CHAPTER 1

COMPUTERS FOR SYMBOLIC PROCESSING

Benjamin W. Wah, Matthew B. Lowrie, and Guo-Jie Li

In this chapter, we provide a detailed survey of the motivations, design, applications, current status, and limitations of computers designed for symbolic processing. Symbolic processing applications are computations performed on words, relations, or meanings. A major difference between symbolic and conventional numeric applications is that the knowledge used in symbolic applications may initially be fuzzy, uncertain, indeterminate, and ill-represented. Hence, the collection, representation, and management of knowledge is more difficult in symbolic applications than in conventional numeric applications.

We first survey various techniques for knowledge representation and processing, from the points of view of both designer and user. Next, we examine the design and choice of a suitable language for symbolic processing and the mapping of applications into a software architecture. Finally, we examine the design process of refining the application requirements into hardware and software architectures and present a discussion of state-of-the art sequential and parallel computers designed for symbolic processing.

1 SYMBOLIC PROCESSING

1.1 Introduction

The development in the 1950s of the programming language IPL by Newell, Shaw, and Simon was a pioneering effort in symbolic processing by computers [172]. Data structures of unpredictable shape and size could be manipulated conveniently by programs written in IPL. Many of the early symbolic

programs, including the Logic Theorist and the General Problem Solver, were written in IPL. The invention of Lisp in 1958 by John McCarthy made symbolic programming yet easier. The language featured the recursive use of conditional expressions, the representation of symbolic information externally by lists and internally by linked lists, and representation of program and data using the same data structures [152].

Recent advances in applications of computers suggest that the processing of symbols rather than numbers will be the basis for the next generation of computers. This is highlighted by the numerous research efforts in Japan, Europe, and the United States [1, 196, 228]. Symbolic processing has been applied in a wide spectrum of areas; among them are pattern recognition, natural language processing, speech understanding, theorem proving, robotics, computer vision, and expert systems. Researchers in artificial intelligence, databases, programming languages, cognitive science, psychology, and many other fields have addressed overlapping issues in symbolic processing.

Conventional computers have been designed with numeric processing power as their focus. The disparity between symbolic and numeric operations, therefore, calls for innovative research in alternative architectures for symbolic processing. In this chapter we present a review of the state of the art in computers for symbolic processing. The discussion proceeds in a top-down fashion. First, we present the relevant features and characteristics of symbolic processing. Next, we discuss the role of techniques and methodologies in the design process and classify hardware and software architectures at different levels of design. The general view of computers designed or used for symbolic processing is depicted in Figure 1.1; the section discussing each portion is indicated in the figure.

Section 1 develops a classification of general computations. From this classification, a definition of symbolic processing is derived in Section 1.1. Typical symbolic processing applications and their characteristics are discussed in Section 1.2.

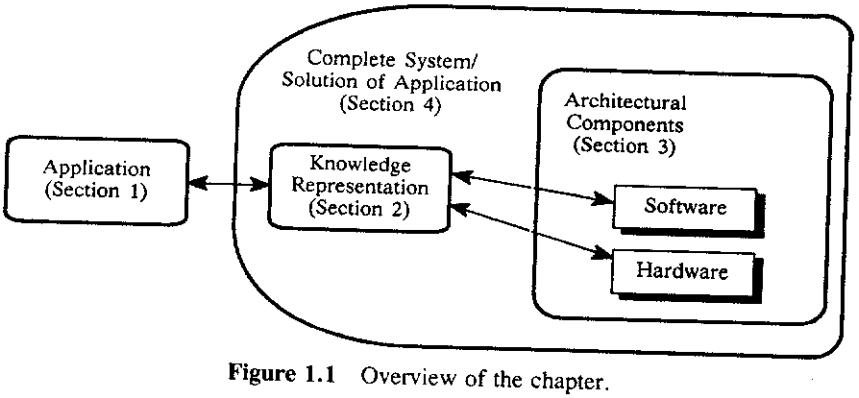

Figure 1.1 Overview of the chapter.

Knowledge representation and knowledge processing are two important characteristics of solutions to a symbolic processing problem. Knowledge representation is the technique for representing data and information in a computer; it is discussed in Section 2.1. Knowledge processing is the technique for controlling the manipulation of knowledge in the system and is the topic of Section 2.2.

The design of a computer relies on various concepts and strategies for implementing knowledge processing techniques. Section 3 examines the architectural concepts behind the design of symbolic processing systems. Software architectures are studied in Section 3.1, and hardware architectures in Section 3.2.

A complete system for symbolic computation is the result of the application of a design philosophy, architectural components, and available technology. Complete systems are the topic of Section 4, in which many existing and experimental systems are discussed and compared.

Symbolic processing systems will evolve as new concepts and technologies develop. Section 5 outlines some recent research that is likely to impact the design of symbolic processing systems in the future.

1.2 Classification of Computations

One of the fundamental debates on intelligent behavior is about the nature of symbols. A number of scientists view human beings and computers as physical symbolic systems that produce an evolving collection of symbolic structures. In their 1975 Turing award lecture, A. Newell and H. Simon stated a general scientific hypothesis—The Physical Symbol System Hypothesis [174]:

> A physical symbol system has the necessary and sufficient means for general intelligent action.

By "necessary," they mean that any system that exhibits general intelligence will prove upon analysis to be a physical symbol system. By "sufficient," they mean that any physical symbol system of sufficient size can be organized to exhibit general intelligence. Research on artificial intelligence (AI) addresses the sufficiency of a physical symbol system for producing intelligent action, while investigators in cognitive psychology attempt to demonstrate that a physical symbol system necessarily exists wherever intelligence is exhibited. Although empirical in nature, the continuous accumulation of empirical evidence on the above hypothesis in the last 30 years has formed the basis of much research on AI and cognitive science.

Since our focus is on computers for symbolic processing, we will first classify computations performed on computers. From this classification is derived the definition of symbolic processing used in this chapter. There are five classes of computations: analog, numeric, word, relational, and meaning. These classes are based on the primary unit of data in the computation.

Analog: The analog class of computation encompasses those computations whose functions have parameters that are continuous variables. This is not the primary area of computation in a digital computer, as digital computers use digital memory. In digital computers, this kind of computation primarily measures environmental parameters.

Numeric: In this class of computation, the primary unit upon which functions are performed represents magnitude. Many applications of computers fall into this category; functions on memory elements containing integers, floating point numbers, and the like are numeric.

Word: In this class of computation, the parameters of functions are words that do not necessarily have quantitative value. Text processing is an example of this kind of computation.

Relational: In relational computations, functions operate on relations among words; that is, the primary units of data are groups of words that have some relational interpretation.

Meaning: Very little research has been done on techniques for automated computation at the meaning level. The primary unit of data is an interrelated series of relations that represent semantics and meaning.

A few examples to illustrate the various classes of computations are shown in Figure 1.2. Figure 1.2(*a*) presents weather forecasting, a standard supercomputer application. The computation begins with analog measurements of the atmosphere (arc 1). These measurements are then converted to numeric entities (arc 2). Most of the computation occurs in the numeric stage (arc 3, representing the conversion of numbers into different sets of numbers), and conversion to meaning is done at the very end (arc 4, which may be done by humans instead of the computer). Story comprehension from speech input is an example that uses the full spectrum of the classes of computations. Figure 1.2(*b*) shows the flow of data. Computation may also flow in the opposite direction; robot control, depicted in Figure 1.2(*c*) is an example of this.

The design of a computer system can be viewed as the maximization of performance with respect to the problem to be solved, subject to cost constraints. Computations in a higher level of abstraction are usually carried out by transforming them into more definite computations. For example, database queries function primarily at the word and relation level of computation. However, if we wish to know the average salary of employees in a database, numeric computations would be required. The design of this system may not, however, benefit from the inclusion of fast arithmetic units, because the performance gain may not be large enough to counteract the loss of other features necessitated by the cost constraint. The difference between this database system and a system that would include fast arithmetic units is in the emphasis of the computations.

A computer that is intended to perform more abstract computations should also be able to perform more definite computations. For instance, the afore-

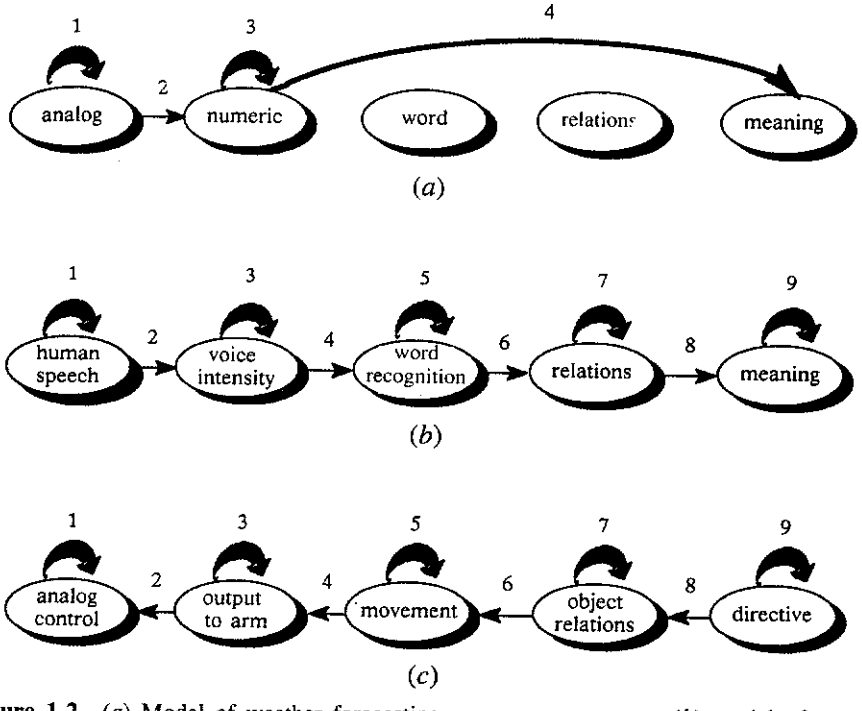

Figure 1.2 (*a*) Model of weather forecasting on a supercomputer; (*b*) model of speech understanding; (*c*) model of robot control.

mentioned database computer should be able to perform fluid dynamics calculations. It may, however, be less efficient than a computer of comparable cost that is intended for numeric calculations.

Using this classification, it is possible to describe concisely what is meant by "symbolic processing". *Symbolic processing is defined as those computations that are performed at the same or a more abstract level than the word level. Symbolic processing computers* are computers that are designed to carry out operations at the same or a more abstract level than the word level.

1.3 Characteristics of Symbolic Processing Applications

This section presents symbolic processing applications and their overall features. A few applications and their characteristics are presented in Table 1.1. The rest of the section lists features that characterize general symbolic processing applications [239]. These features do not apply to every instantiation of a symbolic processing problem and are intended as design guidelines that target general symbolic processing techniques.

TABLE 1.1 Some Symbolic Processing Applications.

Application	Characteristics
Problem Solving general specific accounting programming/compilation text processing human interface	User inputs problem, system attempts to solve User encodes solution of problem Metaknowledge for specific problems is well understood Small-grain parallelism is predominant
Database Management variety of applications often an integral part of larger systems	Organization of information for retrieval Efficient algorithms to consider all run-time possibilities are too complex Metaknowledge is application dependent Large potential for parallelism, both small-grain and large-grain
Expert Systems diagnosis medical plant disease computer system errors design assistance architecture computer architecture computer chips personal systems business finance wine tasting others	Ill-structured collection of facts, inferences, and so on are represented as knowledge-intensive program in specific domain [96] Knowledge and metaknowledge are usually provided by designers Large potential for parallelism
Natural Language Processing understanding generation translation	Translate natural language to machine representation Translate machine representation to natural language Translate between two forms of natural language [2, 13, 229]
Computer Vision signal processing pattern recognition image understanding	Primarily numeric Patterns viewed as sentences—symbolic Higher-level reasoning at the image-understanding level [3, 76]
Learning expert systems deduction improve system efficiently knowledge acquisition	Ability to adapt to environment to increase efficiency Fundamental to symbolic processing [157, 190]

Incomplete Knowledge. Many applications are nondeterministic; it is not possible to predict the flow of computation in advance. This is due to incomplete knowledge and understanding of the application. The lack of complete knowledge may also lead to a need for dynamic execution, which allows the creation of new data structures and functions during the solution of the problem. Furthermore, data structures used in the solution may be arbitrarily large, necessitating dynamic allocation of memory, tasks, and other resources. System design should meet the need to cope with dynamic and nondeterministic execution. An architecture that can adapt to the more efficient performance of computations not anticipated at design time is referred to as an open system.

Knowledge Processing. A computation can be viewed as manipulations of a set of data. In Section 1.1, computations are classified by the data on which they operated. The nature of the operations performed on the data depends on the application and the way the data is stored. As illustrated in Figure 1.3, it is processing knowledge about the computation, whether that be algorithms, techniques for evaluation, or some other reasoning technique, that controls and dictates the manipulation of data. At a level above this is metaknowledge—knowledge about knowledge [7, 49, 80]. Metaknowledge includes the extent and origin of the domain knowledge of a particular object, the reliability of certain information, and the possibility that an event will occur. Meta-level knowledge can be considered to exist at a single level or in a hierarchy [25] containing an arbitrary number of levels, each serving to direct the use of knowledge at the lower levels.

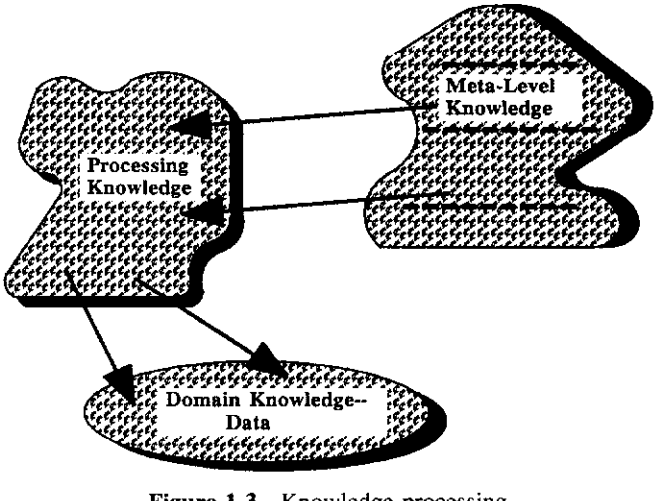

Figure 1.3 Knowledge processing.

Metaknowledge can be classified as deterministic or statistical according to correctness and performance considerations [139]. Deterministic metaknowledge refers to the knowledge about precedence relationships that results from a better understanding of the problem and helps to reduce its resource and time complexity. Statistical metaknowledge can be used to order object-level actions before their performance for greater efficiency.

Rather than adding more heuristics to improve the performance of a system, metaknowledge about the effective use of existing heuristics can be collected and developed. Metaknowledge can also be used in the formalization of belief, default reasoning, inference in changing situations, and other problems [7].

Symbolic Primitives. A general symbolic application may contain primitive symbolic operations. Typical operations are comparison, sorting, selection, matching, and logical operations such as union, negation, intersection, transitive closure, and pattern retrieval and recognition. These operations may be performed at more than one level of computation (for instance, at both word and relation levels). Higher levels of computation may also contain complicated primitive operations such as unification.

Parallel and Distributed Processing. Many symbolic applications exhibit a large potential for parallelism. Parallelism may be categorized into AND-parallelism and OR-parallelism. In AND-parallelism, a number of independent tasks are executed concurrently. OR-parallelism is a technique used to shorten processing time in nondeterministic computations by simultaneously evaluating alternatives at a decision point.

2 KNOWLEDGE REPRESENTATION AND PROCESSING

A symbolic application is a problem whose inputs and outputs are symbolic. *Symbolic processing* refers to the techniques employed by the system for finding the solutions of the problem. The characteristics of symbolic applications were discussed in the previous section. The emphasis of this section is on the *features of symbolic processing* as they relate to the design of computers. Section 2.1 discusses techniques for representing knowledge. The issues involved in the control of knowledge processing are presented in Section 2.2.

2.1 Knowledge Representation

To design an efficient computer for a given application, it is first necessary to characterize the programs that will run on the computer. Of primary importance in the solution of symbolic processing problems is the choice of a knowledge representation [149]. Appropriate symbolic structures to represent knowledge, and appropriate reasoning mechanisms to answer questions and assimilate new

information must both be selected [120]. Four criteria can be used to evaluate a knowledge representation scheme: flexibility, user friendliness, expressiveness, and efficiency of processing. Flexibility, user friendliness, and expressiveness are required to simplify the tasks of programming and comprehension. The efficiency or tractability of a knowledge representation scheme dictates the efficiency of the application. Much of the research in this area represents a tradeoff between expressiveness and tractability.

