIN	Diagnose/remedy bacterial infections.
ONCOCIN	Remedy/manage chemotherapy patients.
ATTENDING	Instruct in anesthetic management.
GUIDON	Instruct in bacterial infections.

Table 1.7 Classic Engineering Expert Systems

Name	Engineering
REACTOR	Diagnose/remedy reactor accidents.
DELTA	Diagnose/remedy GE locomotives.
STEAMER	Instruct operation of steam powerplant.

Table 1.8 Classic Geology Expert Systems

Name	Geology
DIPMETER	Interpret dipmeter logs.
LITHO	Interpret oil well log data.
MUD	Diagnose/remedy drilling problems.
PROSPECTOR	Interpret geological data for minerals.

Table 1.9 Classic Computer Expert Systems

Name	Computer Systems
PTRANS	Give prognosis for managing DEC computers.
BDS	Diagnose bad parts in switching net.
XCON	Configure DEC computer systems.
XSEL	Configure DEC computer sales order.
XSITE	Configure customer site for DEC computers.
YES/MVS	Monitor/control IBM MVS operating system.
TIMM	Diagnose DEC computers.

Appropriate Domains for Expert Systems

Before starting to build an expert system, it is essential to decide if an expert system is the appropriate paradigm. For example, one concern is whether an expert system should be used instead of an alternative paradigm such as conventional programming. The appropriate domain for an expert system depends on a number of factors:

- *Can the problem be solved effectively by conventional programming?* If the answer is yes, then an expert system is not the best choice. For example, consider the problem of diagnosing some equipment. If all the symptoms for all malfunctions are known in advance, then a simple table lookup or decision tree of the fault is adequate. Expert systems are best suited for situations in which there is no efficient algorithmic solution. Such cases are called **ill-structured problems** and reasoning may offer the only hope of a good solution.

As an example of an ill-structured problem, consider the case of a person who cannot decide where to book a vacation online and decides to see a travel agent. Table 1.10 lists some characteristics of an **ill-structured problem** as indicated by the person's responses to the travel agent's questions.

Although this is an extreme case, it does illustrate the basic concept of an ill-structured problem. As you can see, an ill-structured problem would not lend itself well to an algorithmic solution because there are so many possibilities. In this case, a default option should be exercised when all else fails. For example, the travel agent could say, "Ah ha! I have the perfect trip for you: a round-the-world cruise. Please fill out the following credit card application and everything will be taken care of."

Table 1.10 Example of an Ill-Structured Problem

Travel Agent's Questions	Responses
Can I help you?	I'm not sure.
Where do you want to go?	Somewhere.
Any particular destination?	Here and there.
How much can you afford?	I don't know.
Can you get some money?	I don't know.
When do you want to go?	Sooner or later.

In dealing with ill-structured problems, there is a danger that the expert system design may accidentally develop into an algorithmic solution. If there is a good algorithm, there is no need for an expert system. A clue that this has happened occurs if a solution is found that requires a rigid **control structure**. That is, the rules are forced to execute in a certain sequence by the knowledge engineer explicitly setting the priorities of many rules. Forcing a rigid control structure on the expert system cancels a major advantage of expert systems technology, which is dealing with unexpected input that does not follow a predetermined pattern. That is, expert systems react opportunistically to their input, whatever it is. Conventional programs generally expect input to follow a certain sequence. An expert system with a lot of control often indicates a disguised algorithm and may be a good candidate for recoding as a conventional program.

- *Is the domain well bounded?* It is very important to have well-defined limits on what the expert system is expected to know and what its capabilities should be. For example, suppose you wanted to create an expert system to diagnose headaches. Certainly medical knowledge of a physician would be put in the knowledge base. However, for a deep understanding of headaches, you might also put in knowledge about neurochemistry, then its parent area of biochemistry, then chemistry, molecular biophysics, and so forth, perhaps down to subnuclear physics. Other domains such as biofeedback, psychology, psychiatry, physiology, exercise, yoga, and stress management may also have pertinent knowledge about headaches. The point of all this is—when do you stop adding domains? The more domains, the more complex the expert system becomes.

In particular, the task of coordinating all the expertise becomes a major task. In the real world, we know from experience how difficult it is to have coordinated teams of experts working on problems, especially when they come up with conflicting recommendations. If we knew how to program well the coordination of expertise, then we could try programming an expert system to have the knowledge of multiple experts. The first attempts made to coordinate multiple expert systems were the HEARSAY II and HEARSAY III systems. These projects demonstrated the complexity of the task. As a familiar example, try bringing your car into multiple service centers when you have a problem and note how many disparate diagnoses you get. As expertise becomes more scarce, so do the number of opinions, which makes it very difficult to decide on a course of action.

- *Is there a need and a desire for an expert system?* Although it's great experience to build an expert system, it's rather pointless if no one is willing to use it. If there already are many human experts, it's difficult to justify an expert system based on the reason of scarce human expertise. Also, if the experts or users don't want the system, it will not be accepted even if there is a need for it. For example, many simulations have shown AI-controlled traffic lights can cut gas consumption by 50 percent, but it would also reduce gasoline tax revenues to city, state, and Federal governments by 50 percent.

Management especially must be willing to support the system. This is even more critical for expert systems than conventional programs because deployment of expert systems is sometimes viewed as a precursor to downsizing the workforce. Workers must be reassured that the expert system will not lead to job loss but to increased profitability as expertise becomes more widely available at a lower cost. The area of expert systems deserves more support because it attempts to solve the problems that cannot be done by conventional programming. The risks are greater but so are the rewards.

- *Is there at least one human expert who is willing to cooperate?* There must be an expert who is willing, and preferably enthusiastic, about the project. Not all experts are willing to have their knowledge examined for faults and then put into a computer. Even if there are multiple experts willing to cooperate in the development, it might be wise to limit the number of experts involved in development. Different experts may have different ways of solving a problem, such as requesting different diagnostic tests. Sometimes they may even reach different conclusions. Trying to code multiple methods of problem-solving in one knowledge base may create internal conflicts and incompatibilities.
- *Can the expert explain the knowledge so that it is understandable by the knowledge engineer?* Even if the expert is willing to cooperate, there may be difficulty in expressing the knowledge in explicit terms. As a simple example of this difficulty, can you explain in words how you move a finger? Although you could say it's done by contracting a muscle in the finger, the next question is—how do you contract a finger muscle? The other difficulty in communication between expert and knowledge engineer is that the knowledge engineer doesn't know the technical terms of the expert. This problem is particularly acute with medical terminology. It may take a year or longer for the knowledge engineer to even understand what the expert is talking about, let alone translate that knowledge into explicit computer code.
- *Is the problem-solving knowledge mainly heuristic and uncertain?* Expert systems *are* appropriate when the expert's knowledge is largely heuristic and uncertain. That is, the knowledge may be based on experience, called

experiential knowledge, and the expert may have to try various approaches in case one doesn't work. In other words, the expert's knowledge may be a trial-and-error approach, rather than one based on logic and algorithms. However, the expert can still solve the problem faster than someone who is not an expert. This is a good application for expert systems. If the problem can be solved simply by logic and algorithms, it is better solved by a conventional program.

1.8 LANGUAGES, SHELLS, AND TOOLS

A fundamental decision in defining a problem is deciding how best to model it. Sometimes experience is available to aid in choosing the best paradigm. For example, experience suggests that a payroll is best done using conventional procedural programming. Experience also suggests that it is preferable to use a commercial package, if available, rather than writing one from scratch. A general guide to selecting a paradigm is to consider the most traditional one first—conventional programming. The reason for doing this is because of the vast amount of experience we have with conventional programming and the wide variety of commercial packages available. If a problem cannot be effectively solved by conventional programming, then turn to nonconventional paradigms such as AI, which requires knowledge of theory, rather than a single course in C.

Like the standard relational database language SQL, an expert system language is a higher-order language than third-generation languages like LISP or C because it is easier to do certain things; but there is also a smaller range of problems that can be addressed. That is, the specialized nature of expert systems languages makes them very suitable for writing expert systems but not for general-purpose programming. In many situations, it is even necessary to exit temporarily from an expert systems language to perform a function in a procedural language. CLIPS is designed to make this easy.

The primary functional difference between expert systems languages and procedural languages is the focus of representation. Procedural languages focus on providing flexible and robust techniques to represent data. For example, data structures such as arrays, records, linked lists, stacks, queues, and trees are easily created and manipulated. Modern languages such as Java and C# are designed to aid in **data abstraction** by providing structures for encapsulation such as objects, methods, and packages. This provides a level of abstraction that is then implemented in the program. The data and methods to manipulate the data are tightly interwoven in objects. In contrast, expert system languages focus on providing flexible and robust ways to represent knowledge. The expert system paradigm allows two levels of abstraction: data abstraction and **knowledge abstraction**. Expert system languages specifically separate the data from the methods of manipulating the data. An example of this separation is that of facts (data abstraction) and rules (knowledge abstraction) in a rule-based expert system language. In addition, CLIPS provides objects and all the features of a true object-oriented language.

This difference in focus also leads to a difference in program design methodology. Because of the tight interweaving of data and knowledge in procedural languages, programmers must carefully describe the sequence of execution. However, the explicit separation of data from knowledge in expert system languages requires considerably less rigid control of execution sequence. Typically, an entirely separate piece of code, the inference engine, is used to apply the knowledge to the data. This separation of knowledge and data allows a higher degree of parallelism and modularity.

The customary way of defining the need for an expert system program is to decide if you want to program the expertise of a human expert. If such an expert exists and will cooperate, then an expert system approach may be successful. Likewise, a very knowledge-intensive tasks with uncertainty may best be solved with an expert system tool.

The road to selecting an expert system tool is paved with confusion since there is such a rich variety to choose from today. While CLIPS does not have all the features of other languages, it is simpler to learn and is thus a good choice for an introductory textbook. In addition, CLIPS still maintains its original advantages of small program size and fast execution where real-time response is critical.

Besides the confusing choice of the many languages available today, the terminology used to describe the languages is confusing. Some vendors refer to their products as "tools," while others refer to "shells" and still others talk about "integrated environments." For clarity in this book, the terms are defined as follows:

- **language**—A translator of commands written in a specific syntax. An expert system language will also provide an inference engine to execute the statements of the language. Depending on the implementation, the inference engine may provide forward chaining, backward chaining, or both. Under this language definition, LISP is not an expert system language while PROLOG is. However, it is possible to write an expert system language using LISP, and write AI in PROLOG. For that matter you can even write an expert system or AI language in assembly language. Questions of development time, convenience, maintainability, efficiency, and speed determine what language software is written in.
- **tool**—A language plus associated utility programs to facilitate the development, debugging, and delivery of application programs. Utility programs may include text and graphics editors, debuggers, file management, and even code generators. Some tools may even allow the use of different paradigms such as forward and backward chaining in one application.

In some cases, a tool may be integrated with all its utility programs in one environment to present a common interface to the user. This approach minimizes the need for the user to leave the environment to perform a task. For example, a simple tool may not provide facilities for file management and so a user would have to exit the tool to give conventional commands in a C host language, for example. An integrated environment allows easy exchange of data between utility programs in the environment. Some tools do not even require the user to write any code. Instead, the tool allows a user to enter knowledge by examples from tables or spreadsheets and generates the appropriate code itself.

- **shell**—A special-purpose tool designed for certain types of applications in which the user must supply only the knowledge base. The classic example of this is the EMYCIN (empty MYCIN) shell. This shell was made by removing the medical knowledge base of the MYCIN expert system.

MYCIN was designed as a backward-chaining system to diagnose disease. By simply removing the medical knowledge, EMYCIN was created as a shell containing knowledge about other kinds of consultative systems that use backward chaining. The EMYCIN shell demonstrated the reusability of the essential MYCIN software such as the inference engine and user interface. This was a very important step in the development of modern expert system technology because it meant that an expert system would not have to be built from scratch for each new application. Today the field of expert systems tools is quite competitive with many features and GUIs available.

There are many ways of characterizing expert systems such as representation of knowledge, forward or backward chaining, support of uncertainty, hypothetical reasoning, explanation facilities, and so forth. Unless a person has built a number of expert systems, it is difficult to appreciate all of these features, especially those found in the more expensive tools. The best way to learn expert systems technology is to develop a number of systems with an easy-to-learn language and then invest in a more sophisticated tool if you need its features.