Despite a great deal of effort devoted to research in knowledge representation, very little scientific theory is available either to guide the selection of an appropriate representation scheme for a given application or to show how to transform a representation into a more efficient one. Although a number of knowledge representation schemes have been proposed, none is clearly superior to the others for all applications.

The following sections present two attributes for consideration when comparing knowledge representations: the *local versus distributed* and *declarative versus procedural* attributes. The classical knowledge representation schemes are then evaluated on the basis of these features.

2.1.1 Features of Knowledge Representations

Local Versus Distributed Representations. In a local representation, each conceptual datum is stored in a separate hardware unit. A word or data item stored in a register is an example of local storage for that data item. When stored locally, data are simple to read, update, and understand. Unfortunately, if any hardware unit fails, all data contained in that unit is lost to the system. Most current systems, both symbolic and numeric, use local representations for individual pieces of data.

In a distributed representation, a piece of knowledge is represented by many storage units, and each unit may be part of several pieces of knowledge that correspond to features from different concepts. The advantage of such a representation is that it is fault tolerant. If a small proportion of units fails, the integrity of the distributed data undergoes little change. This property is very attractive for practical implementations. Distributed representations also allow a great deal of parallelism in computation [191]. However, they are usually harder for a user to understand and modify.

Table 1.2 summarizes the salient characteristics of local and distributed representations. It should be noted that there is no concrete boundary between local and distributed features. Some features of a knowledge representation scheme are local, whereas others are distributed. At one extreme is a standard implementation of a simple Lisp program, for instance, which can be thought of as a hierarchy of local representations. The program is stored as one unit of information; the data structures used by the program are also stored as a single entity; finally, each piece of data within a data structure is stored in one memory location. At the other extreme is a standard implementation of a neural network. A predicate logic program, however, is neither fully local nor

TABLE 1.2 Attributes of Local and Distributed Representations.

Attribute	Local	Distributed
Storage technique	Each datum stored in dedicated hardware	Data represented over multiple units
Ease of understanding	Easy for humans to comprehend	Difficult for humans to interpret
Modification of stored data	Simple	More difficult
Fault tolerance	Loss of hardware results in loss of all stored data in that unit	Loss of small proportion of units does not seriously damage integrity of data

fully distributed. The complete program is not a single entity but a set of logic statements. Each statement, however, is an example of a local representation. Predicate logic is considered further in Section 2.1.2.

Declarative Versus Procedural Representations. The issue of distributed versus local representations concerns methods of representing information in the computer hardware. In contrast, procedural and declarative representations involve techniques for representing processing knowledge and processing methodologies in computer programs.

A program written in a declarative representation consists of a set of domain-specific facts or statements and a technique for inferring knowledge from these statements. It is, therefore, characterized as a set of statements of knowledge *about* the problem domain. Pure predicate logic and production systems (to be discussed in Section 2.1.2) are examples of declarative representations.

In a procedural representation, program statements consist of steps to be taken in the solution of the problem—statements of knowledge about *how* to solve the problem. Examples of procedural program representations include the C language and Lisp.

Declarative representations are user-oriented and emphasize correctness and user friendliness of programs. They are referentially transparent: the meaning of the whole can be derived solely from the meaning of the parts, independent of the sequence of operations that leads to these parts. This may increase programmer productivity [245] and result in tremendous potential for parallelism [88, 128].

Unfortunately, programs written in declarative representations are often inefficient due to nondeterminism, implicit control aspects, and inconsistent knowledge. It is hard to add domain-specific knowledge and meta-level knowledge to declarative programs. The difficulty with using declarative repre-

TABLE 1.3 Attributes of Declarative and Procedural Representations.

Attribute	Declarative	Procedural
Emphasis	Knowledge of domain	Knowledge of solution
Technique	Domain-specific statements	Solution techniques
Orientation	User friendliness; ease of understanding	Efficiency of solution; ease of representing control knowledge
Parallelism	Natural, but countered by unnecessary search	Constrained and often user specified
Control	Transparent to the user	Specified by the user

sentations to solve symbolic problems lies in determining how to *use* the facts stored in the program's data structures, not in deciding how to store them.

Procedural programs are not as user friendly as declarative ones because the programmer must specify all control knowledge. In addition, the validity of a procedural statement often relies heavily on other procedural statements in the program, and this complicates both the creation and the modification of software. The loss of flexibility in a procedural programming environment is counteracted by the gain in ease of representing control knowledge. Procedural schemes allow the specification and direct interaction of facts and heuristic information, thereby eliminating wasteful search. Metaknowledge can also be included easily in procedures. Overall, procedural representations are as much concerned with the technique and efficiency of the computation as with the ease of representing the domain knowledge.

The salient features of declarative and procedural representations are summarized in Table 1.3. Practical knowledge representation schemes may have both procedural and declarative features.

2.1.2 *Classical Knowledge Representation Schemes.* In this section, the classical knowledge representation schemes are described and evaluated. Those that have received the greatest attention include predicate logic, production systems, semantic networks, frames, procedural languages, and fully distributed representations.

Predicate Logic. Predicate logic studies the relationship of implication between assumptions and conclusions. Logic often seems a natural way to express certain notions, and there are standard methods of determining the meaning of an expression in a logic formalism [125]. Logic is useful for exploring the epistemological problems that determine how the observed facts can be represented in the memory of a computer independent of how the knowl-

edge is used. The major disadvantage of logic stems from the separation of representation and processing.

Production Systems. Production systems use collections of rules to solve problems. The rules consist of condition and action parts, or antecedent and consequent parts [173]. Production systems provide a useful mechanism for controlling interactions between statements of declarative and procedural knowledge. For this reason, production systems have been used extensively in expert systems and knowledge engineering. Unfortunately, the expressive power of production systems is limited. Some researchers have argued that rule-based expert systems cannot achieve expert-level behavior [59]. Production systems are also inefficient because of their high control overhead.

Semantic Networks. A semantic network is a directed graph whose nodes represent objects, concepts, or situations, and whose arcs represent relationships between nodes [183]. The basic inference mechanism in semantic networks is *spreading activation*. The idea has a clear neural inspiration: certain concepts in memory become a source of activation, and activation spreads in parallel to related concepts. This graphical representation allows certain kinds of inference to be performed by simple graph-search techniques. Although simple semantic networks can only express variable-free assertions, several authors have shown that the expressive power of semantic networks can be extended to equal that of predicate logic [126]. Frequently, semantic networks are used as data structures for manipulation by other knowledge representation schemes (such as Lisp, a procedural representation).

Frame Representations. Frame representations employ a data structure called a *frame* for representing stereotypical situations [158]. The frame description form is an elaboration of the semantic network form. Its emphasis is on the structuring of types (called frames) in terms of their attributes (called slots). A frame contains declarative and procedural information in predefined internal relations. Attached to each frame is heuristic information, such as a procedure on how to use the information in the frame. Although many frame implementation issues are unresolved, the framelike structuring of knowledge appears promising and has appeared in various forms in many conventional languages.

Procedural Representations. In a procedural representation, a knowledge base is viewed as a collection of modules expressed in a procedural language, such as Lisp or C. The procedural scheme is capable of representing heuristic knowledge and performing extended logical inferences, such as plausible reasoning. Because it eliminates wasteful search, this representation scheme can be carried out efficiently. However, it is often limited by the available constructs. Conventional FORTRAN or Pascal programs, for example, have been found to be inadequate for the support of efficient symbolic processing.

Connectionist Representations. A connectionist representation is a form of distributed representation: concept representations are spread over a number of modules or units. When presented with input, units having a positive correlation with an input feature are activated, whereas those with a negative correlation produce inhibitory signals. In this fashion, input can be recognized as a function of connection strengths among units (see also Section 3.2.2) [191]. Distributed representations allow automated procedures for learning concepts and representations, and have great potential for parallel computation. Their major drawback is the difficulty of interpreting system states and internal representations. Furthermore, the learning of connection strengths often requires a lengthy training period.

A given representation may exhibit local or distributed and declarative or procedural aspects at different levels of the representation. Table 1.4 summarizes the characteristics of the representations introduced here and categorizes them by the hierarchy of knowledge representations inherent in the technique.

2.2 Knowledge Processing

Different reasoning methods are associated with different knowledge representation schemes and require different architectural supports. Table 1.5 shows the classical knowledge representation paradigms and their respective reasoning techniques.

It is argued that humans use logic-like reasoning in the domain of rational knowledge and apply memory-based reasoning to perceptual actions. For over 30 years, logic-like deduction has been the dominant paradigm in AI research. This paradigm has been applied to a wide range of problems, especially expert systems. Although intelligent behavior often resembles logic-like reasoning with limited search, the intensive use of memory to recall specific episodes from the past (rather than rules) could be another foundation of machine reasoning [209]. Memory-based reasoning (or case-based reasoning) does not use rules; instead it attempts to solve the problem by direct reference to memory. The Connection Machine is an example of a machine designed for memory-based reasoning, although it can also be programmed to perform logic-like reasoning [103].

A knowledge processing technique must be tailored to cope with the application's requirements. The greatest need in symbolic processing applications is the ability to deal with uncertain, incomplete, or conflicting information. Techniques for dealing with these problems are discussed in Section 2.2.1. Section 2.2.2 discusses methods for exploiting parallelism.

2.2.1 Uncertain, Incomplete, and Inconsistent Knowledge Processing. The techniques for dealing with these problems in knowledge processing are detailed in Figure 1.4. The rest of the section is devoted to a brief discussion of the entries in the figure.

TABLE 1.4 Characterization of Knowledge Representation Schemes.

Representation	Level of Representation	Characterization
Logic	Variable	Local/declarative
	Statement/relation	Local/declarative
	Program	Distributed/declarative
Production System	Variable	Local/declarative
	Statement/relation	Local/either
	Program	Distributed/declarative
Semantic Network	Node	Local/declarative
	Arc/relation	Local/declarative
	Network	Local/procedural
	Program	Distributed/declarative
Frames	Variable	Local/declarative
	Statement	Local/either
	Slot	Local/either
	Frame	Local/declarative
	Program	Distributed/declarative
Procedural	Variable	Local/either
	Statement	Local/procedural
	Program	Local/procedural
Connectionist	Connection strength	Local/declarative
	Propagation technique	Local/procedural
	Data and knowledge	Distributed/declarative

TABLE 1.5 Reasoning Techniques.

Representation	Typical Reasoning Technique
Logic	Resolution (unification)
Production system	Forward/backward chaining
Semantic network	Spreading activation
Frames	Procedure attachments
Procedural	Control flow
Connectionist	Propagation of excitation

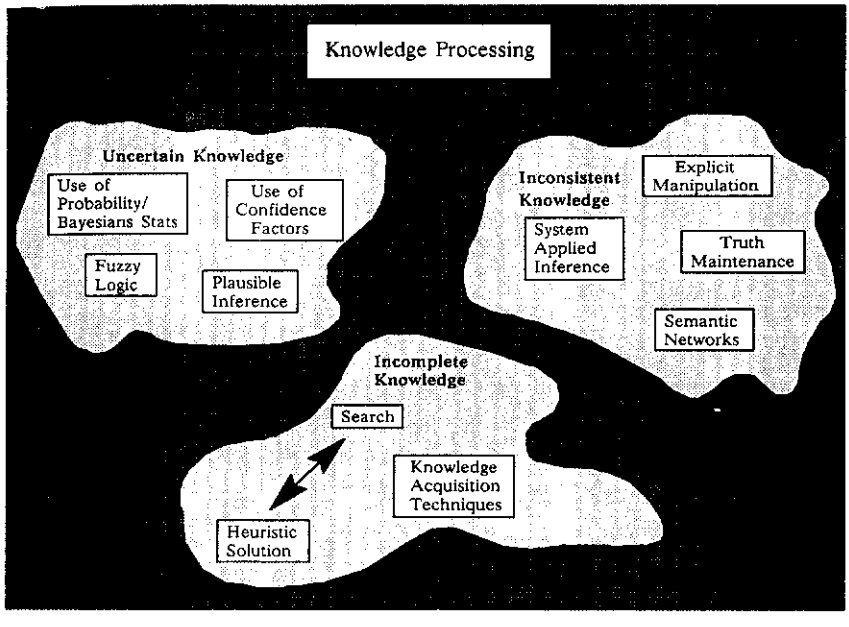

Figure 1.4 Indeterminate knowledge processing: some techniques.

Uncertain Knowledge. Methods such as conventional knowledge representation techniques based on predicate calculus are not well suited for representing common-sense knowledge. Explicit and implicit quantifiers are fuzzy, and the standard inference methods mentioned earlier make no provision for dealing with uncertainty. Two types of uncertainty have been studied. One type is caused by noisy data and the fuzzy meaning of symbols; the other is associated with uncertain inference rules.

A number of methods and theories of capturing uncertainty have been examined in recent years. Probability and Bayesian statistics form the basis of most approaches to this problem, which include fuzzy logic [250, 251], confidence factors [30], Dempster and Shafer's theory of plausible inference [199], odds [60], and endorsements [38].

Uncertain knowledge is most frequently handled by two principal components. The first is a translation system for representing the meaning of propositions and other semantic entities. The second is an inferential system for deriving answers to questions about information resident in a knowledge base. The application of Bayesian statistics to expert systems follows this approach [60]. A confidence factor (CF), or certainty factor, is used to decide among alternative answers during a consultation session. A CF of a rule is a measurement of the strength of the association between premises and conclusions. A positive CF indicates that the evidence confirms the hypothesis, whereas a negative CF disconfirms the hypothesis.

Dempster and Shafer's theory of plausible inference provides a natural and powerful methodology for representing and combining evidence. Ignorance and uncertainty are directly represented in belief functions and remain throughout the combination process.

Endorsements are records of information that affect a hypothesis' certainty. They can be propagated over inferences, but in a manner that is sensitive to the context of the inference.

Incomplete and Inaccurate Knowledge. A key feature of symbolic computations is nondeterminism arising from the fact that almost any intelligent activity is likely to be poorly understood. This implies that no systematic, direct algorithms are available for performing the activity. When a problem becomes well understood and can be solved by a deterministic algorithm, the solution of the problem is no longer considered to be intelligent [206].

The starting point of any conventional computation is a deterministic algorithm. Since most symbolic processing applications are knowledge intensive, such a deterministic algorithm may not exist. Efficient solution of the problem, therefore, requires continual refinement of the computation technique and may employ various knowledge acquisition techniques. When knowledge of the application domain is incomplete or uncertain, heuristic methods are used [133, 134, 180, 181, 205]. A heuristic suggests plausible actions to take, or implausible ones to avoid. It is desirable to use concise and accurate domain-specific knowledge and metaknowledge. Unfortunately, this body of information is difficult to acquire in practice and may be fallible if incomplete or very large if complete.

The nondeterministic nature of computations and the fallibility of heuristics may lead to anomalies of parallelism. For instance, when multiple processors are used, one or more of the processors may be guided by the heuristic function into a part of the search tree that is not explored in the same order as that in sequential processing. This out-of-order exploration of the search tree, coupled with the pruning of undesirable nodes, may result in a speed-up (as compared to sequential processing) by a factor less than one or greater than the number of processors. Some results on the occurrence of anomalies and how to cope with them can be found in the literature [129, 136, 141, 238].

In addition to heuristics, several new forms of logic for belief and knowledge have been introduced. Traditional reasoning methods suffer from the problem of *logical omniscience* [105]. Logical omniscience refers to the assumption that agents are sufficiently intelligent that they know all valid formulas. Thus, if an agent knows proposition p, and knows that p implies q, then the agent must know q. In real life, people are certainly not omniscient. The newly introduced forms of logic are more suitable than traditional logic for modeling beliefs of humans (or computers) with limited reasoning capabilities [63, 87].

Processing Inconsistent Knowledge. Traditional logic is *monotonic*. Monotonicity implies that new axioms may be added to the list of provable theorems only when they are consistent. Nonmonotonic reasoning provides a

more flexible and complete logical system, as well as a more accurate model of human thought processes. The motivations for nonmonotonic reasoning can be classified into two general areas: default reasoning, and reasoning in a changing environment [24].

Default reasoning can be broken into two distinct areas: exceptions to the rule, and autoepistemic logic. As the name implies, exceptions to the rule allows relations that contradict more general relations. A statement with *most* as a relation will not add any information to a monotonic logic system. Nonmonotonic systems allow a representation that includes exceptions to the general rule, without eliminating the validity of the computing environment.

Autoepistemic logic (also known as circumscription or the closed-world assumption) allows the system to reach conclusions about relations about which no facts exist in the database. This requires the assumption that all relevant knowledge is in the database (the closed-world assumption).

In a monotonic system, existing knowledge and data cannot be modified without restarting all inference processes and withdrawing all conclusions. In a world where new discoveries and revisions of previous beliefs are the norm, this is a poor model for a large knowledge-based system. Accommodating a changing environment is particularly important when default reasoning is used. A statement inferred by default may be corrected in light of additional evidence.

The distinguishing feature of the various techniques for dealing with inconsistent knowledge is their method of handling correction of the knowledge base. These methods include explicit encoding, system applied inference, semantic networks, and truth maintenance. In explicit encoding, the programmer is responsible for writing code that will update the database when a new statement is added which may conflict with other statements [72]. A system employing system applied inference contains user-encoded functions that automatically search for inconsistencies in the knowledge base [101]. For example, McCarthy and Hayes have indicated how actions might be described using modal operators like "normally" and "consistent" [151], and Sandewall used a deductive representation of nonmonotonic rules based on a primitive called UNLESS [192]. Reasoning with semantic networks is another technique for the ordering of inferences and default reasoning, although semantic networks have been criticized as insufficiently formal logical systems, lacking a clear inference technique. Doyle's truth maintenance system (TMS) records and maintains the reasons for the system's beliefs. These beliefs can be revised when discoveries contradict assumptions [58]. Improvements have been explored by de Kleer in his assumption-based truth maintenance systems (ATMS) [121-123]. In the IBM YES/MVS expert system, inconsistent deductions are automatically removed and new consequences computed in accordance with the changed facts [197].

2.2.2 *Parallel Knowledge Processing.* Humans are often thought of as the most efficient symbolic processing engines. Some researchers claim that symbolic problems can, therefore, be most effectively solved using techniques

similar to those employed in the human brain. Observations of human intelligence suggest that human cognition can be divided into perceptual and rational cognition, each of which may involve a different degree of parallelism. In the perceptual form of cognition, which includes such activities as vision and speech understanding, massive parallel processing is possible due to the large number of independent data and the simple flow of control. In rational cognition, on the other hand, only limited parallelism is possible. Therefore, the problems of high-level reasoning should be solved by trying to accumulate heuristics rather than by trying to exploit parallelism.

Unfortunately, early experiences with symbolic multiprocessor architectures such as Hearsay-II [70], Eurisko [135], and multiprocessor implementation of forward chaining rule based expert systems [75] have shown that parallel symbolic programs exhibit small speed-ups only [119]. This has led to the possibly incorrect conclusion that symbolic programs written for sequential execution have low potential for parallelism.

Methods of parallel knowledge processing are distinguished by four features: deterministic and nondeterministic parallelism, granularity of parallelism, data- and control-level parallelism, and user- and system-defined parallelism. These features are summarized in Table 1.6. Designs of parallel symbolic processors are presented in detail in Sections 3 and 4.

3 ARCHITECTURAL CONCEPTS FOR SYMBOLIC PROCESSING

Once a symbolic processing application has been characterized and a suitable representation technique selected, it is possible to choose appropriate features and attributes for a computer system for that application. An architectural component of a processing system is defined as a hardware or software structure which supports an application. In this section, we discuss current software and hardware architectures that are useful for symbolic processing systems, focusing on architectures specific to symbolic processing and the way that they relate to fundamental design concepts.

Software architectures are comprised primarily of software languages and environments suitable for encoding particular applications. The selection of a software environment imposes certain features that the software and hardware must support. The design process decides among software and hardware implementations of the required features and is driven by a tradeoff between cost and expected performance improvement.

3.1 Software Architectures

The area of software architectures for symbolic processing has two important facets. The first is the design of appropriate software facilities, tools, and languages for the symbolic application; the second concerns the tools used for mapping a symbolic application into software.

TABLE 1.6 Issues in Parallel Processing.

Issue	Definition	Comments
Deterministic and nondeterministic	Concurrent execution of multiple units of computation, all of which are necessary for job completion	Low overhead guarantees speed-up Tasks must be independent Pure functional programming is deterministic
	Multiple potential solutions evaluated in parallel Parallelism used to replace or augment backtracking	Easy to implement—always independent Nondeterministic nature may lead to anomalies in parallelism
Granularity	Size of units of computation to be executed by a single functional unit	Difficult to determine Is a function of knowledge representation, problem complexity, the shape of the search tree, distribution of processing times, and the dynamic nature of the problem [140]
Data level and control level	Data stored one element per processor Executed in single-instruction stream, multiple-data stream (SIMD) fashion	Can be used for large database operations, sort, set operations, statistical analysis, etc. [240] Can be implemented in memory—referred to as an *active memory* [67]
	Independent control for parallel tasks	Major type of parallelism used Multiple-instruction stream, multiple-data stream (MIMD) systems Detection of parallelism can be more difficult than in numeric programs. Nondeterministic nature requires dynamic mapping
User defined and system defined	User specifies portions of program which can execute in parallel	In numeric processing, DOACROSS is a typical example [247]. The FUTURE construct in Multilisp is a symbolic construct [90].
	Parallelism is detected and exploited automatically by the compiler or run-time software and hardware	Fully distributed representations allow massive system-level parallelism. Some systems may employ both—user only aids in indicating available parallelism.

The discussion of software architectures is organized in the following manner. First, the process of designing software facilities and languages is analyzed. Next follows an overview of the most frequently researched and utilized programming paradigms for symbolic processing. Finally, the problem of mapping applications into software is examined.

The Design of Software Languages. The objective of a software language is to provide software support and implementation of the chosen knowledge representation(s). As when choosing the knowledge representation scheme, the major goals of the language designer are ease and ability to represent the application and the efficient execution of the algorithm. Once the designer has selected a technique or techniques for representing knowledge, the major features of the language become apparent. For example, the choice of a logic representation dominates the characteristics of the Prolog language.

The software technique for implementing these features may not be as clear, however. For instance, Lisp is a procedural language developed for symbolic processing. In its design, functional programming with recursion and list-structured data were selected. These are not obvious choices, but they arise out of the use of a procedural representation. These features were also selected for ease of representation and efficiency of processing. Another example of implementation is the additional "impure" features that may be added to the implementation of a knowledge representation scheme to support efficient computation and provide flexibility to the user. Cuts and side effects in Prolog are well known examples. They can be seen as a procedural addition to a logic representation to support the efficient *implementation* of logic programming.

The extension of conventional von Neumann computer languages for symbolic processing is an issue that has been explored extensively. By their nature, conventional computer languages are based on procedural representations. Examination of the characteristics of symbolic processing, as discussed in Section 3.1.1, reveals the desirable features to incorporate into a conventional programming language. Possible features include data structures, symbolic primitives, and recursion. Conventional languages designed for numeric processing, such as FORTRAN, do not provide adequate support for symbolic processing. In particular, the languages are not sufficiently flexible to enable simple encoding of very complex symbolic operations. For this reason, this section emphasizes the new and less conventional languages for symbolic processing.

As noted in the preceding section, the emphasis in the design of new representations for symbolic processing problems has been on the addition of declarative and distributed features to existing representation schemes. Part of the motivation for this emphasis is referential transparency, or freedom from side effects. This relieves the users of some of the programming burden and allows easier programming of complex applications. The following sections discuss three paradigms for the design of software languages that promote

referential transparency: *functional languages* [16, 44, 97], *rule-based languages* [36, 52, 126], and *object-oriented languages* [81, 208, 210, 242].

3.1.1 Functional Programming Languages. The functional programming approach does not employ states, program counters, or other sequence-related computational constructs. A program is a function in the mathematical sense. The program, or function, is applied to the input, and the function is evaluated to produce the desired output. A functional language can be thought of as a language based on the lambda calculus; operators are applied to data or to the results of other function evaluations. John McCarthy's conception of list processing is a pioneering effort in this area [150]. Examples of functional languages include pure Lisp [152], Backus' FP [16], Hope [32], Val [155], and Id [14].

In a functional language, the meaning of an expression is independent of the history of any computation performed prior to the expression's evaluation (referential transparency). Precedence restrictions occur only as a result of function application. Notions such as side effects and shared memory do not exist in functional programs. The lack of side effects results in the determinacy property that is so valuable in parallel processing. A function has the same value (assuming its evaluation terminates) regardless of the order in which its arguments are computed. Hence, all arguments and distinct elements in dynamically created structures in a functional program can be evaluated concurrently. For example, consider a simple program for computing the average of the numbers in a list *s*.

```
average(s) = div (sum(s), count (s))
```

If we attempt to evaluate `average (1.(2.(3.nil)))`, the computation of `sum (1.(2.(3.nil)))` can clearly proceed independently of the computation of `count (1.(2.(3.nil)))`. The key point is that parallelism in functional languages is implicit and is supported by the underlying semantics. There is no need for special message passing constructs, synchronization primitives, or constructs for specifying parallelism. It has been reported that implementation of functional languages seems easier on a parallel computer than on a sequential computer [46].

Programming in functional languages facilitates specification or prototyping prior to the development of efficient programs. Given a satisfactory specification, it is possible to develop an efficient program through *program transformation*—the systematic refinement of the program specification. Because functional languages are referentially transparent, they can be refined as familiar mathematical forms. Another advantage of functional programming is that it can represent higher-order functions; a function can be passed as an argument. A comparison of functional programming with von Neumann programming is presented in Table 1.7.

TABLE 1.7 Functional Versus von Neumann Programming.

Functional Programming	von Neumann Programming
Programs are built only from other programs.	Programs contain programs, expressions, and variables.
Programs can be freely built from other programs.	Programs are composed only of common data storage.
Same program can treat objects of different structure and size.	Changing size or structure of data means changing the program.
Programs have a strong theoretical background. Programs may be proven correct, as in mathematics.	Few general practical theorems exist about programs. Proving correctness of a program is extremely difficult.

Pure Lisp is a functional language. Many dialects of Lisp, however, are not purely functional. Operations on global variables, property lists, input/output, and other features of these dialects create side effects. Side effects support efficient computation by obviating the recomputation of functions whose results are required in more than one place; they also support convenient input/output. Unfortunately, the property of referential transparency is lost in most practical Lisp languages. Moreover, precedence restrictions are represented not only by functional calls, but also in procedures.

It is not as straightforward to identify the parallel tasks in a language with side effects as it is in a purely functional language. Users must use special primitives to identify independent tasks. Several parallel Lisp languages have been proposed and implemented. Multilisp, developed by R. Halstead at MIT and implemented on a 128-processor Butterfly parallel processor, includes the usual Lisp side effect primitives for altering data structures and changing the values of variables [89]. Parallel symbolic concurrency is introduced by means of the `pcall` and `future` constructs [41]. Both utilize an implicit fork-join. For example, `(pcall A B C)` will result in the concurrent evaluation of expressions A, B, and C, and `(future X)` immediately returns a pseudo location for the value of X and creates a task to concurrently evaluate X. The use of `future` allows concurrency between the computation of a value and the use of that value. The primitive `future` was introduced because `pcall` alone did not provide a great deal of parallelism [90].

Proponents of functional languages believe that their simplicity and elegance will promote more orderly, more rigorous, more verifiable, and ultimately more efficient programming. Opponents worry about loss of expressiveness caused by the expression-evaluation–only model. The crucial disadvantage of functional programming is that it is difficult to represent the inherent nondeterminism of AI problems in a functional program. The recursive formulation and leftmost outermost reduction feature of functional programs enable a natural formulation of depth-first search, but it is difficult to write a heuristic search program in a purely functional language, since heuristic search is inherently

history sensitive. In fact, heuristic search programs written in Lisp include many `setq` and `do` statements that are not pure functional primitives [246]. Because of their inability to represent nondeterminism and their inefficiency in dealing with large data structures, pure functional languages are often unsuitable for general symbolic applications. Nonetheless, they are very useful for deterministic symbolic applications.

3.1.2 Rule-Based Languages. Rule-based languages are associated with two major forms of knowledge representation: logic and production systems. The languages associated with these representations are described as rule-based because they emphasize the relation between a condition and an inference or rule.

Logic. In its modest form, a logic program is the procedural interpretation of Horn clauses or predicate logic [125, 126]. Some logic programming ideas, such as automatic backtracking, were used in the early AI languages **QA3**, **PLANNER**, and **MICROPLANNER** [18, 218]. The newer language Prolog is based on logic programming [37, 241]. A logic programming environment is a reasoning-oriented or deductive programming environment. Logic programming has recently received considerable attention because of its choice as the core computer language for the Fifth Generation Computer System Project in Japan [164].

The motivation of logic programming is to separate knowledge from control. However, logic programming implementations often include extralogical primitives to improve their run-time efficiency and provide additional methods for specifying control information; for example, in Prolog the Cut predicate is an extralogical control mechanism used to define a construct similar to the *if-then* of conventional languages. Furthermore, variables in a logic program are often non-directional—that is, they do not have to be defined as input or output variables at compile time, and their mode can be changed at run time according to the context. Thus, dependencies among subgoals are not defined at compile time, and static detection of parallelism is very difficult. The solution is to require users to specify the tasks to be processed in parallel. In Parlog [35], every argument is annotated with a mode declaration that states whether the argument is input (?) or output (^). In Concurrent Prolog [200], a read-only annotation (?) is used. Users can also distinguish between parallel AND and sequential AND by annotating them with "`,`" and "&" respectively.

Constructs can also be introduced to restrict parallelism until certain preconditions are satisfied. An example is Parlog's guarded clause. A guarded clause has the format `h :- g | b.`; where `g` is the *guard* of the clause and `b` is its *body*. Subgoals in the body can only be evaluated when all subgoals in the guard have succeeded and bound values have been committed to the body.

User specification of parallelism certainly detracts from the objective of declarative programming. This is a problem even in the restricted AND-parallelism (RAP) model [51], in which the user, although not required to

specify parallelism explicitly, must remain aware of the underlying computational model. Both mode declarations in Parlog and read-only annotations in Concurrent Prolog impose a fixed execution order on subgoals, an order which may be inefficient. Choosing the proper subgoals for a guard is sometimes difficult and is not guided by any general principle. The distinction between sequential AND and parallel AND—a linear order—is not sufficient to specify all precedence relationships, which form a partial order. Because of the nondeterministic nature of AI applications, users cannot identify all tasks that can be processed in parallel. A better symbolic processing language should detect parallelism at both compile-time and run-time.

Production Systems. The other major form of rule-based language that separates knowledge from control is based on the production system representation [28, 175]. A production system consists of a set of data and a set of rules that manipulate the data. A rule is composed of a left hand side (LHS) and a right hand side (RHS). The LHS is the antecedent or situation and represents the conditions necessary for applying the rule. The conditions are in the form of a Boolean combination of clauses [28]. The RHS is called the consequent and indicates a set of changes to be made to the data when the conditions of its LHS are met. Thus, a production system functions procedurally by first matching a logical condition and then modifying the data. Strategies are required for matching data conditions with the conditions of the LHS of rules and for resolving conflicts when more than one rule is matched. The conflict set is the set of antecedents and their bindings that match elements in the working memory. Production systems operate in a recognize-act cycle; first the recognize cycle computes the conflict set, and then the act cycle selects one matching production and makes the changes indicated in the RHS.

A popular programming environment for the implementation of production systems is the OPS5 system [28]. The OPS5 system highlights production system design issues. OPS5 employs data typing, and the working memory (data) is viewed as a separate entity from the production memory, where the rules are stored. OPS5 employs a Rete match algorithm which computes the conflict set but does not select a production on which to act. The algorithm stores the matching condition in the form of a tree. After the recognize-act cycle, the entire conflict set is not recomputed; instead the tree is updated via tokens that reflect the addition or deletion of elements from the working memory [28, 74]. OPS5 has two conflict resolution strategies: LEX and MEA. LEX orders the conflict set by recency of the time tags of the working memory elements that match the condition elements of the production rule. In contrast, MEA orders the conflict set by recency of the working memory element that matches the first condition of the matching conditions, even if it is not a maximum.