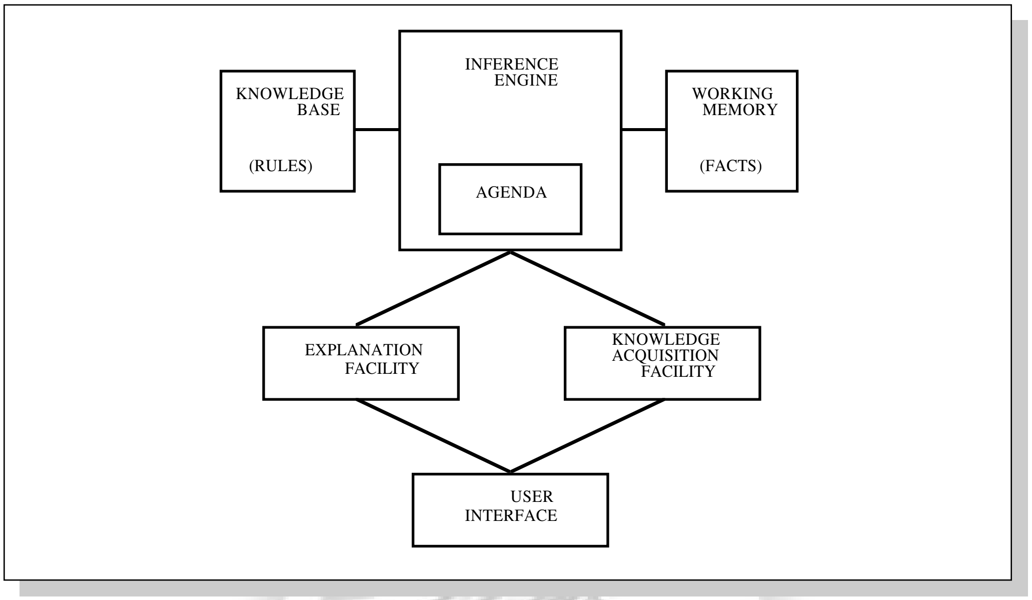
1.9 ELEMENTS OF AN EXPERT SYSTEM

The elements of a typical expert system are shown in Figure 1.6. In a rule-based system, the knowledge base contains the domain knowledge needed to solve problems coded in the form of rules. While rules are a popular paradigm for representing knowledge, other types of expert systems use different representations, as discussed in Chapter 2.

An expert system consists of the following components:

- **user interface**—the mechanism by which the user and the expert system communicate.
- **explanation facility**—explains the reasoning of the system to a user.
- **working memory**—a **global database** of facts used by the rules.
- **inference engine**—makes inferences by deciding which rules are satisfied by facts or objects, prioritizes the satisfied rules, and executes the rule with the highest priority.
- **agenda**—a prioritized list of rules created by the inference engine, whose patterns are satisfied by facts or objects in working memory.
- **knowledge acquisition facility**—an automatic way for the user to enter knowledge in the system rather than by having the knowledge engineer explicitly code the knowledge.

Figure 1.6 Structure of a Rule-Based Expert System



The knowledge acquisition facility is an optional feature on many systems. In some expert systems tools, the tool can learn by rule induction through examples and automatically generate rules. Other methods such as ID3, C 4.5, C 5.1, artificial neural networks, and genetic algorithms have been used in machine learning to generate rules. The major problem with machine learning to generate rules is that there is no explanation as to why this was created. Unlike a human who can explain the reason for a rule, machine learning systems have never been able to explain their actions and so unpredictable results may occur. However, the examples are generally from tabular- or spreadsheet-type data better suited to decision trees. General rules constructed by a knowledge engineer can be much more complex than the simple rules from rule induction.

Depending on the implementation of the system, the user interface may be a simple text-oriented display or a sophisticated high-resolution, bit-mapped display. High-resolution displays are commonly used to simulate a control panel with dials and displays.

The knowledge base is also called the **production memory** in a rule-based expert system. As a very simple example, consider the problem of deciding to cross a street. The productions for the two rules are as follows, where the arrows mean that the system will perform the actions on the right of the arrow if the conditions on the left are true:

```
the light is red → stop
the light is green → go
```

The production rules can be expressed in an equivalent pseudocode IF-THEN format as:

```
Rule: Red_light
IF
      the light is red
THEN
      stop

Rule: Green_light
IF
        the light is green
THEN
        go
```

Each rule is identified by a name. Following the name is the IF part of the rule. The section of the rule between the IF and THEN part of the rule is called by various names such as the **antecedent**, **conditional part**, **pattern part,** or **left-hand-side** (**LHS**). The individual condition:

```
the light is green
```

is called a **conditional element** or a **pattern**.

The following are some examples of rules from the classic systems:

```
MYCIN system for diagnosis of meningitis and
  bacteremia (bacterial infections)
IF
     The site of the culture is blood, and
     The identity of the organism is not known with
        certainty, and
     The stain of the organism is gramneg, and
     The morphology of the organism is rod, and
     The patient has been seriously burned
THEN
     There is weakly suggestive evidence (.4) that
        the identity of the organism is
        pseudomonas

XCON/R1 for configuring DEC VAX computer systems
IF
      The current context is assigning devices to
        Unibus modules and
      There is an unassigned dual-port disk drive
        and
      The type of controller it requires is known
        and
      There are two such controllers, neither of
        which has any devices assigned to it, and
```

```
          The number of devices that these controllers
             can support is known
      THEN
          Assign the disk drive to each of the
             controllers, and
          Note that the two controllers have been
             associated and each supports one drive
```

In a rule-based system, the inference engine determines which rule antecedents, if any, are satisfied by the facts. Two general methods of inferencing are commonly used as the problem-solving strategies of expert systems: **forward chaining** and **backward chaining**. Other methods used for more specific needs may include means-ends analysis, problem reduction, backtracking, plan-generate-test, hierarchical planning and the least commitment principle, and constraint handling.

Forward chaining is reasoning from facts to the conclusions resulting from those facts. For example, if you see that it is raining before leaving home (the fact), then you should take an umbrella (the conclusion).

Backward chaining involves reasoning in reverse from a hypothesis, a potential conclusion to be proved, to the facts that support the hypothesis. For example, if you have not looked outside and someone enters with wet shoes and an umbrella, your hypothesis is that it is raining. In order to support this hypothesis, you could ask the person if it was raining. If the response is yes, then the hypothesis is proven true and becomes a fact. As mentioned before, a hypothesis can be viewed as a fact whose truth is in doubt and needs to be established. The hypothesis can then be interpreted as a goal to be proven.

Depending on the design, an inference engine will do either forward or backward chaining or both. For example, CLIPS is designed for forward chaining, PROLOG performs backward chaining, and the version of CLIPS called Eclipse developed by Paul Haley does both forward and backward chaining. The choice of inference engine depends on the type of problem. Diagnostic problems are better solved with backward chaining while prognosis, monitoring, and control are better done by forward chaining. However, there is a memory size and execution speed penalty involved in using a more enhanced tool since more code is involved in creating the tool. CLIPS was designed to be "lean and mean" so that it would execute as fast as possible and be suitable for deployed applications in small memory devices such as ROM. Although people who use desktop computers with 512 megs of RAM don't normally think about this, developing a product such as an intelligent remote control or microwave oven requires ROM for cheap permanent memory. The cost of memory goes down with size so it is desirable to use as little memory as possible in consumer devices to be competitive. CLIPS can actually generate a C code executable that is small and easily burnt into a small ROM.

The working memory may contain facts regarding the current status of the traffic light such as "the light is green" or "the light is red." Either or both of these facts may be in working memory at the same time. If the traffic light is working normally, only one fact will be in memory. However, it is possible that both facts may be in working memory if there is a malfunction in the light. Notice the difference between the knowledge base and working memory. Facts do not interact with one another. The fact "the light is green" has no effect on the fact "the light is red." Instead, our knowledge of traffic lights says that if both facts are simultaneously present, then there is a malfunction in the light.

If there is a fact "the light is green" in working memory, the inference engine will notice that this fact satisfies the conditional part of the green light rule and put this rule on the agenda. If a rule has multiple patterns, then all of its patterns must be simultaneously satisfied for the rule to be placed on the agenda. Some patterns may even be satisfied by specifying the absence of certain facts in working memory.

A rule whose patterns are all satisfied is said to be **activated** or **instantiated**. Multiple activated rules may be on the agenda at the same time. In this case, the inference engine must select one rule for **firing**. The term *firing* comes from neurophysiology, the study of the way the nervous system works. An individual nerve cell or **neuron** emits an electrical signal when stimulated. No amount of further stimulation can cause the neuron to fire again for a short time period. This phenomenon is called **refraction**. Rule-based expert systems are built using refraction in order to prevent trivial loops. That is, if the green light rule kept firing on the same fact over and over again, the expert system would never accomplish any useful work.

Various methods have been invented to provide refraction. In one type of expert system language, OPS5, each fact is given a unique identifier called a **timetag** when it is entered in working memory. After a rule has fired on a fact, the inference engine will not fire on that fact again because its time stamp has been used too recently.

Following the THEN part of a rule is a list of **actions** to be executed when the rule fires. This part of the rule is known as the **consequent** or **right-hand side** (**RHS**). When the red light rule fires, its action "stop" is executed. Likewise, when the green light rule fires, its action is "go." Specific actions usually include the addition or removal of facts from working memory or printing results. The format of these actions depends on the syntax of the expert system language. For example, in CLIPS, the action to add a new fact called "stop" to working memory would be (assert stop). Because of the LISP ancestry of CLIPS, parentheses are required around patterns and actions.

The inference engine operates in **recognize-act cycles**. Various names have been given to describe this such as **select-execute cycle**, **situation-response cycle**, and **situation-action cycle**. By any name for a cycle, the inference engine will repeatedly execute a group of tasks until certain criteria cause execution to cease. The general tasks are shown in the following pseudocode as **conflict resolution**, **act**, **match**, and **check for halt**:

```
WHILE not done
```

Conflict Resolution: If there are activations, then select the one with highest priority, else done.

Act: Sequentially perform the actions on the RHS of the selected activation. Those that change working memory have immediate effect in this cycle. Remove the activation that has just fired from the agenda.

Match: Update the agenda by checking if the LHS of any rules are satisfied. If so, activate them. Remove activations if the LHS of their rules are no longer satisfied.

Check for Halt: If a halt action is performed or break command given, then done.

```
 END-WHILE
```

Accept a new user command

Multiple rules may be activated and put on the agenda during one cycle. Also, activations will be left on the agenda from previous cycles unless they are deactivated because their LHS is no longer satisfied. Thus the number of activations on the agenda will vary as execution proceeds. Depending on the program, an activation may always be on the agenda but never selected for firing. Likewise some rules may never become activated. In these cases, the purpose of these rules should be reexamined because the rules are either unnecessary or their patterns were not correctly designed.

The inference engine executes the actions of the highest priority activation on the agenda, then the next highest priority activation, and so on until no activations are left. Various priority schemes have been designed into expert system tools. Generally, all tools let the knowledge engineer define the priority of rules.

Agenda conflicts occur when different activations have the same priority and the inference engine must decide on one rule to fire. Different shells have different ways of dealing with this problem. In the original Newell and Simon paradigm, those rules entered first in the system had the highest default priority. In OPS5, rules with more complex patterns automatically have a higher priority. In CLIPS, rules have the same default priority unless assigned different ones by the knowledge engineer.

At this time, control is returned to the **top-level** command interpreter for the user to give further instructions to the expert system shell. The top-level is the default mode in which the user communicates with the expert system, and is indicated by the task "Accept a new user command." It is the top-level that accepts the new command.

The top-level is the user interface to the shell while an expert system application is under development. More sophisticated user interfaces are usually designed for the expert system to facilitate its operation. For example, the expert system may have a user interface for control of a manufacturing plant that shows a block diagram of the plant. Warnings and status messages may appear in flashing colors with simulated dials and gauges. In fact, more effort may go into the design and implementation of the user interface than in the expert system knowledge base, especially in a prototype. Depending on the capabilities of the expert system shell, the user interface may be implemented by rules or in another language called by the expert system. For example, the Java version of CLIPS, called Jess, created by Ernest Friedman-Hill (Friedman-Hill 03), makes it easy to call Java classes to easily display a GUI since Java has many objects designed for this purpose.

A key feature of an expert system is an explanation facility that allows the user to ask how the system came to a certain conclusion and why certain information is needed. The question of how the system came to a certain conclusion is easy to answer in a rule-based system since a history of the activated rules and contents of working memory can be maintained in a stack. This is not readily available in artificial neural networks, genetic algorithms, or other systems that evolve. Although attempts have been made to provide an explanation capability is some systems, they cannot match the clarity of a human-design expert system. Sophisticated explanation facilities may allow the user to ask *what if* questions to explore alternate reasoning paths through hypothetical reasoning.

1.10 PRODUCTION SYSTEMS

The most popular type of expert system today is the rule-based system. Instead of representing knowledge in a relatively declarative, static way (as a number of things that are true), rule-based systems represent knowledge in terms of multiple rules that specify what should or should not be concluded in different situations. A rule-based system consists of IF-THEN *rules*, *facts*, and an *interpreter* that controls which rule is invoked depending on the facts in working memory.

There are two broad kinds of rule systems: forward chaining and backward chaining. A forward-chaining system starts with the known initial facts and keep using the rules to draw new conclusions or take certain actions. A backward-chaining system starts with some hypothesis or goal you are trying to prove, and keeps looking for rules that would allow the hypothesis to be proven true. New subgoals may be created to break up the large problem into smaller pieces that can be more easily proved. Forward-chaining systems are primarily data-driven, while backward-chaining systems are goal-driven (Debenham 98). There will be more extensive discussion of chaining in a later chapter and a complete example of how backward-chaining can be done in CLIPS.