Production systems provide a natural programming paradigm for *if-then* programming environments such as those employed in expert systems. Unfortunately, algorithms using iteration or recursion are difficult to encode. Furthermore, rules are independent, and the structural organization of programs

requires special attention. Thus, it is difficult to develop large programs using production systems [186].

On Functional Versus Rule-Based Languages. The advantages and disadvantages of functional languages stem from the procedural and formal mathematical nature of the lambda calculus. Rule-based languages have complementary advantages and disadvantages that stem from their declarative nature.

Logic languages are more expressive because of their properties of nondirectionality of inputs and outputs, dynamic binding of variables, and nondeterminism. On the other hand, functional programming enables programmers to write more concise programs since higher-order functions permit quantification over individual data items as well as predicates and functions. Programs containing higher-order functions are also easier to understand and reason about.

Although logic languages are more expressive, their implementation in a parallel environment is more difficult because of the non-directionality of their variables. This flexibility complicates the detection of parallelism at compile time and results in the dynamic execution of logic programs. Current Prolog systems also lack a means to describe the termination of computations on conceptually infinite data structures and the concept of function evaluation, which makes the logic base nontransparent. The run-time behavior of pure functional programs is much simpler to control than that of first-order logic programs, particularly in a parallel context. Techniques such as graph reduction and data flow have been studied for use in parallel evaluation of pure functional languages.

It would be advantageous if the simple control mechanism of functional languages could be applied to languages with the great expressive power of logic languages. Considerable efforts have been devoted to the combining of functional and logic programming [52]. Some researchers are trying to simplify logic languages by introducing directionality of information in logic programs [187]; however, this approach will reduce the expressive power of logic programs to that of first-order functional programs.

The alternative approach is to extend functional languages so that they have the expressive power of logic languages while retaining their underlying functional simplicity. The addition of unification to the Hope language is such an example [47]. Subrahmanyam and Yau have proposed FUNLOG, a language that integrates functional and logic programming. FUNLOG provides the programmer with the flexibility to choose between a backtrack-free computational framework and a logic computational framework. *Semantic unification* is the basis for the integration of function and logic, and can be used to replace conventional unification in logic programming [216]. TABLOG, a new approach to logic programming designed by Malachi, Manna, and Waldinger, is based on first-order predicate logic with equality and combines rule-based and functional programming. The use of this richer and more flexible syntax overcomes some of the shortcomings of Prolog syntax [146]. Other languages that com-

bine features of Prolog and Lisp include LOGLISP, QLOG, POPLOG, Qute, and Lambda Prolog [187].

3.1.3 Object-Oriented Languages. New languages and programming systems are being developed to simplify AI programming radically. *Object-oriented programming* holds promise as a programming framework and can be extended to concurrent systems, databases, and knowledge bases.

In conventional software, data and procedures are the main focus of the representation and are treated as separate entities. The choice of procedures and data is made by the programmer. In an object-oriented system, there is only one entity: the *object*. Objects may be manipulated like data, or describe procedure-like manipulation, or both. Processing is performed by sending and receiving messages to and from the object that possesses the appropriate information. A selector in the object specifies the kind of operation to be performed. Message sending is uniform; a message specifies only the result that is required by the sender, with no information on how it should be accomplished. Objects respond to messages by using their own procedures, called *methods*, to perform operations. Since all communication is via messages, one method may not "call" another method. Any knowledge-representation scheme can be used to represent the procedural knowledge, although most implementations of object-oriented languages have employed procedural representations of control knowledge stored within the object.

In addition to objects and messages, object-oriented languages may employ the concepts of class and instance. A class is a description of similar types of objects; classes allow attributes of objects to be shared by providing a mechanism for inheritance or implicit sharing. Inheritance is used to define objects that are almost like other objects. Thus, classes provide an interface for the programmer to interact with the definitions of objects.

Data abstraction is an important concept that is supported by message sending. Object-oriented languages support both the management and collection of data through *abstract data types*, and the composition of abstract data types through an *inheritance* mechanism.

The requirement of typed data abstraction with inheritance is explicit and definitive, suggesting that object-oriented programming should be characterized by its type mechanisms rather than its communication mechanisms. In a sense, object-oriented programming can be defined as

$$\text{object-oriented} = \text{data abstraction} + \text{data types} + \text{type inheritance}$$

The object-oriented programming paradigm is used mainly for organizing knowledge domains but is permissive in its methodology for communication. The message/object model provides no new leverage for expressing concurrent problems. Concurrent models, operating systems, and coordination tools can be built from such lower-level objects as processes, queues, and semaphores.

Early exploration of object-oriented programming was found in Simula [42]. A more contemporary object-oriented language which has received a great deal of attention is Smalltalk [81], and the object-oriented languages also include LOOPS [210], Actor [6], CommonObjects [208], OIL [48], and others [242]. Recently, CommonLoops has been suggested as a standard for object-oriented extensions to Common Lisp [23].

The *Actor* model, developed by Hewitt at MIT, combines a formalization of object-oriented language ideas with the effects of parallelism [98]. An actor is the analog of a class or type instance. Computations in the Actor model are partial orderings of events, inherently parallel and having no assignment commands. The language Act3, based on the Actor model, combines the advantages of object-oriented programming with those of functional programming [5]. The Apiary network architecture has been proposed to support the Actor model [99, 100].

3.1.4 Mapping Applications into Software. Software development, an active area of research in software engineering, is the process of mapping an application into a language. The process begins with the selection of a solution technique, a selection that includes the choice of a knowledge representation and a method for solving the problem within that representation. Naturally, the available languages impact this choice. In life cycle models of software development [185, 201] the choices may be referred to as requirement analysis.

Software development environments can be classified into four generations [186]: discrete tools, toolboxes, life cycle support and knowledge-based tools, and intelligent life cycle support. Discrete tools, typical in the 1960s and 1970s, are individual debuggers. Toolboxes are integrated packages of tools; Interlisp [224] is the most prevalent example. Life cycle support and knowledge-based tools were developed in the 1980s. Life cycle support refers to software-development environments suitable for each stage in the design cycle; knowledge-based tools incorporate domain knowledge to provide interactive assistance to the programmer. Finally, intelligent life cycle support, a topic for future research [186], provides integrated knowledge-based support for all stages of the software-development cycle. Software engineering environments for distributed software development are another important area of research [201].

As software complexity increases, verification, validation, and the enforcement of a structured discipline become more important to the production of reliable software. Although a number of symbolic languages, such as Prolog, have been criticized for lack of structure, a programming style of hierarchical development can be followed as in conventional structured programming languages such as Pascal. Verification and validation of software written in symbolic processing languages are controversial. If programs are treated as algorithms with well specified requirements, then verification and validation procedures are similar to those of programs written in traditional languages,

TABLE 1.8 Features of Two Example Languages that Can Be Supported by Hardware.

Lisp	Prolog
Data typing	
Function calls	
Recursion	
List structures	Condition matching
Garbage collection	Database functions
Individual commands	Search
CAR, CDR, etc.	Search strategy
Parallelism	Backtracking mechanism
FUTURE, etc.	Unification
Application support	Parallelism
Application dependent	Modes, guards, etc.
Database support	Application support
Transitive closure	
Others	

and techniques such as test case generation and path testing can be applied. It is, of course, difficult to test the validity of the knowledge used in a program, since it may be heuristic and fuzzy in nature. The same criticism, however, can be made of programs written in traditional languages, whose validity will depend to a great extent on the experience of experts and the procedures used in deriving the knowledge. Systematic knowledge-capture tools will help, but cannot guarantee proper collection and maintenance of consistent knowledge from multiple experts.

3.2 Hardware Architectural Support for Symbolic Processing

The choice of knowledge representation schemes and software languages largely dictates the desirable hardware architectures. This section discusses the desirable intermediate hardware designs of a symbolic processing architecture. These hardware designs can support language-specific features or primitive symbolic operations such as sorting and pattern matching. Table 1.8 shows some features of Lisp and Prolog that require hardware or software support.

Hardware features, like languages, can incorporate many well established design philosophies: for example pipelining, parallel processing, microprogramming, and redundancy. Consider, for example, the hardware support of type checking in Lisp. Figure 1.5 illustrates the role that design concepts and design requirements play in a hardware architecture, though hardware designers may not utilize such an approach in a real design.

Once a set of hardware and software alternatives has been enumerated, a subset of them must be selected for incorporation into the system. The

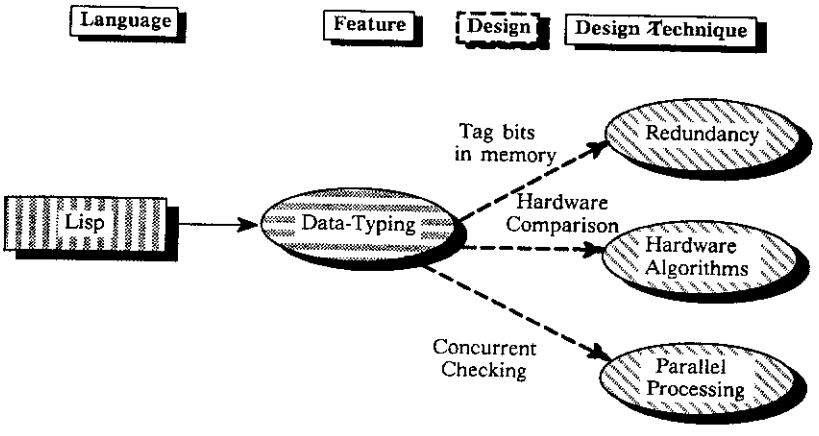

Figure 1.5 Design process for data-typing hardware.

choice will depend on the design philosophy of the system (see Section 4). The selection among competing structures (alternative designs that perform the same function) is made on the basis of anticipated gain in performance versus anticipated cost; both may be difficult to estimate, especially when the structures are not commercially available components. An approximate model is often used to guide the selection.

The hardware architectures currently used to support symbolic computation can be classified into microlevel hardware features, subsystem-level hardware features, and system-level designs. A microlevel architecture is generally a piece of hardware designed to support a feature of the language or primitive operations in the knowlege representation scheme. It is specialized and does not, in general, perform useful computations outside of its role in the system. A subsystem-level architecture is an architecture that performs a complete and useful function, but is often included in a larger system. A system-level design provides a complete hardware/software solution to a symbolic processing application. Microlevel and subsystem-level designs are discussed in Sections 3.2.1 and 3.2.2; system-level designs are presented in Section 4.

3.2.1 Microlevel Hardware Features. A microlevel hardware architecture is a unit dedicated to the support of a specific symbolic processing technique. As the number of features in symbolic applications and their languages is large, the number of corresponding possibilities for microlevel architectures is very large. Table 1.9 presents some microlevel architectures and the features that they support.

Five microlevel architectures are discussed in detail in this section: stacks, data tags, garbage collection, pattern matching, and unification. The remainder

TABLE 1.9 Functions that Can Be Supported by Microlevel Architectures.

Function	Example Architectures
Function calls and recursion	Hardware stacks Register windows Fast memory techniques
Data typing	Memory tags Concurrent tag-checking hardware
Sorting	VLSI sorter
Set intersection	Marker-passing systems such as NETL
Pattern matching	Systolic arrays Content addressable memories Finite state automata
Best-matching	Value-passing system such as Thistle, neural networks
Garbage collection	Hardware pointers Reference counters

of the section is devoted to a discussion of emerging technologies employed in the construction of microlevel hardware.

Hardware Stacks and Fast Stack-Access Techniques. Stack architectures support function calls. This is especially useful for Lisp and other functional programming languages. The Symbolics 3600™computer [161] contains three stacks: the control stack, the binding stack, and the data stack, used to support tail recursion and shallow binding. To speed access to the control stack, up to four virtual-memory pages of the stack are held in a dedicated fast-access 1K-word memory referred to as the stack buffer. The stack buffer contains all of the current function environment or frame plus as many of the older frames as fit. A second stack buffer contains an auxiliary stack for servicing page faults and interrupts without disturbing the primary buffer.

In ALPHA, a commercially available Lisp computer produced in Japan [94], a hardware stack is divided into four 2K-word physical blocks. Variables or arguments of a Lisp function are stored in different locations of cell space or stack. A great deal of time is required to search for a free variable, especially in the case of deep binding. To speed the evaluation of functions, the hardware stack is designed to support value caching. The value of a variable is fetched from the environment stack when it is first accessed during a function evaluation, and is stored in the value cache. Subsequent accesses will refer to values in the value cache. When the function exits, the variables it used

in the value cache are marked invalid. Virtual stacks are also used to avoid the overhead of using a single stack and having to swap the entire stack when process switching.

Fast stack operations are also useful for implementing Prolog. Data is often pushed on and pulled from stacks in backtracking operations. However, the frames of a caller clause may be deeply buried in the stacks; hence, a stack architecture may not be adequate. In the Personal Sequential Inference Machine (PSI, a product of the Japanese Fifth Generation Computer Systems Project), a cache memory is employed. Several operations suited to stack access are carried out in the cache memory.

Tagged Memory. A conventional von Neumann computer does not distinguish between data and program. Both are stored as fixed-size binary words. Meaning is not inherent in the contents of storage but defined by the program manipulating the storage. A tagged architecture, however, relies on self-identifying representations at all levels of storage. Although tagging has been employed since the 1960s, early design philosophy considered tagging a relatively unimportant and expensive peripheral concept. Tagged memory is now a key feature in many symbolic processing computers. Symbolic architectures often require identification of different physical and abstract data types: for example, integer, character, event, and garbage. During processing, the different operands employed in the computation must be identified. The tagging of data improves real-time type checking.

The most common hardware support for data tagging is the allocation of extra bits in each word to represent its data type. Data type checking at run time may be supported by additional hardware and overlapped with regular processing. The speed of a symbolic computer often depends on how effectively it emulates a tagged-memory architecture [50]. Special hardware for data type tagging has been estimated to improve the performance of Lisp computers by as much as an order of magnitude. Data tagging also supports garbage collection, facilitates better register utilization, reduces memory traffic, and simplifies the design of cooperating parallel processors and specialized functional units [71]. Data tagging is also essential in untyped languages, since the programmer does not specify the data types of instructions. For instance, the programmer need not specify the type of an add instruction as integer, long, real, or double. The type of adder used at run time depends on the types of the operands, which are specified by tags and detected at run time.

Data tags can be used to represent information other than data types. In the Classifier Machine [31], no addresses are used at all; instead tags are used to connect classifiers to each other. This no-addressing technique makes the Classifier Machine startlingly different from classical von Neumann architectures.

Garbage Collection. Garbage collection is the process of identifying memory cells whose contents are no longer needed by the computation in progress.

In this case, memory cells are contiguous groups of at least one memory word. The process marks cells as available for future use and compacts free memory into contiguous blocks. It has been estimated that 10 to 30 percent of the execution time of large Lisp programs is spent on garbage collection. As garbage collection often requires large continuous segments of time from the CPU, the impact on interactive or real-time systems is great.

Early techniques for automatic garbage collection centered around the use of reference counts [150]. Each cell had an extra field that indicated the number of times the cell had been referenced. The reference count was updated each time a pointer to the cell was created or destroyed. When the reference count reached zero, the cell could be reclaimed as garbage. These techniques were intuitively simple and distributed the processing overhead evenly throughout the task processing time. However, large amounts of space and time may be required for the use of reference counts, although some of the overhead can be shifted to compile time [19]. Furthermore, cyclic structures cannot be reclaimed. Generation scavenging is an important technique which reduces the overall rate of garbage collection by using different garbage collection rates for memory areas of different ages (or generations) [233].

More recent research has focused on parallel garbage collection methods. Parallel garbage collection is garbage collection that is performed concurrently with program execution. Two processing entities are involved: the *mutator* and the *collector*. The mutator is responsible for program execution, and the collector for garbage collection. The techniques often use coloring cells [57, 127], or divide memory space into (two) distinct regions [17, 142]. In the first case, tagging can be a useful microlevel hardware feature. Parallel garbage collection processors can be designed with very simple and fast components and hence are not a bottleneck in the system. Hibino [102] proposed a design whose collector processor cycle time was 200 ns—6 times faster than typical processor cycle times.