Rules are popular for a number of reasons:

- *Modular nature*. This makes it easy to encapsulate knowledge and expand the expert system by incremental development.
- *Explanation facilities*. It is easy to build explanation facilities with rules since the antecedents of a rule specify exactly what is necessary to activate the rule. By keeping track of which rules have fired, an explanation facility can present the chain of reasoning that led to a certain conclusion.
- *Similarity to the human cognitive process*. Based on the work of Newell and Simon, rules appear to be a natural way of modeling how humans solve problems. The simple IF-THEN representation of rules make it easy to explain to experts the structure of the knowledge that you are trying to elicit from them.

Rules are a type of production whose origins go back to the 1940s. Because of the importance of rule-based systems, it is worthwhile to examine the development of the rule concept. This will give you a better idea of why rule-based systems are so useful for expert systems.

Post Production Systems

Production systems were first used in symbolic logic by Post who originated the name. He proved the important and amazing result that any system of mathematics or logic could be written as a certain type of production rule system. This result established the great capability of production rules for representing major classes of knowledge rather than being limited to a few types. Under the term **rewrite rules**, they are also used in linguistics as a way of defining the grammar of a language. Computer languages are commonly defined using the Backus-Naur Form (BNF) of production rules, like CLIPS in Appendix D.

The basic idea of Post was that any mathematical or logic system is simply a set of rules specifying how to change one string of symbols into another set of symbols. That is, given an input string, the antecedent, a production rule could produce a new string, the consequent. This idea is also valid with programs and expert systems where the initial string of symbols is the input data and the output string is some transformation of the input.

As a very simple case, if the input string is “patient has fever” the output string might be “take an aspirin.” Note that there is no meaning attached to these strings. That is, the manipulations of the strings are based on syntax and not any semantics or understanding of what a fever, aspirin, and patient represent. A human knows what these strings mean in terms of the real world mean but a Post production system is just a way of transforming one string into another. A production rule for this example could be:

```
Antecedent → Consequent
person has fever →take aspirin
```

where the arrow indicates the transformation of one string into another. We can interpret this rule in terms of the more familiar IF-THEN notation as:

```
IF person has fever THEN take aspirin
```

The production rules can also have multiple antecedents. For example:

```
person has fever AND
      fever is greater than 102 →see doctor
```

Note that the special connective AND is not part of the string. The AND indicates that the rule has multiple antecedents.

A Post production system consists of a group of production rules, such as the following (where the numbers in parentheses are for our discussion):

```
(1)  car won't start →check battery
(2)  car won't start →check gas
(3)  check battery AND battery bad →replace
      battery
(4)  check gas AND no gas →fill gas tank
```

If there is a string "car won't start," the rules (1) and (2) may be used to generate the strings "check battery" and "check gas." However, there is no control mechanism that applies both these rules to the string. Only one rule may be applied, both or none. If there is another string "battery bad" and a string "check battery," then rule (3) may be applied to generate the string "replace battery."

Unlike a conventional programming language such as C or C+, there is no special significance to the order in which rules are written. The rules of our example could have been written in the following order and it would still be the same system:

```
(4)   check gas AND no gas → fill gas tank
(2)   car won't start → check gas
(1)   car won't start → check battery
(3)   check battery AND battery bad → replace
       battery
```

Although Post production rules were useful in laying part of the foundation of expert systems, they are not adequate for writing practical programs. The basic limitation of Post production rules for programming is lack of a **control strategy** to guide the application of the rules. A Post system permits the rules to be applied on the strings in any manner because there is no specification given on how the rules should be applied.

As an analogy, suppose you go to the library to find a certain book on expert systems. At the library, you start randomly looking at books on the shelves for the one you want. If the library is fairly large, it may take a long time to find the book you need. Even if you find the section of books on expert systems, your next random choice could take you to an entirely different section, such as French cooking. The situation becomes even worse if you need material from the first book to help you determine the second book that you need to find. A random search for the second book will also take a long time.

Markov Algorithms

The next advance in applying production rules was made by Markov, who specified a control structure for production systems. A **Markov algorithm** is an ordered group of productions that are applied in order of priority to an input string. If the highest priority rule is not applicable, then the next one is applied, and so forth. The Markov algorithm terminates if either, (1) the last production is not applicable to a string or, (2) a production that ends with a period is applied.

Markov algorithms can also be applied to substrings of a string, starting from the left. For example, the production system consisting of the single rule:

```
AB → HIJ
```

when applied to the input string GABKAB produces the new string GHIJKAB. Since the production now applies to the new string, the final result is GHIJKHIJ.

The special character $\wedge$ represents the **null string** of no characters. For example, the production:

```
A → ∧
```

deletes all occurrences of the character A in a string.

Other special symbols represent any single character and are indicated by lowercase letters a, b, c, … x, y, z. These symbols represent single-character variables and are an important part of modern expert system languages. For example, the rule:

```
AxB → BxA
```

will reverse the characters A and B, where *x* is any single character.

The Greek letters α, β, and so forth are used for special punctuation of strings. The Greek letters are used because they are distinct from the alphabet of ordinary letters.

An example of a Markov algorithm that moves the first letter of an input string to the end is shown following. The rules are ordered in terms of highest priority (1), next highest (2), and so forth. The rules are prioritized in the order that they are entered:

```
(1)  αxy →yαx
(2)  α  →∧
(3)  ∧→α
```

For the input string ABC, the execution trace is shown in Table 1.11.

Table 1.11 Execution Trace of a Markov Algorithm

Rule	Success or Failure	String
1	F	ABC
2	F	ABC
3	S	αABC
1	S	BαAC
1	S	BCαA
1	S	BCAα
2	S	BCA

Notice that the α symbol acts analogously to a temporary variable in a conventional programming language. However, instead of holding a value, the α is used as a place holder to mark the progression of changes in the input string. Once its job is done, the α is eliminated by rule 2. Hidden Markov Models (HMM) are widely used in pattern recognition applications, most notably speech recognition.

The Rete Algorithm

Notice that there is a definite control strategy to Markov algorithms with higher-priority rules ordered first. As long as the highest priority rule applies, it is used.

If not, the Markov algorithm tries lower-priority ones. Although the Markov algorithm can be used as the basis of an expert system, it is very inefficient for systems with many rules. The problem of efficiency becomes of major importance if we want to create expert systems for real problems containing hundreds or thousands of rules. No matter how good everything else is about a system, if a user has to wait a long time for a response, the system will not be used. What we really need is an algorithm that knows about all the rules and can apply any rule without having to try each rule sequentially.

A solution to this problem is the **Rete Algorithm** developed by Charles L. Forgy at Carnegie-Mellon University in 1979 for his Ph.D. dissertation on the OPS (Official Production System) expert system shell. The term *Rete* comes from the Latin word *rete,* which means net. The Rete Algorithm functions like a net in holding a lot of information so that much faster response times and rule firings can occur compared to a large group of IF-THEN rules which must be checked one by one in a conventional program. The Rete Algorithm is a dynamic data structure like a standard B+ tree which reorganizes itself to optimize search.

The Rete Algorithm is a very fast pattern-matcher that obtains its speed by storing information about the rules in a network in memory. The Rete Algorithm is intended to improve the speed of forward-chained rule systems by limiting the effort required to recompute the conflict set after a rule is fired. Its drawback is that it has high memory space requirements, but this is not a problem these days in which memory is so cheap. The Rete takes advantage of two empirical observations that were used to come up with its data structure:

Temporal Redundancy—The firing of a rule usually changes only a few facts, and only a few rules are affected by each of those changes.
Structural Similarity—The same pattern often appears in the left-hand side of more than one rule.

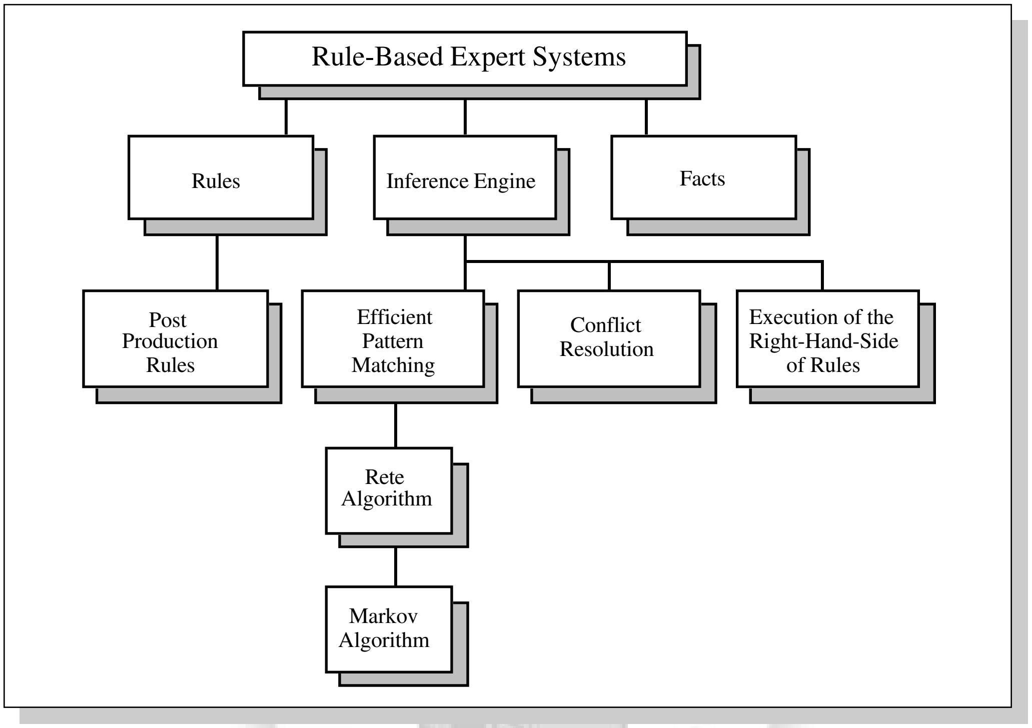
If you have hundreds or thousands of rules, it would be very inefficient for the computer to sequentially check whether each rule is likely to fire. The Rete Algorithm made expert system tools practical on the slow computers of the 1970s. Today the Rete network is still important for fast execution when many rules are involved in an expert system.

Instead of having to match facts against every rule on every recognize-act cycle, the Rete algorithm looks only for changes in matches on every cycle. This greatly speeds up the matching of facts to antecedents since the static data that doesn't change from cycle to cycle can be ignored. This topic will be discussed further in the chapters on CLIPS. Fast pattern matching algorithms such as the Rete completed the foundation for the practical application of expert systems. Figure 1.7 summarizes the foundations of modern rule-based expert system technologies.

1.11 PROCEDURAL PARADIGMS

Programming paradigms can be classified as procedural and nonprocedural. Figure 1.8 shows a **taxonomy** or classification of the procedural paradigms in terms of languages. Figure 1.9 shows a taxonomy for nonprocedural paradigms.

Figure 1.7 Foundations of Modern Rule-Based Expert Systems



These figures illustrate the relationship of expert systems to other paradigms and should be considered a general guide rather than strict definitions. In particular, although CLIPS is shown as rule-based, it is possible to write an entirely object-oriented expert system in CLIPS, or a hybrid system using both rules and objects. Some of the paradigms and languages have characteristics that may place them in more than one class. For example, some consider functional programming procedural, while others consider it declarative.

An **algorithm** is a method of solving a problem in a finite number of steps. The implementation of an algorithm in a program is a **procedural program**. The terms *algorithmic programming*, *procedural programming*, and *conventional*

Figure 1.8 Procedural Languages

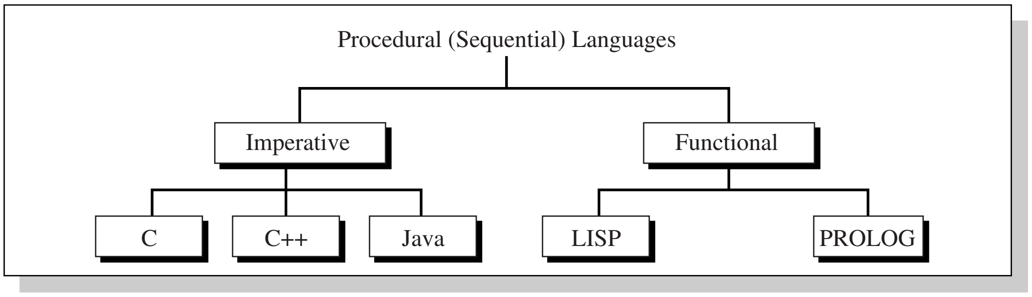
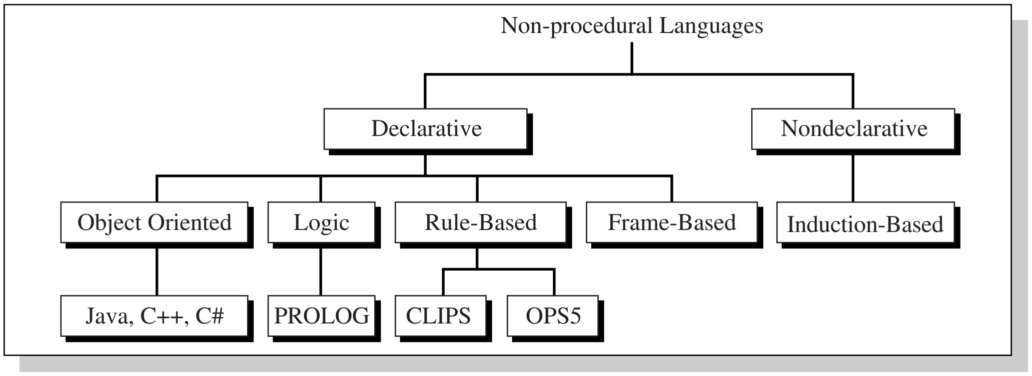


Figure 1.9 Nonprocedural Languages



programming are often used synonymously to mean *non-AI programs*. A common conception of a procedural program is that it proceeds sequentially, statement by statement, unless a branch instruction is encountered. Another often used synonym for procedural program is **sequential program**. However, the term sequential programming implies too much constraint since all modern programming languages support recursion and so programs may not be strictly sequential.