Pattern Matching Hardware Support. Addition and multiplication are the mainstay of scientific computations. Similarly, pattern matching is the basic operation of symbolic processing. Two of the major tasks for which a pattern matcher may be employed are finding entries in a database and choosing the next operation to execute. For example, determining the applicable rule in a production system is a pattern matching problem. Empirical results show that 90 percent of execution time in a production system used for an expert system can be spent in the matching phase [75]. Therefore, hardware pattern matching support can simplify the programming task and improve run-time efficiency.

In most symbolic representations, symbols are represented by strings. Conventional string-matching hardware can be classified into four categories. The first category is associative memory, which although straightforward does not easily handle strings of variable lengths. The second category is the cascaded logic memory array, also called a cellular array [137]; each charac-

ter in a pattern-string is stored in a cell and is compared with a character in the input string. The third category is the finite state automata (FSA) method that uses a transition table to perform complex string matching. Finally, there is the dynamic programming technique that uses statistical characteristics of the general pattern to determine the parameter table for proximity matching. Mukhopadhyay [165] presents a survey of techniques for hardware support of pattern matching, and more recently further techniques have been proposed [69, 220].

Pattern matching in symbolic processors differs from conventional database retrievals in that many symbolic applications contain widely varying field lengths and that uncertainties in data forbid exact matching of patterns. The pattern matching hardware for a symbolic processor must be tailored to the representation(s) for which it will be used.

When matching under uncertainties, *best-matching* is required. Best-matching structures search for the pattern which best matches the defined objective. Best-matching using associative memories has been explored [184], and neural networks also provide potential for performing best-matching (see Section 3.2.2). Kanerva's Sparse Distributed Memory (SDM) is a system designed for best-matching [116]. The proposed prototype consists of a virtual memory that is addressed by a 1024-bit address, and a small physical memory. Each word in the physical memory has a 1024-bit address field and multiple data fields. When a memory address is given, all locations in the physical memory with addresses that differ by less than 450 bits of the given address are accessed, and the corresponding data fields are combined together into a single response. The 450 bits are chosen so that approximately 0.1 percent of the memory words will respond on the average for a physical memory of 4 Mwords.

In a semantic network representation, pattern matching and other functions can be performed in parallel using a marker passing operation [68]. A high-bandwidth communication channel is important for this type of pattern matching.

In a forward chaining rule-based production system, the objects to be matched are constants, and multipattern multiobject pattern matching is required. The Rete Match Algorithm is an efficient solution to this problem [74], and a number of hardware implementations of this technique have been proposed, including tree architectures [202, 212], an SIMD Cellular Array Processor (CAP) [26], and a tagged token data-driven multiprocessor [78]. The key architectural requirements to support the Rete Match Algorithm in parallel production systems are a memory to maintain information across multiple recognize-act cycles, and the proper granularity of parallelism [86].

Unification Hardware. Unification, a form of pattern matching, is the fundamental technique in logic programming. It determines if two terms can be made textually identical by finding a set of substitutions for variables in the terms,

and replaces all occurrences of each variable by that variable's substitution. In general, both terms to be resolved in unification are allowed to contain variables; hence, unification can be thought of as a bidirectional pattern matching operation [126]. Since unification is applied extensively and is known to consume over 60 percent of the execution time in sequential executions, additional hardware or firmware support is desirable.

The primitive operations in unification are: (1) search for the called clause, (2) fetching of arguments of the caller and called predicates, and (3) examination of equivalence of arguments. Fast memory access is required to carry out unification in hardware. To support dynamic memory allocation, an efficient garbage collection technique is also required. Finally, hardware support for data type checking can also speed performance.

Research in this area has concentrated on string-matching hardware [204], uniprocessor machines [170], special unification chips [188], and pipelined unification [169]. To reduce the required memory space and improve performance, structure sharing [29] and techniques for structure copying [156] have also been explored.

The Parallel Inference Engine (PIE) developed at the University of Tokyo employs special hardware, referred to as UNIRED, for unification and reduction. UNIRED is characterized by the following features: tagged memory, high-speed local memory that can be accessed in parallel, parallel hardware stacks, and dedicated internal busses. The unify processor fetches a goal from a memory module and candidate clauses from definition memory, attempts to unify them, generates new goals, and returns the goals to the memory module.

Parallel unification is also an area of great interest. Parallel unification can be performed either by simultaneously unifying each term pair in two atoms, or by finding many possible unifications concurrently [164]. Unfortunately, the unification problem is proven to be log-space-complete in the number of processors. This means that it is not possible to perform parallel unification in $O(\log_k n)$ time using a polynomial number of processors, where k is any constant and n is the total number of nodes and edges of the directed acyclic graph representation of the clauses. It has also been shown, however, that near-linear speed-up can be achieved in parallel unification. Thus, from a practical perspecive, unification algorithms are parallelizable [237]. Array architectures for parallel unification have been proposed, such as the Cellular Array Processor (CAP) [26] and others [204]. A mesh-connected array of unifiers has been proposed to exploit AND-parallelism in unification and may achieve superlinear speed-up [203].

VLSI and Emerging Technologies. Very Large Scale Integrated circuit or VLSI technology has been a major factor in the cost reduction and increased functionality of symbolic processing systems. The high degree of space-time complexity in AI and symbolic computations has necessitated the use of both parallel processing and VLSI technology. The development of specialized

microelectronic functional units is among the major objectives of the Japanese Fifth Generation Computing project [77], MCC [73], and DARPA's Strategic Computing projects [45].

VLSI technologies allow a single chip computer to be realized. Although many functions can be implemented on a single chip, the size of the chip and the number of input/output pins are usually limited, and chip area has to be carefully allocated to achieve the highest performance. The Reduced Instruction Set Computer (RISC) is a highly popular design approach that carries out only the most frequently used instructions in hardware and the rest in software [179]. The small chip area required by the control unit of a RISC computer enables it to be placed on a single chip. It has been found that a large number of registers on a single chip is a good design tradeoff that reduces the overhead of swapping registers in context changes.

SOAR, or Smalltalk on a RISC, is a project to develop a RISC chip for Smalltalk-80 [235]. SOAR design details a 32-bit NMOS microprocessor containing 35,700 transistors that runs roughly 400 ns per instruction. Cycle time may be decreased to 290 ns if 3-micron lines are used. FAIM-1 is another project that designs RISC chips to carry out specialized functions in the system [12].

Specialized symbolic processing functions can also be carried out in hardware. An example is the Texas Instruments Lisp chip with over 500,000 transistors on a 1 cm^2 chip, implementing approximately 60 percent of the functions of a Texas Instruments Explorer [147].

Many existing computers for symbolic computation employ VLSI technology. The major building block of the Connection Machine CM-1 and CM-2 is a custom VLSI chip containing 16 processor cells [103]. The chip is implemented on a CMOS die about 1-12cm212 in area. There are approximately 50,000 active devices. Although each addition takes approximately 21 μs, an aggregate maximum rate of 2500 MIPS or 5000 MFLOPS can be achieved with 64K processors implemented in 4K processor chips and 4K floating-point chips.

Cellular array structures are a widely studied technique that can take advantage of the available VLSI technology and exploit data-level parallelism in many symbolic processing problems. The Cellular Array Processor (CAP) is an example in this class [26], and systolic cellular hardware design has been explored for performing unification [204].

In the near future, other emerging technologies may become cost effective for implementing computers for symbolic processing. They include GaAs circuits, wafer-scale integration (WSI), and optical computing techniques.

GaAs circuits have design requirements similar to those of conventional semiconductor circuits, but are much faster. Switching speeds on the order of 10 ps have been reported in High Electron Mobility Transistor (HEMT) GaAs circuits [130]. Gate propagation delays are typically on the order of 200 ps. Unfortunately, the scale of fabrication of GaAs circuits is subject to limited

TABLE 1.10 Microlevel Architectures and Their Significance.

Architecture	Significance
Stacks	For function calls and value binding; more than one stack may be used
Tagging	Data type checking; up to an order of magnitude speed improvement for Lisp
Garbage collection	Reclamation of usable storage; accounts for 10 to 30 percent of run time in Lisp
Pattern matching	Fundamental operation; up to 90 percent of execution time in production systems
Unification	Type of pattern matching for logic; over 60 percent of execution time in sequential logic programs

size and greater numbers of defects; these problems increase costs, already high [154].

WSI is the integration of multiple circuits on the same wafer in order to avoid the high performance and cost penalty of off-chip connections. Unfortunately chip yield in manufacturing is low; the yield of complete wafers would be negligible, although techniques such as Focused Ion Beam (FIB) repair increase yield. Low yields make WSI more suitable for implementing distributed knowledge representation schemes, such as neural networks, in which the loss of a small fraction of the distributed knowledge may not be critical. When GaAs and WSI are combined, it is possible to implement a 32-bit GaAs processor on a single wafer [154]. Design of such a processor for production systems has been explored [131].

Optical processing can be used in the same way as silicon gating. Switching speeds on the order of 5–10 ps are possible in optical gates. Optical circuits do not have the penalty of capacitance [244] and can communicate with low propagation delay and no interference. Two forms of optical processing have been developed: arrays of light rays [21, 79] and optical crossbars [148]. The use of optics in storage media may greatly improve performance in symbolic systems with erratic memory behavior. It has been proposed that optical techniques have the potential for improving data- and knowledge-base processing speeds by two orders of magnitude [21].

Table 1.10 summarizes the microlevel architectures presented in this section. Each type of architecture is accompanied by an example application, and remarks about its significance in supporting that application.

3.2.2 Subsystem-Level Architectures. The subsystem-level architecture represents an intermediate level between microlevel and complete-system designs. In this section, we identify three techniques for classifying the subsystem-level architectures that support different types of knowledge processing.

Three different types of subsystems will emerge from this analysis: data- and knowledge-base machines, inference engines, and neural networks.

Control flow, data flow and demand flow are the names of three important approaches that are used in the design of subsystem-level architectures [236]. Their definitions and relative advantages and disadvantages are summarized in Table 1.11.

Knowledge representation plays an important role in the way that a complete system integrates its components. The Japanese FGCS project emphasizes logic representations and stresses the separate development of knowledge-base and inference engine, which are integrated only after development [9]. This approach to the system architecture is illustrated in Figure 1.6a.

The Connection Machine [103] is designed with an alternative perspective and reflects a semantic-network knowledge representation and memory-based reasoning. All the knowledge in the system is embodied in a collection of facts; no intentional knowledge or rules are employed. The database is implemented directly on the architecture, and inferences are carried out in software and

TABLE 1.11 Control-Driven, Data-Driven, and Demand-Driven Computations.

	Control Flow (control-driven)	Data Flow (data-driven)	Production (demand-driven)
Definition	Conventional computation; token of control indicates when a statement should be executed	Eager evaluation; statements are executed when all of their operands are available	Lazy evaluation; statements are only executed when their result is required for another computation
Advantages	Full control	Very high potential parallelism	Only required instructions are executed
	Complex data and control structures are easily implemented	High throughput	High degree of parallelism
		Free from side effects	Easy manipulation of data structures
Disadvantages	Less efficient	Time lost waiting for unneeded arguments	Does not support sharing of objects with changing local state
	Difficulty in programming	High control overhead	Time needed to propagate demand tokens
	Difficulty in preventing run-time error	Difficulty in manipulating data structures	

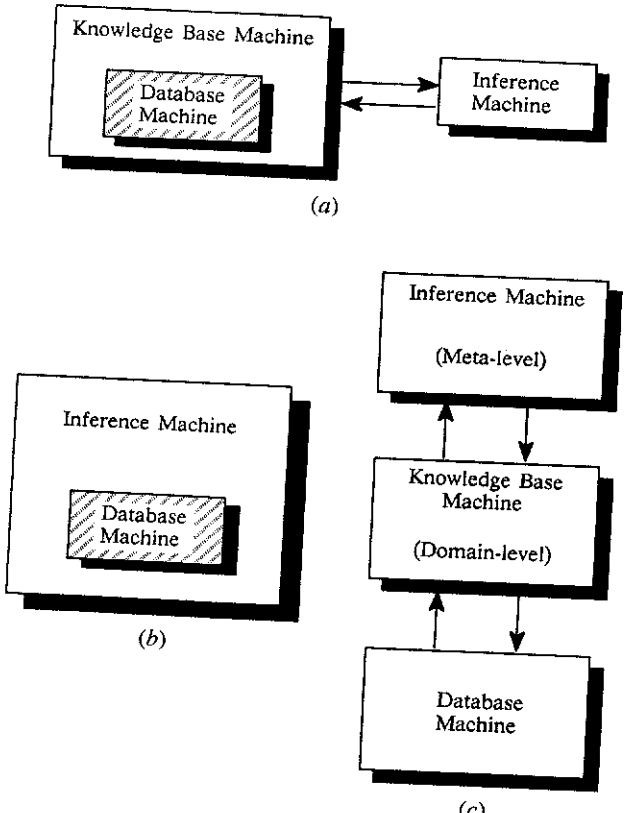

Figure 1.6 Relationships among knowledge-base machine, inference machine, and database machine: (*a*) Japanese FGCS model; (*b*) connection machine model; (*c*) hierarchical model.

message exchanges. This perspective on system design is illustrated in Figure 1.6b.

The last design perspective is hierarchical in nature, as shown in Figure 1.6c. An inference machine handles all meta-level inferences, while a knowledge-base machine uses rules to manipulate domain knowledge. A separate database machine carries out search and selection operations on the domain knowledge.

Database Architectures. Early studies of specialized database architectures emphasized the use of parallelism; such designs include CASSM [215], RAP [178], and DIRECT [56]. Some later systems, such as the Connection Machine, are designed with massive parallelism for symbolic applications and can be used for a number of specialized database functions [240]. Hawthorne and Dewitt [93] present a comparison of several early parallel database comput-

TABLE 1.12 Differences between Databases and Knowledge-Bases.

Issue	Database	Knowledge-Base
Contents	Collection of data facts	Higher level of abstraction
		Classes of objects
Complexity	Stored items are simple	Stored items are complex relations
Time and Dependence	Data changes over time	Knowledge changes less frequently, except for situation knowledge
Size	Large number of facts	Fewer relations on classes of objects
Use	Operational purposes	Analysis, planning, etc.

ers, and Malabarba [145] gives a survey of commercially available database computers. Commercial manufacturers of database computers include Britton-Lee, Hitachi [227], International Computers Limited [15], and Teradata [171].

These systems may not function well for some applications because the bottleneck in database retrieval is disk input/output and not processor cycles [214]; hence, intelligent database processing using application-dependent knowledge and indexing may be preferable to massive parallelism [214].

Knowledge-Base Architectures. The objectives and requirements of a knowledge-base computer are different from those of a database architecture. The most prominent differences are noted in Table 1.12. An evolving knowledge-base subsystem should include a mechanism that either rejects inconsistent data or rules or performs truth maintenance when such data or rules are inserted. Support for the inference mechanism is also desirable and may take the form of an automatic rule-selection mechanism, logic support, or special hardware for operations such as joins and projections of relations. Finally, the interface to the host computer should be intelligent and may draw on the resources of the knowledge-base.

The issues in the design of a knowlege-base computer include:

Storage and Manipulation of Intentional and Extensional Data.

Extensional data are data that represent facts, that is, statements with no quantified variables. Intentional data are general facts or rules. In

Figure 1.6c, the extensional and intentional data are stored and manipulated separately. In Figures 1.6a and 1.6b, they are processed by the same physical entity.

Relational Operations.

As already mentioned, hardware and software support must be provided.

Hierarchical Storage.

Knowledge can be classified into categories by its degree of generality: from fact through the most general metaknowledge. Access characteristics are highly dependent on the type of knowledge. A hierarchical storage for metaknowledge may be used to exploit the knowledge structure efficiently.

Access-Control Algorithms.

A knowledge-base subsystem may be required to control access to its contents for security, integrity, and concurrency control.

Parallel and Distributed Processing.

Database updates are history-sensitive. The choice of the best technique(s) for exploiting parallel and distributed processing in knowledge-base systems remains an open issue. The data flow approach may be helpful.

In many applications, a database computer may be integrated with an existing host to form a knowledge-base computer: for example, the Intelligent Information Resource Assistant developed at System Development Corporation [118]. The system consists of a Britton-Lee IDM 600 backend database computer, a Xerox 1100 workstation acting as a logic-based deductive engine, and a VAX 11/780 computer as a file and print server. This prototype has demonstrated that a knowledge-base computer can be easily constructed from existing hardware components.

The Japanese FGCS project has developed Delta, a combined knowledge- and database computer. The motivation behind a specialized design is that the integrated system calls for a performance that cannot be met by commercial components. The system consists of a control processor, a relational database engine, and a hierarchical memory. The control processor translates commands (received from an interface processor communicating with a parallel inference machine) into subcommands that the relational database engine can obey. The relational database engine communicates through a data path with the hierarchical memory, which is composed of semiconductor and magnetic disk storage [166].

Hardware Support for Inference Engines. Inference engines are key components of knowledge processing architectures. Their structure is highly dependent on the knowledge representation and programming language employed.

An important consideration when designing hardware support for inference engines is the architectural support for searching the knowledge base. Deduction and search have been the dominant paradigms for machine inference over the last 30 years. As discussed in Section 2, the best approach to search is the development of superior heuristics combined with efficient hardware. There follow some of the key reasearch issues in search architectures. (More general issues on parallel processing were presented in Table 1.6.)

Prediction of Performance.

A major difficulty in developing search-based inference engines is our inability to estimate their performance without execution of the search. This is because of both the nondeterministic nature of searches and anomalies in parallel search algorithms [238].

Space-Time Tradeoff.

There is a space-time tradeoff in using heuristic knowledge. Very accurate heuristic functions may require greater amounts of space and computation time than less-accurate ones. This relationship must be understood for the design of effective search subsystems.

Architectural Support for Machine Learning.

Heuristic functions used in search algorithms should be improved over time by automatic learning methods. Architectural support for nonmonotonic processing may be helpful.

Management of Large Memory Space.

Heuristic search strategies may require large amounts of memory space to store intermediate results and the heuristic information. The tradeoff between the effectiveness of a search strategy and its overhead must be considered. Techniques for efficient memory management tuned to the search behavior are vital in such a system.

Granularity of Parallelism.

The proper choice of granularity is difficult to determine at design time because of the dynamic nature of the problem. Granularity may have to be varied at run time when more is known about the application.

Scheduling and Load Balancing.

Because many search problems are nondeterministic, direct mapping of a sequential search strategy into a parallel system may not produce the best performance. The key to effective scheduling is the proper order of execution; it is not enough to keep the available processors busy. Counter to intuition, depth-first search is sometimes preferable to best-first search in real systems when memory constraints are considered [249]. Conditions for the sequential search strategy may also have to be relaxed to accommodate the architectural constraints [238]. For instance, when performing a heuristic search, it may not be desirable to select the subproblem with the minimum heuristic value when the overhead of selecting subproblems distributed in local memories of multiple processors is high. Selecting the local subproblem with the minimum heuristic value will suffice in most cases.

Communication of Pruning Information.

When the search space is explored in parallel, excess computation may be performed if pruning information cannot be shared among the processors. In general, a tradeoff exists between search efficiency and communication overhead.

Microlevel hardware features discussed in Section 3.2.2 are often components of the inference subsystem. Data flow, control flow, and demand flow techniques (see Table 1.11) have been employed in designing hardware supports for inference engines for Lisp and logic [10, 114, 167]. An example of a combined data-driven and demand-driven approach is demonstrated in ALICE [43] and Rediflow [117]. Although the theory behind data-driven and demand-driven computations appears very promising, it offers no clear solution for design problems such as the selection of the granularity of parallelism. Furthermore, the proper tradeoff among demand flow, data flow, and control flow in an inference computer remains an open problem.

Artificial Neural Networks. Automated computation based on a neural network design philosophy originated many years ago. One pioneering contributor was Rosenblatt, who developed the concept of perceptrons [189]. Recently, a great deal of interest has revived in this area. Neural networks, besides capturing the imagination by their resemblance to the human brain, offer great potential for automated learning. Because information is stored in the connections between neurons, neural networks may also be called connectionist systems.

A neural network can be viewed as a collection of eight components: a set of processing units, a state of activation, an output function, a pattern of connectivity, a propagation rule, an activation rule, a learning rule, and an environment [190]. These components are described in Table 1.13. The first

TABLE 1.13 Neural Network Components.

Component	Description
Processing units	Three types: input, output, and hidden
State of activation	Vector of the activation levels of the units in the system
Output function	Function, on the activation level of a unit, which produces the unit's output; may vary between units, but most systems are homogeneous
Pattern of connectivity	The connections determine the performance and function of the system
Propagation rule	A way of combining outputs of units and pattern of connectivity into an input for each unit; usually is a weighted sum of the inputs and the excitatory (+) and inhibitory (−) connection strengths
Activation rule	A function for determining the new activation level of a unit on the basis of current activation and inputs to the unit
Learning rule	Three types: develop new connections, abandon old connections, modify weights; only last has been pursued; almost all learning rules based on Hebbian learning rule
Environment	In which computing engine functions

design consideration is how knowledge should be represented in the set of processing units. There are two possible approaches: local and distributed. A local representation allocates each concept to its own unit. Hopfield's network for solving the traveling salesman problem is an example of a local representation in a neural network [110]; however, although it may solve the problem efficiently, it does not perform learning. In contrast, each unit in a distributed representation participates in the storage of many concepts [106]; in this way, a processing unit in a distributed representation stores an abstract feature at the microlevel [190].

Each unit in a neural network has an associated activation level that may be analog, analog and bounded, or discrete. A unit's output is a function of its activation level—usually a threshold or sigmoidal function. The output of each unit is distributed to a set of processing units. The influence of a unit's output on another unit is determined by the weight of the connection

between the units. Finally, a node's activation is modified by the activation rule, a function of inputs, their relative connection strengths, and the current activation level.

Classes of neural networks may be distinguished by their learning paradigms [190]. Networks in the associative learning class learn associations between inputs and their desired outputs, whereas regularity detectors learn to recognize interesting patterns in the input. The literature on this topic includes overviews of designs for neural networks of both classes [4, 144] and an extensive review of learning techniques in alternative neural network strategies [106, 107].

The use of neural networks has been proposed for a variety of applications, although they are not yet widely used in practice. A neural network can easily be adapted to form an associative memory with capability for inexact matching with speed and accuracy in the presence of noise [53]. Neural networks can also be used for speech recognition [198] and for vision tasks such as letter recognition [153]. Neural networks have been applied to combinatorial search problems such as the traveling salesman problem [110, 111], although the solutions of large search problems by neural networks are not yet of as high a quality as those produced by good digital algorithms [20]. Neural networks are also useful for some strategy-learning tasks, and experiments have been performed with a balance-control system [11].

The design of neural networks is still plagued by a number of difficult problems. Firstly, a neural network must be trained for a given application and must be retrained when the system parameters change; there is no systematic method for taking a neural network trained for one application and generalizing it for another. Secondly, all known learning algorithms require extensive training for good performance: for example, over 9,000 learning sweeps of 40 phases each are required to train a shifter network using the Boltzmann Machine learning algorithm [107]. Moreover, learning speed depends on the configuration of the neural network, which cannot be selected systematically. Dedicated hardware to emulate various configurations of neural networks and map the inner-loop operations into analog instead of digital circuits can improve the learning speed significantly, however. An improvement in speed by a factor of a million has been demonstrated using hardware emulation [8]. Thirdly, extremely large neural networks cannot be built with the current technology, in spite of numerous industrial efforts (for example, AT&T's R3 chip has a word size of 1048 bits and performs learning by back propagation). It has been estimated that by the year 2010 it will be possible to build 4.5 million neurons onto one wafer using 0.25 micron lithography and 250 million transistors [62]. It is therefore unlikely that a complete symbolic processor as intelligent as the human brain (with over 10 billion biological neurons) can be built with neural networks alone in the near future. Finally, because neural networks are limited in size, the problem must be partitioned so that part of it can be learned by traditional method and the other part by neural processing. However, knowledge representations in neural networks are

drastically different from procedural and declarative representations in conventional symbolic processors, and systematic methods to integrate them are still missing.

4 COMPLETE SYSTEMS

Up to this point, we have focused on individual techniques for effective symbolic computations. In this section, *complete systems* for symbolic applications are classified into single-processor systems, parallel computers, and connectionist systems, and, as before, we analyze the role of individual components in the design of complete systems.

There are two prevailing trends in designs. With the increasing complexity of many symbolic processing applications, the emphasis of knowledge representations, software languages, and support is shifting away from strongly procedural techniques and toward distributed and declarative computing environments. Parallel processing is another obvious trend in the design of computer systems for symbolic processing.

Depending on its starting point, system design can be classified as following either a top-down or a bottom-up approach. Top-down design begins with the specification and analysis of the application. A knowledge representation is then designed and tailored to the needs of the application. Finally a language is designed, and the system is mapped into software and microlevel and subsystem-level structures. This process may be iterated many times if the functional requirements of the design cannot be implemented using the current technology. The FGCS project used a top-down design approach.

In contrast, for a bottom-up design, the designer first selects the technology and design options, such as data- or demand-driven calculation. A language and application suitable for implementation with these options are then sought. Like the top-down design process, this process may be iterated many times if the selected applications cannot be suitably supported by the original design. ZMOB [243] and the Butterfly Multiprocessor [27] were designed using the bottom-up technique.

The designer may also compromise between the top-down and bottom-up approaches, resulting in a middle-out design approach [239]. The middle-out approach begins with the selection of an appropriate and well established knowledge-representation scheme that is suitable for the application. The representation scheme should already have a well developed programming environment that can be modified later according to the needs of a specific system. Primitives for concurrent execution may be added to allow users to annotate concurrent tasks in the application. The hardware and software architectures are then designed. When selecting features to incorporate, the designer considers structures previously designed and used. The middle-out approach can be further subdivided into top-first and bottom-first approaches. In a top-first

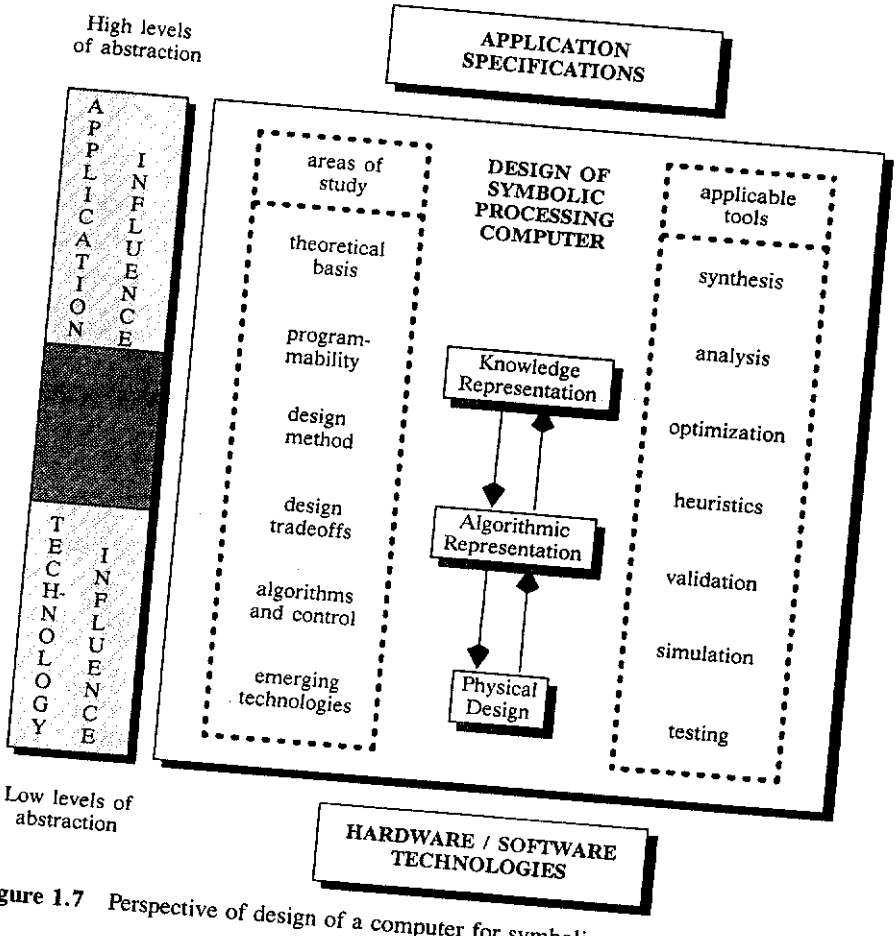

Figure 1.7 Perspective of design of a computer for symbolic processing applications.

middle-out approach, the designers start with a well defined knowledge representation scheme and tailor it to the given application. The architectures of ALICE [45] and FAIM-1 [12] were designed according to this philosophy. In a bottom-first middle-out approach, the designers first develop the architecture to support a well established representation scheme, then map the application to the chosen scheme. DADO [212] was developed using this approach.

The process of designing a computer for symbolic processing is iterative (see Figure 1.7). In mapping from application specifications to hardware and software technologies, the designers repeatedly propose and select knowledge representation schemes, algorithms, and physical design until a feasible and cost-effective mapping is found. Applicable design tools include methods for synthesis, analysis, optimization, heuristic design, validation, simulation, and testing. During the design process, the designers must address issues related

to theory, programmability, design methodology, design tradeoffs, effective control, and emerging technologies.

4.1 Single-Processor Symbolic Computers

Lisp has enjoyed the longest tenure in the mainstream of languages for symbolic processing, and has led to the greatest number of computers devoted to its execution. The earliest Lisp computers were PDP-6s; they were followed by the PDP-10s and PDP-20s of Digital Equipment Corporation [152]. The half-word instructions and stack operations of these computers were particularly well suited to Lisp. A great deal of work has also been done to improve garbage collection on the PDP-10s and PDP-20s.

The MIT AI Laboratory introduced a Lisp computer called CONS in 1976 [124], to be followed in 1978 by CADR, their second generation Lisp computer. This computer was the basis for many commercial Lisp computers—the Symbolics LM2, the Xerox 1100 Interlisp computer, and the Lisp Machines Incorporated Series III CADR, all introduced in 1981. Table 1.14 shows some notable and more recent commercial and research-oriented Lisp computers.

In addition to special hardware for improved efficiency, commercial Lisp computers also provide an integrated software development environment, such as KEE or ART, that allows programmers to develop, debug, and maintain large Lisp programs. A recent trend is to implement these development environments efficiently in software (rather than in microcode) on high-speed general purpose workstations. It is likely that in the future special purpose Lisp computers will be used to execute Lisp programs rather than to develop them.

The design of special purpose Lisp computers continues to be a popular research area. Many experimental computer designs have been reported [55, 83–85, 168, 182, 193–195, 222]. A single chip design to support Scheme, a dialect of Lisp, has been demonstrated in the Scheme-79 and Scheme-81 chips [219]. Scheme-79 was limited by its implementation of a register file and slow programmable array logic, but Scheme-81, a redesign, has proved to be much faster.

The popularity of rule-based systems arose at a time when parallel processing was highly popular. As a result, most architectures designed for supporting ruled-based languages have been parallel computers. However, the study of single-processor rule-based systems is still important because a parallel system is limited by the speed of its inferences, and the building blocks of a parallel system are likely to be single-processor symbolic computers.

The three most notable designs for executing logic languages are SRI's pipelined Prolog processor [226], and the Personal Sequential Inference (PSI) Machine and the Cooperative High-Speed Inference Machine (CHI), both developed by the Japanese FGCS Project [223, 230]. PSI, an integrated workstation with an execution speed of 30 KLIPS, was intended to be a software development tool for the project; it has since been redesigned into PSI-II, which

TABLE 1.14 Notable Sequential Symbolic Processing Machines.

Machine	Year/ Status[a]	Primary Language	Features
Scheme-79, Scheme-81	1981/PO	Scheme	Single hip; tail recursion Lexical scoping
Lisp Machines Lambda	1983/CA	Zetalisp, LMLisp	NuBus–multiprocessor capability Stack orientation
Symbolics 3600	1983/CA	Zetalisp, Flavors	Tagged memory Stack buffer Hardware garbage collection Single-address instruction
Tektronix 4400	1984/CA	Smalltalk, Franz Lisp	Lower-end AI workstation Motorola 68010/20
TI Explorer	1984/CA	Common Lisp	NuBus; tagged memory Microprogrammed Mega-chip version contains 60% of processor in 1 chip with 550,000 transistors [147]
Fujitsu ALPHA	1983/CA	Utilisp	Value cache; hardware stack Virtual stack
FGCS PSI-I, PSI-II	1985/CA	KL0 (Logic)	PSI-II; Cmos-GaAs; TTL 200 ns cycle time; cache; stacks Copying for structure data Hardware unification Tagged data Interpretive execution 150 KLIPS average speed
FGCS CHI-I, CHI-II	1986/PO	Current Mode Logic (CML)	CHI-II; CMOS-GaAs; TTL 170 ns Cycle time; Cache About 400 KLIPS average speed for append
SRI's Pipelined Prolog Processor	1984/SI	Prolog	Pipelined execution Instruction execution units Microprogrammed controller Interleaved memory FCFS module queues
SOAR	1984/PO	Smalltalk-80	RISC Expensive procedure calls in Smalltalk-80 Tagged and untagged instructions Large number of registers Automatic storage reclamation Direct object addressing Fast type checking

[a] Status CA: Commercially available PO: Prototype operational
HS: Hardware simulated UC: Under construction
SI: Simulations completed PD: Paper design

has an average performance of 150 KLIPS [230]. CHI was designed with speed in mind, using a less constrained technique. Like PSI the original CHI was redesigned, becoming CHI-II, with an estimated improvement in performance from 280 to 400 KLIPS for the *append* operation [230].