The distinguishing feature of the procedural paradigm is that the programmer must specify exactly *how* a problem solution must be coded. Even code generators must produce procedural code. In a sense, the use of code generators is **nonprocedural** programming because it removes most or all of the procedural code writing from the programmer. The goal of nonprocedural programming is to have the programmer specify *what* the goal is and let the system determine *how to* accomplish it.

Imperative Programming

The terms *imperative* and *statement-oriented* are used synonymously. A language such as C has the dominant characteristic that statements are imperatives or commands to the computer telling it what to do. Note that in an object-oriented version C++, objects may also be used and the general concept is that of objects passing messages to each other. Thus the execution of an object-oriented program is more similar to an event-driven system rather than an imperative one in which program flow is assumed to execute sequentially from beginning to end.

The imperative programming paradigm is an abstraction of real computers which in turn are based on the Turing machine and the von Neumann machine with its registers and store (memory). At the heart of these machines is the concept of a modifiable store. Variables and assignments are the programming language analog of the modifiable **store**. The *store* is the object that is manipulated by the program. Imperative programming languages have a rich variety of commands to provide structure to code and to manipulate the store. Each imperative programming language defines a particular view of hardware. These views are so distinct that it is common to speak of a Pascal virtual machine, or a Java virtual machine that executes bytecodes that will execute on any hardware platform. This feature of bytecode portability accounted for much of the success of Pascal in the 1970s and 80s until Java used the same system. In fact, a compiler implements the virtual machine defined by the programming language in the language supported by the actual hardware and operating system.

In imperative programming, a name may be assigned to a value and later reassigned to another value. The collection of names and the associated values and the location of control in the program constitute the **state**. The state is a logical model of storage that is an association between memory locations and values. During execution, a program can be characterized as making a transition from the initial state to a final state by passing through a sequence of intermediate states. The transition from one state to the next is determined by assignment operations, input, and sequencing commands.

Imperative languages developed as a way of freeing the programmer from coding assembly language in the von Neumann architecture. Consequently, imperative languages offer great support to variables, assignment operations, and repetition. These are all low-level operations that modern languages attempt to hide by providing features such as recursion, procedures, modules, packages, and so forth. Imperative languages are also characterized by their emphasis on rigid control structure and their associated **top-down** program designs.

A serious problem with all languages is the difficulty of proving the correctness of programs which means the program is valid (discussed more in Chapter 2). From the AI standpoint, another serious problem is that imperative languages are not very efficient symbol manipulators. Because the imperative language architecture was molded to fit the von Neumann computer architecture, we have languages that can support number-crunching very well but not symbolic manipulation. However, imperative languages such as C and object-oriented languages such as C++ and Java have been used as the underlying base language to write expert system shells. These languages and the shells built from them run more efficiently and quickly on common general-purpose computers than the early shells built using LISP that required special hardware expressly designed for LISP.

Because of their sequential nature, imperative languages are not very efficient for directly implementing expert systems, especially rule-based ones. As an illustration of this problem, consider the task of encoding the information of a real-world problem with hundreds or thousands of rules. For example, the classic XCON system used to configure computer systems had about 7000 rules in its knowledge base. Early unsuccessful attempts were made to code this program in FORTRAN and BASIC before settling on the successful expert systems approach. An expert system tool like CLIPS makes it much easier to create such a large rule base than programming in a 3rd generation language or object-oriented language like Java, C++, or C#. To fully appreciate the advantages of an expert system language, you need to try coding in it. Programming in CLIPS is covered in the second part of this book.

The direct way of coding this knowledge in an imperative language would require 7000 IF-THEN statements or a very, very long CASE. This style of coding would present major efficiency problems since all 7000 rules would need to be searched for matching patterns on every recognize–act cycle. Note that the inference engine and its recognize–act cycle would also have to be coded in the imperative language. However the situation is much more complex than merely writing 7000 rules since many of the rules are triggered by other rules. For example, in the case of stopping for a red light, this would only apply if you were approaching the light; not if you had passed it. Many other interactions are possible in a rule-based system. Some rules assert facts, some may retract facts, some may modify facts. More sophisticated rules may create new rules or delete rules in the case of machine learning.

The efficiency of the program could be improved if rules were ordered so that those most likely to be executed were put at the beginning. However, this would require considerable tuning of the system and would change as new rules are added, or old ones deleted and modified. A better method for improving efficiency would be to build a dynamic network in memory of the rule patterns to reduce search time in determining which rules should be activated. Rather than making the programmer manually construct the tree, it should preferably be built automatically by the computer, based on the pattern and action syntax of the IF-THEN rules. It would also be helpful to have an IF-THEN syntax that was more conducive to representing knowledge and had powerful pattern matching tests. This requires the development of a parser to analyze input structure and an interpreter or compiler to execute the new IF-THEN syntax.

When all of these techniques for improving efficiency are implemented, the result is a dedicated expert system. If the inference engine, parser, and interpreter are removed to provide easy development of other expert systems, they compose an expert system shell. Of course, instead of doing all of this development from scratch, today it is much easier to use an existing tool like CLIPS, that is well-documented and extensively tested.

Functional Programming

The nature of **functional programming**, as exemplified by languages such as LISP, is very different from statement-oriented languages with their heavy reliance on elaborate control structures and top-down design. Conventional languages place conceptual limits on the way problems can be modularized. Functional languages push those limits back. Two features of functional languages in particular, higher-order functions and lazy evaluation, can contribute greatly to modularity. The fundamental idea of functional programming is to combine simple functions to yield more powerful functions. This is essentially a **bottom-up** design in contrast to the common **top-down** designs of imperative languages.

Functional programming is centered around **functions**. Mathematically, a function is an **association** or rule that maps members of one set, the **domain**, into another set, the **codomain**. The following is an example of a function definition:

```
cube(x) ≡ x*x*x, where x is a real number and
cube is a function with real values
```

The three parts of the function definition are:

(1) the association, *x*x*x*
(2) the domain, real numbers
(3) codomain, real numbers

of the cube function. The symbol ≡ means "is equivalent to" or "is defined as." The following notation is a shorthand way of writing that the cube mapping is from the domain of real numbers, symbolized as $\Re$, to the codomain of real numbers:

```
cube:ℜ → ℜ
```

A general notation for a function *f* that maps from a domain *S* to a domain *T* is $f{:}S \rightarrow T$. The **range** of the function *f* is the set of all **images** *f(s)* where *s* is an element of *S*. For the case of the cube function, the images of *s* are *s*s*s* and the range is the set of all real numbers. The range and codomain are the same for the cube function. However, this may not be true for other functions such as the square function, *x*x*, with domain and codomain of real numbers. Since the range of the square function is only nonnegative real numbers, the range and codomain are not the same.

Using set notation, the range of a function can be written as:

```
R ≡{f(s) | s ∈ S}
```

The **curly braces { }** denote a **set**. The **bar**, |, is read as "where." The previous statement can be read that the range *R* is equivalent to the set of values *f(s)* where every element *s* is in the set *S*. The association is a set of ordered pairs *(s,t)*, where $s \in S$, $t \in T$, and $t=f(s)$. Every member of *S* must have one and only one element of *T* associated with it. However, multiple *t* values may be associated with a single *s*. As a simple example, every positive number *n* has two square roots, $\pm\sqrt{n}$.

Functions may also be defined recursively as in:

```
factorial(n) ≡ n*factorial(n-1)
      where n is an integer and
      factorial is an integer function
```

Recursive functions are commonly used in functional languages such as LISP.

Mathematical concepts and expressions are **referentially transparent** because the meaning of the whole is completely determined from its parts. No synergism is involved between the parts. For example, consider the functional expression *x+(2*x)*. The result is obviously *3*x*. Both *x+(2*x)* and *3*x* give the same result no matter what values are substituted for *x*. Even other functions can be substituted for *x* and the result is the same. For example, let *h(y)* be some arbitrary function. Then *h(y) + (2 * h(y))* would still be equivalent to *3*h(y)*.

Now consider the following assignment statement in an imperative computer language such as C:

```
    sum = f(x) + x
```

If the parameter *x* is passed by reference and its value is changed in the function call, *f(x)*, what value will be used for *x*? Depending on how the compiler is written, the value of *x* might be the original value if it was saved on a stack, or the new value if *x* was not saved. Another source of confusion occurs if one compiler evaluates expressions right-to-left while another evaluates left-to-right. In this case, *f(x)+x* would not evaluate the same as *x+f(x)* on different compilers even if the same language was used. Other side effects may occur due to global variables. Thus, unlike mathematical functions, program functions are not referentially transparent.

Functional programming languages were created to be referentially transparent. Five parts make up a functional language:

- **data objects** for the language functions to operate on
- **primitive functions** to operate on the data objects
- **functional forms** to synthesize new functions from other functions
- **application operations** on functions that return a value and
- **naming procedures** to identify new functions.

Functional languages are generally implemented as interpreters for ease of construction and immediate user response.

In LISP (LISt Processing), data objects are **symbolic expressions** (S-expressions) that are either **lists** or **atoms**. The following are examples of lists and are shown because the style is similar to how patterns are programmed in CLIPS:

```
    (milk eggs cheese)
    (shopping (groceries (milk eggs cheese) clothes
(pants)))
    ()
```

Patterns like this may be programmed in the conditional or **left-hand-side** (**LHS**) of a rule and represent facts or lists within lists as is the case of the "shopping" example. If the conditional is true, the **right-hand-side** (**RHS**) of the rule is executed and new lists made. Other things may be programmed to occur on the right-hand-side of rules such as retracting facts. Lists are always enclosed in matching parentheses with spaces separating the elements. The elements of lists can be atoms, such as milk, eggs, and cheese, or embedded lists such as (milk eggs cheese) and (pants). Lists can be split up but atoms cannot. The **empty list**, (), contains no elements and is called **nil**.

The original version of LISP was called *pure LISP* because it was purely functional. However, it was also not very efficient for writing programs. Non-functional additions have been made to LISP to increase the efficiency of writing programs. For example, SET acts as the assignment operator, while LET and PROG can be used to create local variables and execute a sequence of S-expressions. Although they act like functions, they are not functional in the original mathematical sense.

Since its creation, LISP has been the leading AI language in the United States. Many of the original expert system shells were written in LISP because it is so easy to experiment with LISP. However, conventional computers do not execute LISP very efficiently and execute the shells built using LISP even more inefficiently. Of course, as processors and clock rates have improved, so has the performance of LISP.

This problem of high cost has an impact on both the development and the **delivery problem**. It is not enough just to develop a great program if it cannot be delivered for use because of high cost. A good development workstation is not necessarily a good delivery vehicle due to speed, power, size, weight, environmental, or cost constraints. Some applications may even require that the final code be placed in ROM for reasons of cost and nonvolatility. Putting code into ROM can be a problem with some AI and expert systems tools that require special hardware to run. It's better to consider this possibility in advance rather than have to recode a program later.

An additional problem is that of embedding AI with conventional programming languages such as C, C++, C#, and Java to create intelligent systems. Applications that require extensive number crunching are best done in conventional languages rather than in an expert systems tool. This is why CLIPS makes it easy for you to write your own functions in the host language. For more details on this and other advanced topics, see the *CLIPS Reference Manual* available online at (http://www.ghgcorp.com/clips/CLIPS.html).

Unless special provisions are made, expert systems that are written in LISP are generally difficult to embed in anything other than LISP programs. One major consideration in selecting an AI language should be the language in which the tool is written. For reasons of portability, efficiency, and speed, many expert systems tools are now being written in or converted to C. This also eliminates the problem of requiring expensive special hardware for LISP-based applications.

1.12 NONPROCEDURAL PARADIGMS

Nonprocedural paradigms do not depend on the programmer giving exact details for how a problem is to be solved. This is the opposite of the procedural paradigms which specify *how* a function or statement sequence computes. In nonprocedural paradigms, the emphasis is on specifying *what* is to be accomplished and letting the system determine how to accomplish it.

Declarative Programming

The **declarative paradigm** separates the **goal** from the methods used to achieve the goal. The user specifies the goal while the underlying mechanism of the implementation tries to satisfy the goal. A number of paradigms and associated programming languages have been created to implement the declarative model.

Object-Oriented Programming

The **object-oriented** paradigm is another case of a paradigm that can be considered partly imperative and partly declarative. Today, the term *object-oriented* is used programming languages such as C++, Java, and C#. The basic idea is to design a program by considering the data used in the program as objects and then implementing operations on those objects. This is the opposite of top-down design which proceeds by stepwise refinement of a program's control structure. The Unified Modeling Language (UML) is a popular method for object-oriented design.

As an example of object-oriented design, consider the task of writing a program to manage a charge account with an interactive menu. The important data objects are current balance, amount of charge, and amount of payment. Various methods can be defined to act on the data objects once the appropriate classes have been defined. These operations would be add charge, make payment, and add monthly interest. Once all data objects, operations, and the menu interface are defined, coding can begin. This object-oriented design methodology is well-suited to a program that has a weak control structure. It would not be as suitable in a program that requires a strong control structure such as a payroll application. However OOP is popular today because of maintenances and code reuse of objects.

The term **object-oriented programming** was originally used for languages such as Smalltalk, which was specifically designed for objects. Sometimes the term is used to mean programming of an object-oriented design even in a language that has no true object support.

Modern object-oriented languages such as C++, Java, and C# have features to support objects built into the language. Smalltalk is descended from SIMULA 67, a language developed for simulation. SIMULA 67 introduced the concept of **class** which led to the concept of information hiding so that a programmer need only know how to use an object and not the inner details of how it was programmed. A class is not a type. An **instance** of a class is a data object that can be manipulated. The term *instance* has carried over to expert systems where it denotes a fact that matches a pattern. Likewise, a rule is said to be instantiated if its LHS is satisfied. The terms *activated* and *instantiated* in rule-based systems are synonymous.

Another significant concept that came from SIMULA 67 is **inheritance** in which a **subclass** could be defined to inherit the properties of classes. For example, one class may be defined to consist of objects that can be used in a stack and another class defined to be complex numbers. Now a subclass can be easily defined as objects that are complex numbers used in a stack. That is, these objects have inherited properties from both classes above them, called **superclasses**. The concept of inheritance can be extended to organize objects in a hierarchy where objects can inherit from their classes, which can inherit from their classes, and so on. Inheritance is very useful since objects can inherit properties from their classes without the programmer having to specify every property. Implementing multiple inheritance is not easy which is why Java and C# do not have it. However since CLIPS is written in C++ it takes advantage of multiple inheritance. The full object-oriented language embedded within CLIPS is called **COOL** (**Common Object-Oriented Language**). The operation of COOL is transparent to the user so you just have to understand the features of object-oriented programming.

Logic Programming

One of the first AI applications of computers was in proving logic theorems with the Logic Theorist program of Newell and Simon. This program was first reported at the Dartmouth Conference on AI in 1956 and caused a sensation because electronic computers previously had been used only for numeric calculations. Now a computer was mechanically reasoning the proofs of mathematical theorems, which had been a task that only mathematicians were thought capable of doing. The term *mechanical* means automated and dates back to Babbage's Mechanical computer of the nineteenth century.

In the Logic Theorist and its successor, the General Problem Solver (GPS), Newell and Simon concentrated on trying to implement powerful algorithms that could solve any problem. While the Logic Theorist was meant only for mathematical theorem proving, GPS was designed to solve any kind of logic problem, including games and puzzles such as chess, Tower of Hanoi, Missionaries and Cannibals, and cryptarithmetic. An example of their famous cryptarithmetic (secret arithmetic) puzzle is:

```
   DONALD
 + GERALD
   ROBERT
```

and it is known that D=5. The object is to figure out the arithmetic values of the other letters in the range 0–9.

GPS was the first problem solving program to clearly separate the problem-solving knowledge from the domain knowledge. This paradigm of explicitly separating the problem-solving knowledge from the domain knowledge is now used as the basis of expert systems. In expert systems today, the inference engine decides what knowledge should be used and how it should be applied.

Efforts continued to improve mechanical theorem proving. By the early 1970s, it had been discovered that computation is a special case of mechanical, logical deduction. When backward chaining was applied to sentences of the form "conclusion if conditions," it was powerful enough for significant theorem proving. The conditions can be thought of as representing patterns to be matched as in production rules discussed earlier. Sentences expressed in this form are called **Horn clauses** after Alfred Horn, who first investigated them. In 1972, the language PROLOG was created by Kowalski, Colmerauer, and Roussel to implement logic programming by backward chaining using Horn clauses.

Backward chaining can be used both to express the knowledge in a declarative representation and also control the reasoning process. Typically, backward chaining proceeds by defining smaller **subgoals** that must be satisfied if the initial goal is to be satisfied. These subgoals are then further broken down into smaller subgoals and so forth.

An example of declarative knowledge is the following classic example:

```
All men are mortal
Socrates is a man
```

which can be expressed in the Horn clauses:

```
someone is mortal
  if someone is a man
Socrates is a man
  if (in all cases)
```

For the sentence about Socrates, the IF condition is true in all cases. In other words, the knowledge about Socrates does not require any pattern to match. Contrast this with the mortal case in which someone must be a man for the pattern of the IF condition to be satisfied.

Notice that a Horn clause can be interpreted as a procedure that tells how to satisfy a goal. That is, to determine if someone is mortal, it is necessary to determine if someone is a man. As a slightly more complex example:

```
A car needs gas, oil, and inflated tires to run
```

which can be expressed in a Horn clause as:

```
x is a car and runs
  if x has gas and
  if x has oil and
  if x has inflated tires
```

Notice how the problem of determining if a car will run has been reduced to three simpler subproblems or subgoals. Now suppose there is some additional declarative knowledge as follows:

```
The fuel gauge shows not-empty if a car has gas
The dipstick shows not-empty if a car has oil
The air pressure gauge shows at least 20 if a car
  has inflated tires
The fuel gauge shows not-empty
The dipstick reads empty
The air pressure gauge shows 15
```

These can be translated into the following Horn clauses:

```
x has gas
  if the fuel gauge shows not-empty
x has oil
  if the dipstick shows not-empty
x has inflated tires
  if the air pressure gauge shows at least 20
```

```
Fuel gauge is not empty
  if (in all cases)
Dipstick reads empty
Air pressure is 15
  if (in all cases)
```

From these clauses, a mechanical theorem prover can prove that the car will not run because there is no oil and insufficient air pressure.

One of the advantages of backward-chaining systems is that execution can proceed in parallel. That is, if multiple processors were available, they could work on satisfying subgoals simultaneously. PROLOG is more than just a language since it incorporates a backward-chaining inference engine. At a minimum, PROLOG is a shell since it requires:

- an interpreter or inference engine
- a database (facts and rules)
- a form of pattern matching called **unification**
- a **backtracking** mechanism to pursue alternate subgoals if a search to satisfy a goal is unsuccessful.

As an example of backward chaining, suppose you can pay for the oil to make the car run if you have cash or a credit card. One subgoal is checked to see if you have cash. If there is no fact that you do have cash, the backtracking mechanism will then explore the other subgoal to see if you have a credit card. If you have a credit card, the goal of paying for oil can be satisfied. Notice that the absence of a fact to prove a goal is just as effective, although perhaps less efficient, than a negative fact such as "Dipstick reads empty." Either negative facts or missing facts can cause a goal to be unsatisfied (except in political logic.)

If the backtracking and pattern-matching mechanisms are not needed by the problem, then the programmer must work around them or code in a different language. One of the advantages of logic programming is executable specifications. That is, specifying the requirements of a problem by Horn clauses produces an executable program. This is very different from conventional programming in which the requirements document does not look at all like the final executable code.

Unlike production rule systems, the order in which subgoals, facts, and rules are entered in a PROLOG program has significant effects. Efficiency, and therefore speed, are affected by the way that PROLOG searches its database. Furthermore, there are programs that execute correctly if subgoals, facts, and rules are entered one way but go into an infinite loop or have a run-time error if the order changes.

Expert Systems

Expert systems can be considered **declarative programming** because the programmer does not specify how a program is to achieve its goal at the level of an algorithm. For example, in a rule-based expert system, any of the rules may become activated and put on the agenda if its LHS matches the facts. The order that the rules were entered does not affect which rules are activated. Thus, the program statement order does not specify a rigid control flow. Other types of expert systems are based on **frames**, discussed in Chapter 2, and **inference nets** discussed in Chapter 4.

There are a number of differences between expert systems and conventional programs. Table 1.12 lists some differences.

Table 1.12 Some Differences Between Conventional Programs and Expert Systems

Characteristic	Conventional Program	Expert System
Control by . . .	Statement order	Inference engine
Control and data	Implicit integration	Explicit separation
Control Strength	Strong	Weak
Solution by . . .	Algorithm	Rules and inference
Solution search	Small or none	Large
Problem solving	Algorithm is correct	Rules
Input	Assumed correct	Incomplete, incorrect
Unexpected input	Difficult to deal with	Very responsive
Output	Always correct	Varies with problem
Explanation	None	Usually
Applications	Numeric, file, and text	Symbolic reasoning
Execution	Generally sequential	Opportunistic rules
Program design	Structured design	Little or no structure
Modifiability	Difficult	Reasonable
Expansion	Done in major jumps	Incremental

Expert systems are normally used to deal with uncertainty. The uncertainty may arise in the input data to the expert system and even the knowledge base itself. At first this may seem surprising to people used to conventional programming. However, much of human knowledge is heuristic, which means that it may only work correctly part of the time so we try something else. In addition, the input data may be incorrect, incomplete, inconsistent, and have other errors. Algorithmic solutions are not capable of dealing with situations like this because an algorithm guarantees the solution of a problem in a finite series of steps.

Depending on the input data and the knowledge base, an expert system may come up with the correct answer, a good answer, a bad answer, or no answer at all. While this may seem shocking at first, the alternative is to have *no* answer all of the time. Again, the important thing to keep in mind is that a good expert system will perform no worse than the best problem solver for problems like this—a human expert—and may do better. If we knew an efficient algorithmic method that was better than an expert system, we would use it. The important thing is to use the best, and perhaps only, tool for the job.

Nondeclarative Programming

Nondeclarative paradigms are very popular in PROLOG and SQL. New versions of PROLOG have been developed for AI_1 and SQL is the standard for relational databases. They are used for a wide variety of applications, from standalone applications or in conjunction with other paradigms.

Induction-Based Programming

An application of AI that has attracted a great deal of interest is **induction-based** programming such as the classic ID3 Algorithm for machine learning and the newer C4.5 and C5.1. In this paradigm, the program learns by generalizing from a sample, just like a person would from the sequence 2,4,6,"?". One application to which this paradigm has been applied is database access. Instead of the user having to type in the specific values for one or more fields for a search, it is necessary to select only one or more appropriate example fields with those characteristics. The database program infers the characteristics of the data and searches the database for a match. Oracle and other relational database management systems which use the structured English query language (SQL) are good examples of using pattern recognition in searches.

Some expert systems tools offer induction learning by which they accept examples and case studies and automatically generate rules (see Appendix G.)

1.13 ARTIFICIAL NEURAL SYSTEMS

In the 1980s, a new development in programming paradigms arose called **artificial neural systems** (**ANS**), which is based on one way the brain processes information. This paradigm is sometimes called **connectionism** because it models solutions to problems by training simulated neurons connected in a network. Today ANS are found in many applications ranging from face recognition, medical diagnosis, games, speech recognition to car engines.

Connectionist researchers conjecture that thinking about computation in terms of the brain metaphor rather than the digital computer metaphor will lead to insights into the nature of intelligent behavior. The idea behind connectionism, then, is that we may see significant advances in AI if we approach problems from the point of view of brain-style computation rather than rule-based symbol manipulation. Neural networks are an information processing technique based on the way biological nervous systems, such as the brain, process information. The fundamental concept of neural networks is the structure of the information processing system. Composed of a large number of highly interconnected processing elements or neurons, a neural network system uses the human-like technique of learning by example to resolve problems. The neural network is configured for a specific application, such as data classification or pattern recognition, through a learning process called **training**. Just as in biological systems, learning involves adjustments to the synaptic connections that exist between the neurons.

There are many ways to classify the types of ANS but a useful distinction is whether a training set of input and output data is provided or not. If a training set is provided, the ANS is called a supervised model. If it has to learn to classify input without knowing any output, it is called unsupervised. A good example of supervised ANS is in face recognition. In contrast, if you do not know what the output should be, an ANS serves as a good classifier to group input as in the case of identifying outbreaks of disease.

Neural networks can differ on the way their neurons are connected, the specific kinds of computations their neurons do, the way they transmit patterns of activity throughout the network, and the way and rate at which they learn. Neural networks have been applied to all kinds of real-world problems. Their primary advantage is that they can solve problems that are too complex for conventional technologies—problems that do not have an algorithmic solution or for which an algorithmic solution is too complex to be defined. In general, neural networks are well suited to problems that people are good at solving, but cannot explain how they do it. These problems include pattern recognition and forecasting which require the recognition of trends in data. Today the field of **data mining** makes extensive use of AI to seek patterns in historical data that may serve to guide a company in the future. For example, data mining could be used to show a company when to stock certain kinds of items because of a seasonal variation.