Single-processor support for production systems has focused on additional data memories [132] and RISC architectures [75].

Object-oriented languages have been implemented on sequential processors. The computers of the Xerox 1100 family were among the first workstations on which Smalltalk-80 was installed. Smalltalk-80 has also been implemented on a single chip using the RISC approach in the Smalltalk-On-A-RISC (SOAR) project [234]. SOAR has no microcode or fine-grained addressing hardware, and few multicycle instructions.

4.2 Parallel Symbolic Processors

In this section, we classify parallel symbolic processors in terms of their representations or programming techniques. Section 4.2.1 discusses the methods of communication and synchronization used in parallel symbolic processors. The rest of this section is devoted to discussions of the use of parallel symbolic processors for functional, rule-based, and object-oriented representations.

4.2.1 Communication and Synchronization

Communication. Message passing, marker passing, and value passing are the three predominant communication methods used by parallel symbolic processors.

Message passing is the conventional method of communication: information is formulated into a message and sent over the interconnection network. The computing elements are generally complex, and communication costs are high. Nonetheless, message passing is popular and has been used in many parallel symbolic processors.

Marker passing is the transfer of single-bit markers from one processor to another. A marker indicates the presence of a given property, and a set of markers indicates the conjunction of a set of properties. Each processor is simple and can store a few distinct marker bits. There is never any conflict: if two markers arrive at the same destination, they are simply ORed together. The basic inference operation is, therefore, set intersection. Marker passing is especially suitable for implementing semantic networks and for recognition problems in hardware. One such system is NETL [64, 66]. The Connection Machine was originally designed as a marker-passing system, but was later modified to use more powerful processing and communication methods and to support virtual processors [68].

The third method of communicating information is value passing, in which information is passed as continuous quantities. Only simple operations are performed on the numbers. The salient feature of this approach is that if several values arrive simultaneously at a single point, they are combined into a single value by a mathematical function, eliminating conflict in information transfer.

Examples of value-passing systems include the Boltzmann machine [66] and other neural computation systems [109]. Iterative relaxation techniques for problems such as low-level vision, speech understanding, and optimization all seem to be suited to value-passing architectures.

Synchronization. Synchronization is the control of concurrent access to shared items in a parallel processing system. It is important in message-passing systems, because such systems do not prevent competition for shared resources. Synchronization is not critical in marker- or value-passing systems because their predefined methods of passing markers and combining values prevent conflict.

Synchronization is important when data items are shared. In a program written in a procedural language, the order of statements dictates the order of execution; if two statements share a variable, the first must be executed before the second. Hence, when data sharing is necessary, synchronization control is implicitly defined by the order of statements. In contrast, when a program is written in a pure declarative language, the order of execution is not defined; when two statements share a variable, the order of execution is indeterminate. Thus explicit specification of synchronization control is needed when data is shared. Unfortunately, synchronization control cannot be specified explicitly in most declarative languages.

Synchronization can be carried out by shared memory or message passing. Shared memory is popular and has been used in systems such as Aquarius [54], the Concurrent Lisp Machine [217], the Concert MultiLisp Multiprocessor [92], and the Parallel Inference Engine [82]. Blackboard architectures and shared variables are two techniques that can be used for shared-memory synchronization.

The blackboard model was originally developed for abstracting features of the HEARSAY-II speech understanding system [61, 176]. There are three components: a set of knowledge sources, a blackboard, and control. The knowledge for processing the application is partitioned into separate knowledge sources. The data, including input/output and partial solutions, are stored in the blackboard. The blackboard may be partitioned into smaller blackboards, forming a hierarchy of solution spaces. To reach a solution, the knowledge sources manipulate the data in the blackboard, which is the only communication mechanism between them. A monitor is present to ensure that only one knowledge source is changing the blackboard at any time.

A more powerful blackboard architecture has been proposed in which control information (or metaknowledge) is allocated a separate blackboard [95]. This approach is more flexible and suits the nondeterministic nature of symbolic processing.

Synchronization may also be achieved through shared variables. Lisp languages that have been modified for parallel processing often contain shared variables for synchronization. Multilisp provides a mechanism that allows a process to wait for needed values to be generated; as in other languages, procedure activations may not be well nested, and a process can terminate prior to an

activation that it began. This problem must be addressed by the programming system [91].

Single assignment languages such as pure Prolog and pure data flow languages do not require careful synchronization of shared variables since a variable may be written to only once [143]. Prolog delays process reduction until enough information is available to make an effective decision. In Guarded Horn Clauses (GHC) [231], the current kernel language of the Japanese FGCS project, OR-parallelism is not exploited, and strict synchronization rule suspends a subgoal if it tries to modify its parent environment. This simplifies implementation, but the resulting language is less expressive [221].

Synchronization in a message-passing system is accomplished through a protocol implemented in hardware or software. In a standard message-passing environment—in an object-oriented programming system, for example—messages may be of arbitrary complexity. Actor is a paradigm for systems with message passing of this nature [98, 100]. When an actor receives a message, it performs predefined primitive actions; in this sense, actor systems are inherently parallel. The Apiary architecture is based on actors [99]. Other message-passing systems for symbolic applications include the Contract Net system [207] and the Rand Distributed Air Traffic Control System [33].

4.2.2 Parallel Functional Programming Computers. The majority of special purpose parallel processors designed to support functional languages are oriented toward Lisp: for example Concert [92], EM-3 [248], and a multimicroprocessor Concurrent Lisp system developed at Kyoto University [217]. All these systems require users to specify, to some extent, the tasks to be decomposed. Compilers for automatic detection of parallelism in sequential Lisp programs are an area of active research. Table 1.15 presents some of the more highly publicized parallel systems for functional programming.

The majority of computers designed for general purpose applications have only a few features specifically appropriate to symbolic processing. In these computers the inference engine and knowledge base are not separated and are almost exclusively implemented by sophisticated software structures. Lisp is added as one of the several languages to accompany a general purpose parallel computer. A Lisp compiler is used for decomposing tasks for parallel processing, and users must perform some task annotation. Examples of commercial multiprocessors include the Butterfly [41], the Connection Machine [39], and the Intel iPSC concurrent computer [22].

The Connection Machine has special chips containing 16-bit serial processors and router circuits. *Lisp in CM-2 allows users to specify a parallel variable (*pvar*), a first-class object with a value for each processor in the computer [40]. The primitive *pvar* can be accessed concurrently (with possible masks) by all local or remote processors in SIMD or multiple SIMD mode. CM-Lisp is a dialect of Common Lisp extended to allow fine-grained, data-oriented parallel processing; it provides higher-level data abstractions called *zappings*, which are similar in structure to arrays or hash tables. Broadcasts, reductions, and combinations can be specified.

TABLE 1.15 Notable Parallel Functional Programming Computers.

Machine	Year/ Status[a]	Primary Language	Intercon- nection	Commu- nication	Features
Butterfly	1985/CA	Multilisp	Butterfly switch	Shared memory	256 MC68000-series PEs Homogeneous, tightly coupled General purpose multiprocessor
iPSC	1986/CA	Common Lisp	Hypercube	Message passing	256 Intel 80286 and 80386 processors No shared environment User decomposes program into concurrent processes that communicate by messages General purpose multiprocessor
Connection Machine	1986/CA	*Lisp CM-Lisp	Hypercube	Message passing	Model CM-2 has 4096 bit serial processors Users annotate Lisp programs from SIMD or multiple SIMD parallel processing C*, Fortran, and Paris (CM-2 assembly language) are also supported
Concurrent Lisp Machine	1983/PO	C-Lisp	Multiple buses	Shared memory	17 MC68000-series PEs Special cell interface Control stack Garbage collector
EM-3	1984/PO	EMLISP	Modified delta network	Message passing	Listlike data-driven language 16 MC68000-series PEs Special router chip Control for function evaluation
Concert	1986/PO	Multilisp	Ringbus	Shared memory	32-64 MC68000-series PEs Network is segmented bus in shape of ring
Rediflow	1984/SI	Functional Equation Language (FEL)	Mesh or richer connections	Message passing	Demand/data-driven Loosely coupled Hardware support for load balancing Distributed garbage collection
Alice	–/UC	Hope Lisp, , Prolog	Cluster of processors, ring buffer	Message passing	Transputer as basic processor Reference counter for garbage collection

[a] Refer to Table 1.14 for explanation of status codes.

Other computers with limited support for parallel symbolic processing, such as ZMOB, are being developed at universities [243]. These computers are not just symbolic processing computers, but also general purpose computers, appropriate for both numeric and symbolic computations.

4.2.3 Parallel Logic Architectures. This section presents parallel systems suitable for evaluating logic programs. Table 1.16 summarizes notable projects.

Unification and search are two key features by which to evaluate logic programs. Architectures that emphasize efficient search of logic programs include the BAGOF architecture [34] and MANIP-2 [138]. The MANIP-2 architecture is particularly interesting because of its emphasis on heuristic parallel search strategies.

Two significant parallel logic systems have been developed at universities. The Aquarius multiprocessor, developed at the University of California, Berkeley, couples intensive numeric calculations with symbolic manipulations [54]. Its designers used parallelism at all levels of computation and considered cost to be secondary to performance considerations. The Parallel Inference Engine (PIE), under development at the University of Tokyo, has a target of 1,000 processors, and a speed-up of 170 has already been estimated for 256 processors [163]. PIE utilizes only OR-parallelism.

Probably the most massive effort in the development of parallel logic systems has been made by the Japanese Fifth Generation Computing System project (FGCS). The project distinguishes three major development areas: problem solving and inference machines (hardware), knowledge-base management systems (software and algorithms), and an intelligent man-machine interface [162]. The project is divided into three stages. The initial stage, now complete, explored basic computer technology and processing techniques. The middle stage is for the development and construction of experimental subsystems. The final stage is devoted to the development of the complete system.

The initial-stage designs of the Parallel Inference Machines (PIM) were based on two concepts: reduction and data flow [115, 166]. The architectures for the two computers (PIM-R and PIM-D, respectively) were similar, but the evaluation techniques reflected two different philosophies. The hardware of PIM-R and PIM-D was simulated.

PIM-I is a hardware design for the intermediate stage of the FGCS project. The target speed for the 100-processor PIM-I is 10–20 MLIPS, with a target speed of 200–500 KLIPS for the individual processors [230]. The machine language for this computer will be KL1-B, which is based on Guarded Horn Clauses. The software will be developed on a network of PSI systems (multi-PSI) [113].

4.2.4 Parallel Systems for Production System Computations. The exploration of computers for production systems has been carried out primarily at universities. Table 1.17 presents a summary of these projects.

TABLE 1.16 Notable Parallel Computers for Logic Representations.

Machine	Year/ Status[a]	Interconnection	Communication	Features
BAGOF	1984/PD	Bus	Shared memory	OR-parallelism Separate static and dynamic memory Token pool
MANIP-2	1985/PD	Global broadcast bus	Message passing	Cluster of PEs with local memory Distributed selection Heuristic guiding and pruning
Aquarius	–/UC	Bus and crossbar	Shared memory	Heterogeneous MIMD 16 PEs Synchronization through Goodman Cache Crossbar to shared memory modules Special Prolog, floating point, and I/O processors
Parallel Inference Engine (PIE)	1984/SI	Switching network	Shared memory	100s to 1000s of inference units Goal rewriting model OR-parallelism Sequential AND processing Activity controllers to control inference tree Unify processors connected to definition memory containing program
Parallel Inference Machine-Reduction (PIM-R)	1986/HS	Multistage network	Shared memory	Many inference modules connected to structure memory units through network Structure copying
Parallel Inference Machine Dataflow (PID-M)	1986/HS	Multistage network	Shared memory	Multiple PEs connected to structure memory Unfolding interpreter Asynchronous communication Streams for nondeterministic control
PIM-1	–/UC	Hierarchy	Shared memory, message passing	100 PEs 8-PE clusters interconnected with shared memory and parallel cache

[a] Refer to Table 1.14 for explanation of status codes.

TABLE 1.17 Notable Parallel Machines for Production Systems.

Machine	Year/ Status[a]	Interconnection	Communication	Features
DADO1 DADO2	1986/PO	Binary tree	Message passing	DADO2; 1023 8-bit processors; 16K of user memory Two modes: MIMD and multiple SIMD Special I/O circuits
Non-Von	1985/PO	Binary tree with leaf connections Connections to LPEs	Message passing	Binary SIMD tree of small PEs Leaves are connected in mesh Large PEs connected by network, with connections to high-level nodes in tree Intelligent disk drives connected to LPEs
PSM	1986/SI	Shared buses	Shared memory	32–64 processors Parallel Rete Match algorithm PEs connected to memory modules through cache Local memory Hardware task scheduler

[a] Refer to Table 1.14 for explanation of status codes.

The DADO1 and DADO2 projects [211, 213] at Columbia University are developing a class of computers based on tree architectures. The upper-level nodes of the tree synchronize and select rules, intermediate nodes match and store rules, and the leaves act as working memory.

Another project in progress at Columbia University is the Non-Von computer—Non-Von-1 was an early version [104, 112]. Unlike DADO2, Non-Von connects smaller processing elements, which are subject to the control of large processing elements into a binary tree. Most of the pattern matching tasks that are done in the working memory have small granularity and are more suitable to be executed on a large number of small processing elements.

Finally, the PSM computer is a large-grain machine that is specifically designed to support the OPS5 system and a parallel Rete Match algorithm. Simulations have shown promising speed-ups, and that 32 processors are sufficient to exploit most of the parallelism in this system [86].

Numerous studies have been made of strategies for mapping production systems to multiprocessors [132, 177, 225, 232].

4.2.5 Parallel Object-Oriented Architectures. Most development work in object-oriented programming has been done on computers such as the Intel iPSC that are not specifically designed for object-oriented computation. Two notable multiprocessors, however, are designed specifically for object-oriented computations. FAIM-1 [12] is a multiprocessor with special RISC processors

TABLE 1.18 Notable Parallel Computers for Object Oriented Computation.

Machine	Year/ Status[a]	Interconnection	Communication	Features
FAIM-1	–/UC	Hexagonal mesh	Message passing	PEs = Hectagons (hexagonal configuration) Heterogeneous shared memory multiprocessor Instruction stream memory, post office communication processor, evaluation processor, and others Three-port switch at edge of array for I/O and wrapping of connections Up to 10 32-bit workstation/processors
Dragon	–/UC	Bus	Shared memory	Tightly coupled Associative cache at each processor
Apiary	1980/PD	Single-stage network	Message passing	Implements Actor model Computations in the Actor model are partial orders of events with no assignment commands

[a] Refer to Table 1.14 for explanation of status codes.

connected by a hexagonal mesh. At first an intermediate language, OIL, was developed for interfacing between its modules. However, the scope of the project has recently been changed to exclude the development of OIL and program the computer in MultiScheme. The second specialized multiprocessor was developed by the Dragon project; it supports only 10 processors [160]. Table 1.18 summarizes notable projects.

4.3 Connectionist Processing

Many connectionist implementations focus on correlations between nodes in a graph and have been designed primarily for semantic networks. Artificial neural networks are also of interest, but unfortunately, current technology for their design has precluded their development in a role greater than that of subsystem.

The four connectionist designs for the implementation of semantic networks correspond to the three types of message-passing environments. NETL uses the most elementary processing elements in a marker-passing system [65, 66]. THISTLE is a similar design, but employs value passing instead of marker passing [66]. The Connection Machine can be programmed to simulate marker

TABLE 1.19 Parallel Connectionist Systems.