Neural nets are also used as the front end of expert systems that require massive amounts of input from sensors as well as real-time response. Over 50 free ANS are available for download from the FAQ of the newsgroup comp.ai.neural-nets, and other resources mentioned in Appendix G.

The Traveling Salesman Problem

ANS has had remarkable success in providing real-time response to complex pattern recognition problems. In the 1980s, a neural net running on an ordinary microcomputer obtained a very good solution to the Traveling Salesman problem in 0.1 seconds compared to the optimum solution that required an hour of CPU time on a mainframe. The Traveling Salesman problem is important because it is the classic problem faced in optimizing the routing of signals in a telecommunications system. Optimizing routing is important in minimizing the travel time, and thus efficiency and speed whether it is routing packets through the Internet or delivering parcels in the mail to many locations

The basic Traveling Salesman problem is to compute the shortest route through a given list of cities. Table 1.13 shows the possible routes for one to four cities. Notice that the number of routes is proportional to the factorial of the number of cities minus one, $(N - 1)!$.

Table 1.13 Traveling Salesman Problem Routes

Number of Cities	Routes
1	1
2	1–2–1
3	1–2–3–1
	1–3–2–1
4	1–2–3–4–1
	1–2–4–3–1
	1–3–2–4–1
	1–3–4–2–1
	1–4–2–3–1
	1–4–3–2–1

While there are 9! = 362880 routes for 10 cities, there are 29! = 8.8E30 possible routes for thirty cities. The Traveling Salesman problem is a classic example of **combinatorial explosion** because the number of possible routes increases so rapidly that there are no practical solutions for realistic numbers of cities. If it takes one hour of CPU time to solve for 30 cities, it will take 30 hours for 31 cities, and 330 hours for 32 cities. These are actually very small numbers when compared to the thousands of telecommunications switches and cities that are used in routing of data packets and real items.

A neural net can solve the 10-city case just as fast as the 30-city case while a conventional computer takes much longer. For the 10-city case, the neural net came up with one of the two best routes and for the 30-city case, it came up with one of the best 100,000,000 routes. This is more impressive if it is realized that this route is in the top 1E-22 of the best solutions. Although neural nets may not always give the optimum answer, they can provide a best guess in real time. In many cases, a 99.999999999999999999% correct answer in 0.1 second is better than a 100% correct answer in 30 hours. Other techniques to solve the problem have used genetic algorithms, as shown in Appendix G, and the Evolutionary Art Algorithm (Dorigo 04). Another approach is real DNA (Sipper 02).

Elements of an ANS

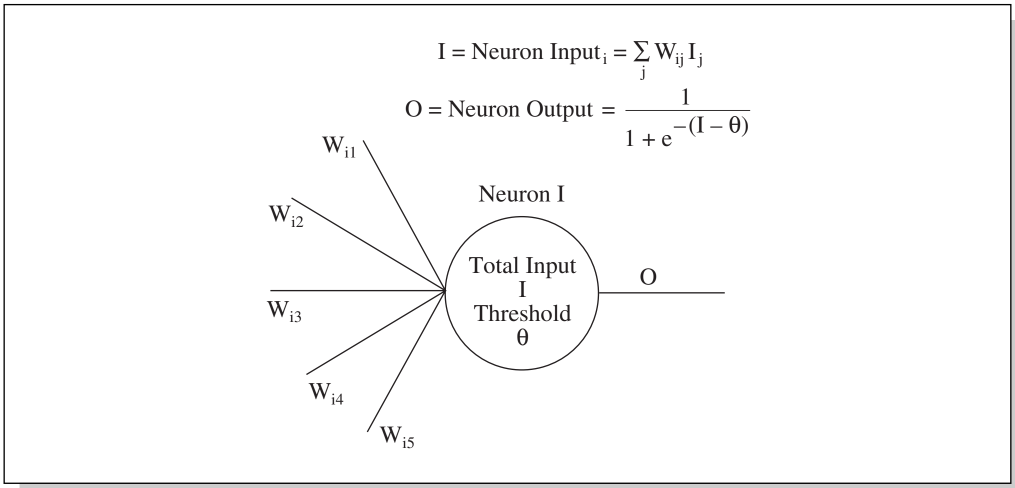
An **ANS** can be thought of as an analog computer that uses simple processing elements connected in a highly parallel manner. The processing elements perform very simple Boolean or arithmetic functions on their inputs. The key to the functioning of an ANS is the **weights** associated with each element. The weights represent the information stored in the system.

A typical artificial neuron is shown in Figure 1.10. The neuron may have multiple inputs but only one output. The human brain contains about 10^{11} neurons and one neuron may have thousands of connections to another. The input signals to the neuron are multiplied by the weights and are summed to yield the total neuron input, *I*. The weights can be represented as a matrix and identified by subscripts.

The neuron output is often taken as a **sigmoid function** of the input. The sigmoid is representative of real neurons, which approach limits for very small and very large inputs. The sigmoid is called an **activation function** and a commonly used function is $(1 + e^{-X})^{-1}$. Each neuron also has an associated **threshold value**, θ, which is subtracted from the total input, *I*. Figure 1.11 shows an ANS that can compute the **exclusive-OR** (**XOR**) of its inputs using a technique called **back propagation**. The XOR gives a true output only when its inputs are not all true or not all false. The number of nodes in the hidden layer will vary depending on the application and design.

Neural nets are not programmed in the conventional sense. There are many neural net learning algorithms, such as **counter propagation** and back propagation, to train nets. The programmer "programs" the net by simply supplying the input and corresponding output data. The net learns by automatically adjusting weights in the network which connect the neurons. The weights and the threshold

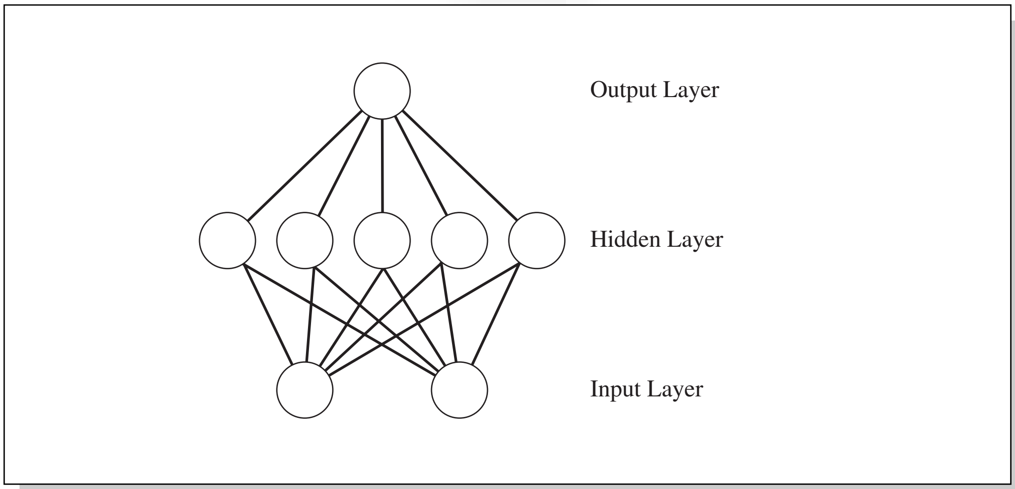
Figure 1.10 Neuron Processing Element


$I = \text{Neuron Input}_i = \sum_j W_{ij} I_j$
$O = \text{Neuron Output} = \frac{1}{1 + e^{-(I-\theta)}}$
Neuron I
Total Input I Threshold θ
O
W_{i1}
W_{i2}
W_{i3}
W_{i4}
W_{i5}

values of neurons determine the propagation of data through the net and so its correct response to the training data. Training the net to the correct responses may take hours or days, depending on the number of patterns that the net must learn and the necessary hardware and software. However, once the learning is accomplished, the net responds very quickly.

If software simulation is not fast enough, ANS can be fabricated in chips for real-time response. Once the network has been trained and the weights determined, a chip can be constructed.

Figure 1.11 A Back-Propagation Net



Characteristics of ANS

ANS architecture is very different from conventional computer architecture. In a conventional computer it is possible to correlate discrete information with memory cells. For example, a Social Security number could be stored as ASCII code in a contiguous group of memory cells. By examining the contents of this contiguous group the Social Security number could be directly reconstructed. This reconstruction is possible because there is a one-to-one relationship between each character of the Social Security number and the memory cell that contains the ASCII code of that character.

ANSs are modeled after current brain theories, in which information is represented by the weights. However, there is no direct correlation between a specific weight and a specific item of stored information. This distributed representation of information is similar to that of a hologram in which the lines of the hologram act as a diffraction grating to reconstruct the stored image when laser light is passed through.

A neural net is a good choice when there is much empirical data and no algorithm exists that provides sufficient accuracy and speed. ANS offers several advantages compared to the storage of conventional computers:

- *Storage is **fault tolerant***. Portions of the net can be removed and there is only a degradation in quality of the stored data. This occurs because the information is stored in a distributed manner.
- *The quality of the stored image degrades gracefully in proportion to the amount of net removed*. There is no catastrophic loss of information. The storage and quality features are also characteristic of holograms.
- *Data is stored naturally in the form of **associative memory***. An associative memory is one in which partial data is sufficient to recall all of the complete stored information. This contrasts with conventional memory in which data is recalled by specifying the address of the data to be recalled. A partial or noisy input may still elicit the complete original information.
- *Nets can extrapolate and interpolate from their stored information*. Training teaches a net to look for significant features or relationships in the data. Afterwards, the net can extrapolate to suggest relationships on new data. In one experiment a neural net was trained on the family relationships of 24 hypothetical people. Afterwards, the net could also answer correctly relationships about which it had not been trained.
- *Nets have **plasticity***. Even if a number of neurons are removed, the net can be retrained to its original skill level if enough neurons remain. This is also a characteristic of the brain, in which portions can be destroyed and the original skill levels can be relearned in time.

These characteristics make ANS very attractive for robot spacecraft, oil field equipment, underwater devices, process control, and other applications that need to function a long time without repair in a hostile environment. Besides the issue of reliability, ANS offers the potential of low maintenance costs because of plasticity. Even if hardware repair can be done, it will probably be more cost effective to reprogram the neural net than to replace it.

ANSs are generally not well suited for applications that require number crunching or an optimum solution. Also, if a practical algorithmic solution exists, an ANS is not a good choice unless the ANS is cheaper to implement than a chip.

Developments in ANS Technology

A central concern for connectionists is the idea that we must "take the brain seriously" in our psychological theorizing. This idea, of course, dates to ancient Greek speculations concerning the action of animal spirits in the nervous system. Later, notable speculations may be found in Rene Descartes' *Treatise of Man*. The origins of ANS started with the mathematical modeling of neurons by McCulloch and Pitts in 1943. An explanation of learning by neurons was given by Hebb in 1949. In Hebbian learning, a neuron's efficiency in triggering another neuron increases with firing. The term **firing** means that a neuron emits an electrochemical impulse which can stimulate other neurons connected to it. There is evidence that the conductivity of connections between neurons at their connections, called **synapses**, increases with firing. In ANS, the weight of connections between neurons is changed to simulate the changing conductance of natural neurons (Swingler 96).

In 1961, Rosenblatt published an influential book dealing with a new type of artificial neuron system he had been investigating called a **perceptron**. The perceptron was a remarkable device that showed capabilities for learning and pattern recognition. It basically consisted of two layers of neurons and a simple learning algorithm. The weights had to be manually set, in contrast to modern ANS, that set the weights themselves based on training. Many researchers entered the field of ANS and began studying perceptrons during the 1960s.

The early perceptron era came to an end in 1969 when Minsky and Papert published a book, called *Perceptrons*, that showed the theoretical limitations of perceptrons as a general computing machine. They pointed out a deficiency of the perceptron in being able to compute only 14 of the 16 basic logic functions, which means that a perceptron is not a general-purpose computing device. In particular, they proved a perceptron could not recognize the exclusive-OR. Although they had not seriously investigated multiple layer ANS, they gave the pessimistic view that multiple layers would probably not be able to solve the XOR problem. Government funding of ANS research ceased in favor of the symbolic approach to AI using languages such as LISP and algorithms. New methods of representing symbolic AI information by frames, invented by Minsky, became popular during the 1970s. Frames have evolved to modern scripts. Because of their simplicity, perceptrons and other ANSs are easy to construct with modern integrated circuit technology.

ANS research continued on a small scale in the 1970s. During the late 1970s, Geoffrey Hinton, James McClelland, David Rumelhart, Paul Smolensky, and other members of the Parallel Distributed Processing Research Group became interested in neural network theories of cognition. Their landmark book published in 1986, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition,* marked the return of connectionism as a significant theory of cognition. Hopfield put ANS on a firm theoretical foundation with the Hopfield Net and demonstrated how ANS could solve a wide variety of problems. The general structure of a Hopfield Net is shown in Figure 1.12. In particular, he showed how an ANS could solve the Traveling Salesman problem in constant time as compared to the combinatorial explosion encountered by conventional algorithmic solutions. An electronic circuit form of an ANS could solve the Traveling Salesman problem in 1 μ second or less. Other combinatorial optimization problems can easily be done by ANS such as the four-color map, the Euclidean match, and the transposition code.

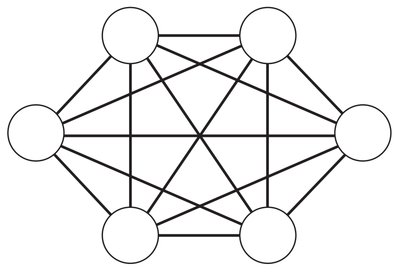
An ANS that can easily solve the XOR problem is the **back-propagation** net, also known as the **generalized delta rule**. The back-propagation net is commonly implemented as a three-layer net, although additional layers can be specified. The layers between the input and output layers are called **hidden layers** because only the input and output layers are visible to the external world. Another popular type of ANS is counterpropagation, invented by Hecht-Nielsen in 1986. An important theoretical result from mathematics, the Kolmogorov Theorem, can be interpreted as proving that a three-layer network with n inputs and 2n + 1 neurons in the hidden layer can map any continuous function.

Applications of ANS Technology

A significant example of learning by back propagation was demonstrated by a neural net that learned correct pronunciation of words from text. The ANS was trained by correcting its output using a text-to-speech device called DECTalk. It required 20 years of linguistic research to devise rules for correct pronunciation used by DECTalk. The ANS taught itself equivalent pronunciation skills overnight by simply listening to the correct pronunciation of speech from text. No linguistic skills were programmed into the ANS.

ANSs are used for recognition of radar targets by electronic and optical computers. New implementations of neural nets using optical components promise optical computers with speeds millions of times faster than electronic ones. Optical implementation of ANS is attractive because of the inherent parallelism

Figure 1.12 Hopfield Artificial Neural Net



of light. That is, light rays do not interfere with one another as they travel. Huge numbers of photons can easily be generated and manipulated by optical components such as mirrors, lenses, high-speed programmable spatial light modulators, arrays of optical bistable devices that can function as optical neurons, and diffraction gratings. Optical computers designed as ANS appear to be complementary to one another.

Classic ANS applications are discussed in (Giarratano 90a). In particular, ANSs are useful in control systems where conventional approaches are not satisfactory (Giarratano 91b). In fact, ANSs are widely used in a variety of industrial control systems (Hrycej 97) and in the links in Appendix G.

1.14 CONNECTIONIST EXPERT SYSTEMS AND INDUCTIVE LEARNING

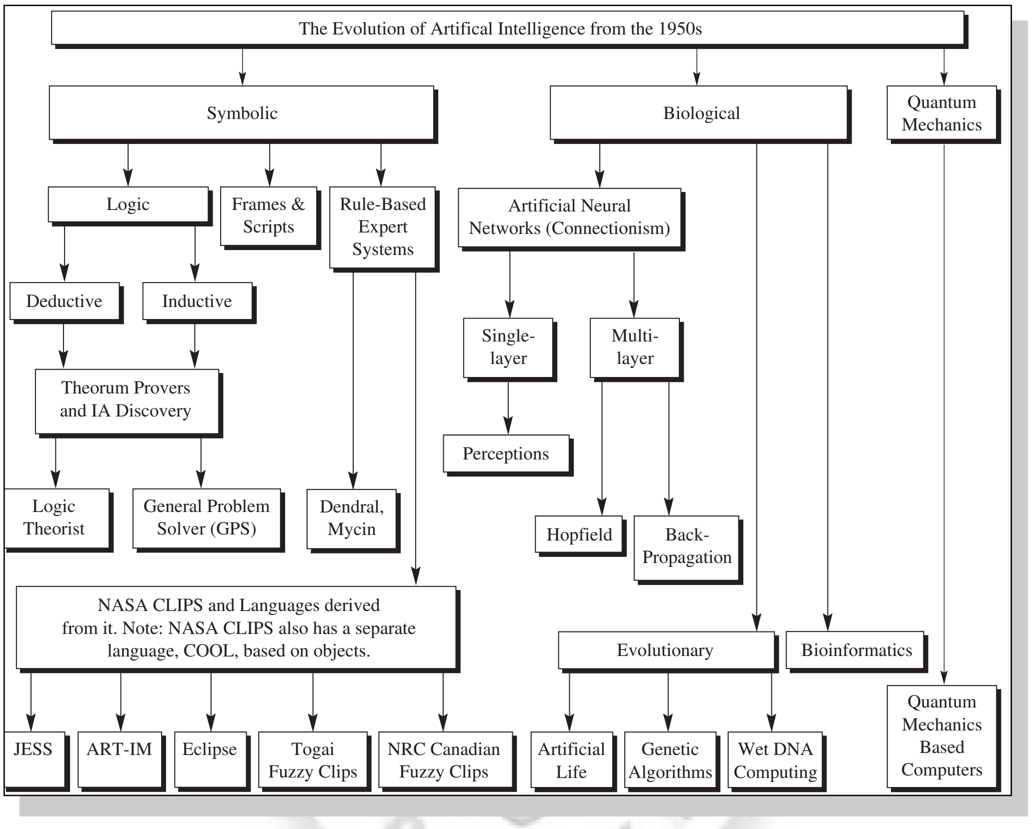
It is possible to build expert systems using ANS. In one system the ANS is the knowledge base constructed by training examples from medicine for diagnosis. This expert system tries to classify a disease from its symptoms into one of the known diseases that the system has been trained on. An inference engine called MACIE (Matrix Controlled Inference Engine) was designed that uses the ANS knowledge base. The system uses forward chaining to make inferences and backward chaining to query the user for any additional data needed to produce a solution. Although an ANS by itself cannot explain why its weights are set to certain values, MACIE can interpret the ANS and generate IF-THEN rules to explain its knowledge.

An ANS expert system such as this uses **inductive learning**. That is, the system **induces** the information in its knowledge base by example. Induction is the process of inferring the general case from the specific. Besides ANS, there are a number of commercially available decision software as discussed in Appendix G that explicitly generate rules from examples. The goal of inductive learning is to reduce or eliminate the knowledge acquisition bottleneck. By placing the burden of knowledge acquisition on the expert system, the development time may be reduced and the reliability may be increased if the system induces rules that were not known by a human. Expert systems have been combined with ANS (Giarratano 90b).

1.15 THE STATE OF THE ART IN ARTIFICIAL INTELLIGENCE

Many advances were made in AI during the 1990s and continuing on in the 21st Century. The strong AI view that the only good AI is that produced by logic and reasoning has had very limited success outside the closed world of the laboratory. Successful AI-based systems that have withstood the harsh reality of a tough, competitive marketplace have usually relied on biologically-based systems. Figure 1.13 shows the evolution of AI since the 1950s where the vertical scaledown indicates the passage of time. In the beginning, AI was divided into two broad paradigms based on models that were inspired by symbolic logic, with the other main branch inspired by biological processes. There has always been fierce competition between the two branches but based on experience constructing AI-based solutions, we can now say that no one approach will generally give the right

Figure 1.13 Evolution of Artificial Intelligence


The Evolution of Artifical Intelligence from the 1950s
Symbolic
Biological
Quantum Mechanics
Logic
Frames & Scripts
Rule-Based Expert Systems
Artificial Neural Networks (Connectionism)
Deductive
Inductive
Single-layer
Multi-layer
Theorum Provers and IA Discovery
Perceptions
Logic Theorist
General Problem Solver (GPS)
Dendral, Mycin
Hopfield
Back-Propagation
NASA CLIPS and Languages derived from it. Note: NASA CLIPS also has a separate language, COOL, based on objects.
Evolutionary
Bioinformatics
JESS
ART-IM
Eclipse
Togai Fuzzy Clips
NRC Canadian Fuzzy Clips
Artificial Life
Genetic Algorithms
Wet DNA Computing
Quantum Mechanics Based Computers

answer to a hard problem. The best we should hope for is an optimum solution, and be satisfied with a merely good solution.

The right part of the diagram in Figure 1.13 shows newer approaches based on physics, especially using quantum computers. While today we are thinking of using quantum computers to speed up searches only by many orders of magnitudes, that is a simplistic way of using these devices. When quantum mechanics was discovered in the 20th Century through Schroedinger's Equation and its equivalent formalism, Heisenberg's Matrix Mechanics, many people including Albert Einstein would not accept it. As Einstein was fond of saying, "God does not play dice with the Universe." Yet quantum mechanics is inextricably intertwined with probability and so unlike classical Newtonian theory, now nothing is certain. In fact, there is a theory that suggests consciousness is a quantum mechanical phenomena (Satinover 01).

A number of authors have speculated that the mind and consciousness itself may be an emergent property of the brain, which itself is built from neurons that in the final analysis are built from atoms and subatomic particles right on down to the quantum foam underlying all existence at distances so small, 10^{-60} meters, that space itself is grainy, not smooth as mathematics and classical

Newtonian physicists assume. At such small-distance scales, physicists have speculated that space contains not the usual 3 dimensions, but 11 dimensions. While it would be nice to think everything can be solved in terms of human reasoning and thinking, as philosophers have done since the time of the ancient Greeks 2500 years ago, thinking without real experimentation ignores the real world and leads to such conclusions as the following:

Premise: I am not moving because I don't feel myself moving.
Premise: I see the sun rise and set.
Therefore the sun is moving around me.

The term **emergence** has a special meaning when discussing evolutionary algorithms. Emergence is the term used for an unexpected behavior to arise that could not have been predicted in advance. For example, running water in a creek may appear to flow very smoothly until a rock is dropped in the water. Depending on the speed of the water, size of the rock, and other factors, turbulence may occur. The water exhibits nonlinear behavior with whitewater, rip currents, and other dynamic behavior coming into play that was not predicted by simple fluid dynamics but requires second- or higher-order differential equations to describe. The solution of these equations analytically is often not possible; therefore, approximate numerical solutions are the best that can be obtained. Many important examples of emergent behavior have been identified and applications to cities, management, and other widely disparate fields have been obtained (Johnson 01).

A classic example of emergent behavior is that of an ant colony in which a distributed type of intelligence, called **swarm intelligence,** is at work (Kennedy 01). This is in contrast to the centralized intelligence that humans and other mammals exhibit but which works well with colony type insects. The application of swarm intelligence is being studied for establishing robust robot colonies on other planets where each individual robot may not have much capacity for intelligence, thus saving cost and power, but the colony as a whole may prosper. Other applications to factories, networks, and distribution systems appear promising. In fact the Internet itself can be viewed as a swarm intelligence which uses the distributed low-level intelligence of packet switches to route packets with a high degree of survivability. This has met well one of the original goals of the Internet (or Arpanet as it was originally called) when started in the 1960s.

The major new player that has arisen in AI is Evolutionary algorithms (Fogel 03). These are generally used with other AI techniques such as ANS to increase the reliability of the result. The problem with ANN, Genetic algorithms, and other schemes is that the final solutions may become trapped in a local minimum rather than converging to the true solution of a global maximum. For this reason, some techniques purposely keep in some "unfit" solutions for reproduction rather than only the most fit since the unfit ones may wander farther from the local minimum and stumble across the global maximum.

Evolving connectionist systems is a powerful way to optimize the ANN architecture while making little or no presumptions as to what it might be (Kasabov 02). These parallel processing techniques take a lot of computer power but are more feasible as computers become more powerful and the Internet Grid and private companies provide access to many computers for computation (Foster 03).

Strictly speaking, the type of evolutionary algorithms used in computers are not based on Darwin's Theory of Evolution, which proceeds in small gradual steps over many thousands or millions of years, but on a competitive theory by Lamarck. He was a contemporary of Darwin's in the 19th Century who believed that if an animal gained extra strength in a limb through exercise, the offspring of that animal would have a stronger limb. Using computers, we force evolution to proceed as in the Lamarckian case by selecting for reproduction only those that come closest to our fitness criteria.

Evolutionary algorithms have also been applied to creating artificial life forms as part of a computer program. These are not just genetic algorithms or neural nets used to solve a particular problem; they are used in the broader context of fields such as ecology and economics to study predator-prey relationships in the wild, or more domestically, any environment such as a manufacturer-consumer in the global economy. Video games also make heavy use of environmental algorithms in defining new virtual creatures for the user to compete with. A classic example is the immensely powerful game *The Sims,* and the online version which people can play interactively with others all around the world.

When using evolutionary techniques with connectionist systems, surprising things may occur that were not planned on. This only adds to the criticism of connectionist systems by those who support logic-based AI since ANN cannot explain how their rules of connection and weights were achieved. Although many people have devised ANSs that purport to "explain" their rules, the fact remains that the rules and weights were not designed by a human but were formed as a "reflex" to the inputs and desired output of the system. In a sense, an ANN or evolutionary algorithm grows the correct solution or it does not survive.

With the decoding of the Human Genome, the field of **bioinformatics** has achieved increasing importance as it applies computer techniques to deal with a truly massive amount of information. Many jobs are open in this field to people with a background in artificial intelligence. Also the new fields of **genomics** and **proteomics** have appeared. The human genome is made up of about 30,000 genes. Each gene is basically a living factory that produces proteins to carry out the functions of the body either by acting in cells or sending messages to cells. The great challenge now is to determine what each of these genes and proteins does; not only in healthy genes but also in genetic diseases. These tasks require huge amounts of computing power because of the immense data and possible solutions. Artificial intelligence is a vital tool.

AI has also proven its worth in defense. At the end of the first Gulf War in 1990, it was stated that the savings from the use of AI in military logistics planning had repaid many times over all the money the Defense Advanced Research Projects Agency (DARPA) had put into AI research since the 1950s (http://www.au.af.mil/au/aul/school/acsc/ai02.htm). Today, AI saves many billions in computer games for military training as opposed to live war games where fuel and ammunition are used.

It would have been very difficult in the 1950s to imagine how commercially successful AI has become. At that time and for the next quarter century, the emphasis of AI researchers was on finding noble, serious applications such as theorem proving and automated discoveries. Although using AI programs to play checkers, chess, and other board games was also undertaken, AI programs were developed with serious goals in mind of understanding how humans reason, not for their entertainment value.

Today the commercial success of AI in video gaming and special effects in movies would be a great surprise to those early researchers. The video gaming industry now competes on an equal footing with movies and music for the consumer's money. People enjoy interactive games, especially when coupled with playing other people in other countries, cities, or states. Advanced video games rely heavily on AI to make creatures behave in realistic ways, not in a simple predictable pattern.

In fact a growing number of universities are now offering not only courses but degrees in video games, recognizing that a multibillion dollar industry is here to stay and there are good job opportunities in it. AI is also found in business, most notably as knowledge-based screening assistants to weed out the easy 90% of problems. Expert systems with real expertise are also used in many businesses. The only problem is that these business systems are based on the company's proprietary information and so the companies generally do not publish details about the systems lest their competitors get an advantage.

However there are still many examples of expert systems described in Appendix G on the Internet, many on the World Wide Web, some in papers located at sites like CiteSeer, and others discussed in newsgroups such as comp.ai.shells, which has much discussion of CLIPS expert systems. The advantage of a newsgroup is that you can post questions and hopefully someone will reply.

Besides job opportunities in AI and expert systems, a new job is that of **ontological engineer**. This can be considered a branch of philosophical engineering (in fact the only commercially viable branch to date.) In AI and expert systems, the term *ontology* has a different meaning than in the traditional philosophical sense of an ontology.

An ontology is the explicit formal specifications of the terms in the domain and relations among them (Gruber 93). Although the average Web user does not realize this, ontologies are common on the Web behind the scenes. These range from large ontologies organized as taxonomies, i.e., as a hierarchical top-down collection of related information in sites such as Yahoo, to sites like Amazon.com, eBay, and others that categorize items by sale price or time of auction and starting bid. Basically an ontology is a standard, agreed upon set of terms used to describe a domain, whether it be books, auction items, or something else. Without a common vocabulary, a Tower of Babel scenario would develop where no one knew what was being discussed in the domain. Many organizations are developing their own ontologies to better clarify what is being discussed in an unequivocal manner.

One of the biggest advantages of a standardized ontology is that it can become machine readable and thus computers can then aid humans in searching for the desired item. A similar effort is being made in specifying the Extensible Markup Language (XML) for different fields. Today there is a real need for people with a background in AI who are interested in becoming ontological engineers and classifying huge amounts of knowledge. Ontological engineering is also used in setting up and maintaining a database used by an expert system. This is not a trivial task, especially as the knowledge base becomes bigger and the user is allowed to enter in new knowledge for the expert system to use.

1.16 SUMMARY

In this chapter we have reviewed the problems and developments that have led to expert systems. The problems that expert systems are used to solve are generally not solvable by conventional programs because they lack a known or efficient algorithm. Since expert systems are knowledge-based, they can be effectively used for real-world problems that are ill-structured and difficult to solve by other means. A number of different paradigms for representing knowledge have been discussed, along with their pros and cons. For a more detailed discussion see (Giarratano 04).

The advantages and disadvantages of expert systems were also discussed in the context of selecting an appropriate problem domain for an expert systems application. Criteria for selecting appropriate applications were given.

The essentials of an expert system shell were discussed with reference to rule-based expert systems. The basic recognize-act inference engine cycle was described and illustrated by a simple rule example. Finally, the relationship of expert systems to other programming paradigms was described in terms of the appropriate domain of each paradigm. The important point is the concept that expert systems should be viewed as another programming tool that is suitable for some applications and unsuitable for others. Later chapters will describe the features and suitability of expert systems in much more detail. The advantages and disadvantages of expert systems were also discussed in the context of selecting an appropriate problem domain for an expert.

Future advances in AI will undoubtedly involve revolutionary new quantum computers (Brown 00), as well as computers using massive computation ability by linking up millions of computers on the Internet using the Grid, and biological wetlab using strands of RNA. These two methods can be used to solve a very complex Traveling Salesman problem in only a couple of days, much faster than any single supercomputer.

Appendix G has many links for general current information about artificial intelligence. A huge amount of information and software is available online. Other newsgroups are concerned with fuzzy logic, neural nets, expert system shells, and many other topics. In addition to the information and software, these newsgroups allow you to post questions and receive answers from other people. It is highly recommended you explore the online resources available in Appendix G for online resources by chapter.

PROBLEMS

1.1 Identify a person other than yourself who is considered either an expert or very knowledgeable. Interview this expert and discuss how well this person's expertise would be modeled by an expert system in terms of *each* criterion in the section "Advantages of Expert Systems."

1.2 (a) Write 10 nontrivial rules expressing the knowledge of the expert in Problem 1.1.
(b) Write a program that will give your expert's advice. Include test results to show that each of the 10 rules gives the correct advice. For ease of programming, you may allow the user to provide input from a menu.

1.3 (a) In Newell and Simon's classic book *Human Problem Solving* they mention the Nine-Dot Problem. Given nine dots arranged as follows, how can you draw four lines through all the dots without (a) lifting your pencil from the paper and (b) crossing any dot? (Hint: you can extend the line past the dots.)

```
• • •
• • •
• • •
```

(b) Explain your reasoning (if any) in finding the solution and discuss whether an expert system or some other type of program would be a good paradigm to solve this type of problem.

1.4 Write a program that can solve cryptarithmetic problems. Show the result for the following problem, where D=5:

$$\begin{array}{rl} & \text{DONALD} \\ + & \underline{\text{GERALD}} \\ & \text{ROBERT} \end{array}$$

1.5 Write a set of production rules to extinguish five different types of fire such as oil, chemical, and so forth, given the type of fire.

1.6 (a) Write a set of production rules to diagnose three types of poison based on symptoms.
(b) Modify the program so that it will also recommend a treatment once the poison has been identified.

1.7 Give 10 IF-THEN type heuristic rules for planning a vacation.

1.8 Give 10 IF-THEN type heuristic rules for buying a used car.

1.9 Give 10 IF-THEN type heuristic rules for planning your class schedule.

1.10 Give 10 IF-THEN type heuristic rules for buying an SUV or car.

1.11 Give 10 IF-THEN type heuristic rules for buying stocks or bonds.

1.12 Give 10 IF-THEN type heuristic rules for excuses on a late assignment.

1.13 Write a report on a modern expert system. Good sources are *PCAI*, *IEEE Expert* magazines, and Appendix G.

BIBLIOGRAPHY

(Adami 98). Christoph Adami, *Knowledge Introduction to Artificial Life*, Springer-Verlag, pp. 5, 1998.

(Bentley 02). Peter J. Bentley et al., *Creative Evolutionary Systems*, Academic Press, 2002.

(Berlinski 00). David Berlinski, *The Advent of the Algorithm*, Harcourt, Inc., pp. xvix, 2000.

(Brown 00). Julian Brown, *Minds, Machines, and the Multiverse*, Simon & Schuster, 2000.

(Bramer 99). Ed. by M.A. Bramer, *Knowledge Discovery and Data Mining*, IEEE Press, 1999.

(Cotterill 98). Rodney Cotterill, *Enchanted Looms*, Cambridge University Press, pp. 360, 1998.

(Debenham 98). John Debenham, *Knowledge Engineering: Unifying Knowledge base and Database Design*, Springer-Verlag, 1998.

(de Silva 00). Ed. by Clarence W. de Silva, *Intelligent Machines: Myths and Realities*, CRC Press, pp. 13, 2000.

(Dorigo 04). Marco Dorigo and Thomas Stützle, *Ant Colony Optimization* by MIT Press, 2004. Note the book also has a CD with software for solving the Traveling Salesman Problem using the Ant Evolutionary Algorithm.

(Fetzer 01). James H. Fetzer, *Computers and Cognition: Why Minds Are Not Machines,* Kluwer Academic Publishers, pp. 25-182, 2001.

(Foster 03). Ian Foster, "The Grid: Computing without Bounds" *Scientific American,* Volume 288, Number 4, pp. 78–85, April 2003.

(Friedman-Hill 03), Ernest Friedman-Hill, *Jess in Action: Rule-Based Systems in Java*, Manning Publications, 2003

(Fogel 03). Ed. by Gary B. Fogel and David W. Corne, *Evolutionary Computation in Bioinformatics*, Morgan Kaufmann Publisher, 2003.

(Giarratano 90a). Joseph C. Giarratano, et al., "Future Impacts of Artificial Neural Systems on Industry," *ISA Transactions*, pp. 9-14, Jan. 1990.

(Giarratano 90b). Joseph C. Giarratano, et al., "The State of the Art for Current and Future Expert System Tools," *ISA Transactions*, pp. 17-25, Jan. 1990.

(Giarratano 91a). Joseph C. Giarratano, et al., "An Intelligent SQL Tutor," 1991 Conference on Intelligent Computer-Aided Training (ICAT '91), pp. 309-316, 1991.

(Giarratano 91b). Joseph C. Giarratano, et al.,"Neural Network Techniques in Manufacturing and Automation Systems," in *Control and Dynamic Systems*, Vol. 49, ed. by C.T. Leondes, Academic Press, pp. 37-98, 1991.

(Giarratano 04). Joseph C. Giarratano, (http://www.pcai.com/Paid/Issues/PCAI-Online-Issues/17.4_OL/New_Folder/So&9i2/17.4_PA/PCAI-17.4-Paid-pp.18-Art1.htm).

(Gruber 93). T.R. Gruber, "A Translation Approach to Portable Ontology Specification," *Knowledge Acquisition* 5: 199-220, 1993.

(Hecht-Nielsen 90). Robert Hecht-Nielsen, *Neurocomputing*, Addison-Wesley Publishing Co., pp. 147, 1990.

(Helmreich 98) Stefan Helmreich, *Silicon Second Nature*, University of California Press, pp. 180-202, 1998.

(Hopgood 01). Adrian A. Hopgood, *Intelligent Systems for Engineers and Scientists*, CRC Press, 2001.

(Hrycej 97). Tomas Hrycej, *Neurocontrol*, John Wiley & Sons, Inc., 1997.

(Luger 02). George F. Luger, *Artificial Intelligence,* Fourth Edition, Addison-Wesley, 2002.

(Jackson 99). Peter Jackson, *An Introduction to Expert Systems*, Addison-Wesley Publishing Co., 1999.

(Johnson 01). Steven Johnson, *Emergence: The connected lives of ants, brains, cities, and software*, Scribner, 2001.

(Kantardzic 03). Mehmed Kantardzic, *Data Mining,* IEEE Press, 2003.

(Kasabov 02). Nikola Kasabov, *Evolving Connectionist Systems*, Springer-Verlag, pp. 7-29, 2002.

(Kennedy 01). James Kennedy and Russel C. Eberhart, Swarm Intelligence, Morgan-Kaufmann Publishers, 2001.

(Lajoie 00). Ed. by Susanne P. Lajoie, *Computers As Cognitive Tools, Volume Two: No More Walls*, Lawrence Erlbaum Associates, Publishers, 2000.

(Lakemeyer 03) ed. by Gerhard Lakemeyer and Bernhard Nebel, *Artificial Intelligence in the New Millenium,* Morgan-Kaufmann Publishers, 2003.

(Mendel 01). Jerry M. Mendel, *"Introduction to Rule-Based Fuzzy Logic Systems,"* IEEE Press, 2001.

(Satinover 01). Jeffrey Satinover, *The Quantum Brain*, John Wiley, 2001.

(Sipper 02). Moshe Sipper, *Machine Nature: The Coming Age of Bio-Inspired Computing,* McGraw-Hill, 2002.

(Swingler 96). Kevin Swingler, *"Applying Neural Networks: A Practical Guide,"* 1996.

(Wagman 99). Morton Wagman, *The Human Mind According To Artificial Intelligence*, Praeger Publishers, p. 76, 1999.