Machine	Year/ Status[a]	Interconnection	Communication	Features
NETL	1979/PD	Multilevel switching network	Marker passing	For semantic networks A million processors, each can store 16 markers Simple PEs Only Boolean functions
THISTLE	1983/PD	Multilevel switching	Value passing	For semantic networks Similar to NETL, but with 8-bit value passing
Connection Machine	1986/CA	Hypercube	Message passing	General purpose SIMD and multiple SIMD processing Can be programmed for marker passing operations in semantic networks and simulating neural networks
SNAP	1985/PD	Mesh with global bus	Message passing	Square array of identical processors CACM in each PE for relationships between nodes Communication unit Processing unit

[a] Refer to Table 1.14 for explanation of status codes.

passing in semantic networks and value passing in artificial neural networks by using massive data-level parallelism [39]. Finally, SNAP relies on message passing [159]. The details of these machines are summarized in Table 1.19.

4.4 Summary

The purpose of this section was to give a high-level perspective of existing special purpose computers designed for symbolic processing. Complete systems can be classified by two attributes. The first attribute is the fundamental approach to processing taken by the system: sequential, parallel, or connectionist.

The second attribute is the overall processing technique. For current systems the processing technique can be defined in one of two ways. The first way is by knowledge representation: for example, computers for parallel logic computation. The second definition is based on the programming paradigm: for example, functional or object-oriented.

Systems can be further classified by various design decisions, and by the micro/macro-level architectures employed. From a high-level perspective,

however, it is not necessary to do this in order to understand the state of the art in symbolic processing systems.

Figure 1.8 classifies the computers presented in this section on the basis of the two suggested attributes. However, a number of systems have been designed for diverse symbolic applications and cannot be classified into a unique category. The Connection Machine is a notable example of this class.

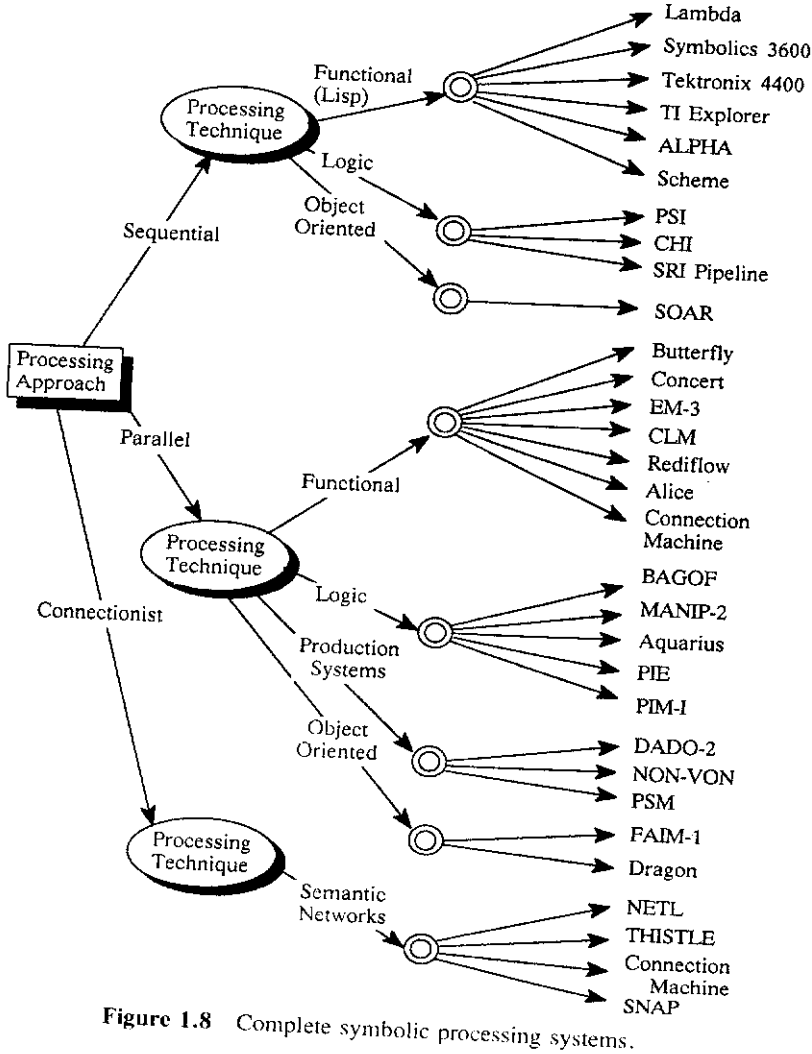

Figure 1.8 Complete symbolic processing systems.

5 RESEARCH DIRECTIONS

We have presented in this paper an extensive discussion and analysis of the state of the art in computer solutions of symbolic processing problems. We conclude this section by indicating some of the research areas where advances will most likely bring about fast and efficient computer solution of these problems.

Technologies. The design of a system is often driven by its cost; hence, while the fastest technologies are preferable, their selection is subject to cost constraints. New and emerging technologies may give higher performance but are often prohibitively expensive. The candidate technologies that will likely become cost effective in the near future include GaAs circuits [130, 131], Wafer-Scale Integrated circuits (WSI) [154], analog–digital VLSI circuits, and optical computing techniques [21, 79, 244].

These emerging technologies offer tremendous potential for increasing the processing speeds of current computers; the extension of processing power is especially valuable for real-time systems. However, the most that can be expected from these technologies is an improvement in speed of about one to two orders of magnitude in the next ten years. They will not greatly impact the size or type of symbolic applications that are addressed today, since many of these applications involve huge search spaces and an increase in computational speed by one to two orders of magnitude will do little to increase the size of a solvable instance of such a problem [238].

Algorithms. Research in the area of application-specific algorithms has the greatest potential for speeding applications. New and improved algorithms may be developed by finding alternative ways to incorporate knowledge about the application domain into the computer solution. In this way, advancement of symbolic processing capabilities in the area of application-specific algorithms is tightly linked to advancement in the area of knowledge representations.

Knowledge Representations. Most new knowledge representations for symbolic processing have emphasized declarative and distributed features, reducing programming complexity. These representation schemes may have to be modified or extended to fit the applications and the computational environment. The addition of temporal features and nonmonotonicity would also be helpful.

A major problem in the area of knowledge representations is the lack of technique to guide the evaluation and selection of a knowledge representation scheme. Research in this area could prove extremely valuable. Learning techniques for incorporating new knowledge about application domains into current knowledge intensive applications may also have a great impact on symbolic processing. Artificial neural networks and connectionist representations already incorporate automated learning techniques into their design at the knowledge representation level.

Software Architecture. Software architectures are highly dependent on research in the area of knowledge representation. The generation of new software environments, tools, and languages will probably rely on amalgamation of known knowledge representation techniques. Software development systems and automated intelligent programming assistants are prime areas for the advancement of symbolic programming. The problems of program verification and validation and continuous maintenance of symbolic programs are important related topics.

Hardware Architectures. As with software, hardware architectures are often based on known design techniques such as parallel processing and pipelining. Architectural innovation may be encouraged, however, by the availability of new and emerging technologies.

New hardware architectures are best utilized for operations that the computer performs frequently. Counter to intuition, identification of these tasks is very difficult. Operations may be instructions, parts of instructions, groups of instructions, or frequently recurring tasks. Identification of new and valid areas for development of new hardware architectures is an important area of research.

System Design. System-level design is often based on an overall design philosophy; for example, systems may contain a mix of data- and control-flow computation. The proper mix of control, data, and demand flow is one area of research that may impact systems for symbolic processing. New systems for symbolic processing may also greatly benefit from the integration of new hardware subsystems and microlevel architectures, and of new and emerging technologies. The major difficulty lies in integrating designs with radically different knowledge representations. The combination of distributed representation (offered by artificial neural networks) and procedural or declarative representation (offered by standard computers) is an interesting area for development.

6 ACKNOWLEDGMENTS

This research was partially supported by National Aeronautics and Space Administration Grant NCC 2-481 and National Science Foundation Grant MIP 88-10584. The research of M. B. Lowrie was supported by a Ph. D Scholarship from AT&T Bell Laboratories.

REFERENCES

1. "ESPRIT: Europe Challenges U.S. and Japanese Competitors," *Future Generation Computer Systems*, vol. 1, no. 1, pp. 61–69, North-Holland, New York, 1984.
2. "Special Issue on Natural Language Processing," *Proc. IEEE*, July 1986.

3. "Special Issue on Computer Vision," *Proc. IEEE*, 1987.
4. Defense Advanced Research Project Agency, *DARPA Neural Network Study*, Lincoln Laboratory, Massachusetts Institute of Technology, Lexington, July 1988.
5. G. Agha and C. Hewitt, "Concurrent Programming Using Actors: Exploiting Large-Scale Parallelism," *Lecture Notes in Computer Science*, no. 206, pp. 19–41, Springer-Verlag, New York, December 1985.
6. G. Agha, *Actor: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, 1986.
7. L. Aiello, C. Cecchi, and D. Sartini, "Representation and Use of Metaknowledge," *Proc. IEEE*, pp. 1304–1321, October 1986.
8. J. Alspector and R. B. Allen, "A Neuromorphic VLSI Learning System," *Advanced Research in VLSI: Proc. 1987 Stanford Conference*, ed. P. Loseleben, MIT Press, Cambridge, 1987.
9. M. Amamiya et. al., "New Architecture for Knowledge Based Mechanisms," *Proc. Int'l Conf. on Fifth Generation Computer Systems*, pp. 179–188, Japan, 1981.
10. M. Amamiya et. al., "Implementation and Evaluation of List-Processing-Oriented Data Flow Machine," *Proc. 13th Int'l Symp. on Computer Architecture*, pp. 10–19, IEEE/ACM, 1986.
11. C. W. Anderson, "Strategy Learning with Multilayer Connectionist Representations," *Proc. 4th Int'l Workshop on Machine Learning*, pp. 103–114, June, 1987.
12. J. M. Anderson, W. S. Coates, A. L. Davis, R. W. Hon, I. N. Robinson, S. V. Robison, and K. S. Stevens, "The Architecture of FAIM-1," *Computer*, vol. 20, no. 1, pp. 55–65, IEEE, January 1987.
13. H. L. Andrews, "Speech Processing," *Computer*, vol. 17, no. 10, pp. 315–324, IEEE, October 1984.
14. Arvind, K. Gostelow and W. Plouffe, *An Asynchronous Programming Language and Computing Machine*, Techical Report 114a, University of California, Irvine, December 1978.
15. E. Babb, "Implementing a Relational Database by Means of Specialized Hardware," *Trans. on Database Systems*, vol. 4, no. 1, pp. 1–29, ACM, March 1979.
16. J. Backus, "Can Programming be Liberated From the von Neumann Style? A Functional Style and Algebra of Programs," *Comm. of the ACM*, vol. 21, no. 8, pp. 613–641, 1978.
17. H. G. Baker, Jr., "Optimizing Allocation and Garbage Collection of Spaces," *Artificial Intelligence: An MIT Perspective*, ed. P. H. Winston and R. H. Brown, vol. 1, pp. 391–396, MIT Press, 1979.
18. A. Barr and E. A. Feigenbaum, *The Handbook of Artificial Intelligence*, vol. 1, 2, and 3, William Kaufmann, Los Altos, CA, 1981, 1982.
19. J. M. Barth, "Shifting Garbage Collection Overhead to Compile Time," *Comm. of the ACM*, vol. 20, no. 7, pp. 513–518, July 1977.
20. E. B. Baum, "Towards Practical 'Neural' Computation for Combinatorial Optimization Problems," *AIP Conf. Proc. on Neural Networks for Computing*, pp. 53–58, 1986.
21. P. B. Berra and N. B. Troullinos, Optical Techniques and Data/Knowledge Base Machines," *Computer*, vol. 20, no. 10, pp. 59–70, IEEE, October 1987.
22. D. Billstrom, J. Brandenburg, and J. Teeter, "CCLISP on the iPSC Concurrent Computer," *Proc. 6th Int'l Conf. on Artificial Intelligence*, pp. 7–12, AAAI, Seattle, WA, 1987.

23. D. G. Bobrow, et al., *CommonLoops: Merging Common Lisp and Object-Oriented Programming*, Technical Report ISL-85-8, Xerox Palo Alto Research Center, August 1985.

24. D. G. Bobrow and P. J. Hayes, (ed.), "Special Issue on Nonmonotonic Logic," *Artificial Intelligence*, vol. 13, no. 1 & 2, North-Holland, New York, April 1980.

25. K. Bowen, "Meta-Level Programming and Knowledge Representation," *New Generation Computing*, vol. 3, no. 4, pp. 359–383, 1985.

26. R. Brooks and R. Lum, "Yes, An SIMD Machine Can be Used for AI," *Proc. Int'l Joint Conf. on Artificial Intelligence*, pp. 73–79, 1985.

27. C. M. Brown, C. S. Ellis, J. A. Feldman, T. J. LeBlank, and G. L. Peterson, Research with the Butterfly Multicomputer," *Rochester Research Review*, pp. 3–23, 1984.

28. L. Brownston, R. Farrell, E. Kant, and N. Martin, *Programming Expert Systems in OPS5*, Addison-Wesley, Reading, MA, 1985.

29. M. Bruynooghe, "The Memory Management of PROLOG Implementations," in *Logic Programming*, ed. K. Clark and S. A. Tarnlund, pp. 83–89, Academic Press, New York, 1982.

30. B. G. Buchanan and E. H. Shortliffe, *Rule-Based Experts Programs: The MYCIN Experiments of the Stanford Heuristic Programming Project*, Addison-Wesley, Reading, MA, 1984.

31. A. W. Burks, "Keynote of CONPAR86," *Lecture Notes in Computer Science No. 237*, pp. 1–17, Springer-Verlag, New York, 1986.

32. R. M. Burstall, D. B. MacQueen, and D. T. Sannella, "HOPE: An Experimental Applicative Language," *Conf. Record of Lisp Conf.*, pp. 136–143, Stanford Univ., Menlo Park, CA, 1980.

33. S. Cammarata et al., "Strategies of Cooperation in Distributed Problem Solving," *Proc. Int'l Joint Conf. on Artificial Intelligence*, pp. 767–770, IJCAI, Inc., 1983.

34. A. Ciepielewski and S. Haridi, "Execution of Bagof on the OR-Parallel Token Machine," *Proc. Int'l Conf. Fifth Generation Computer Systems*, pp. 551–560, ICOT and North-Holland, New York, 1984.

35. K. Clark and S. Gregory, "Note on System Programming in PARLOG," *Proc. Int'l Conf. Fifth Generation Computer Systems*, pp. 299–306, ICOT and North-Holland, New York, 1984.

36. K. L. Clark and S. A. Tarnlund, (eds.), *Logic Programming*, Academic Press, New York, 1982.

37. W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.

38. P. R. Cohen and M. R. Grinberg, "A Theory of Heuristic Reasoning About Uncertainty," *AI Magazine*, pp. 17–24, AAAI, Summer 1983.

39. Thinking Machines Corporation, *Connection Machine Model CM-2 Technical Summary*, Technical Report HA87-4, Cambridge, MA, April 1987.

40. Thinking Machines Corporation, *Lisp Reference Manual*, Version 4.0, Cambridge, MA, October 1987.

41. A. S. Cromarty, What Are Current Expert System Tools Missing?" *Proc. COMPCON Spring*, pp. 411–418, IEEE, 1985.

42. O. J. Dahl and K. Nygaard, "SYMULA—An Algol-Based Simulation Language," *Communications of the ACM*, vol. 9, no. 9, pp. 671–8, September 1966.

43. J. Darlington and M. Reeve, "ALICE: A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages," *Proc. Conf. on Functional Programming Languages and Computer Architecture*, pp. 65–74, ACM, 1981.
44. J. Darlington, P. Henderson, and D. Turner, *Functional Programming and Its Applications*, Cambridge University Press, Cambridge, UK, 1982.
45. J. Darlington and M. Reeve, *ALICE and the Parallel Evaluation of Logic Programs*, Preliminary Draft, Department of Computing, Imperial College of Science and Technology, London, June 1983.
46. J. Darlington, "Functional Programming," *Distributed Computing*, eds. F. B. Chambers, D. A. Duce, and G. P. Jones, Academic Press, London, 1984.
47. J. Darlington, A. J. Field, and H. Pull, *The Unification of Functional and Logic Languages*, Technical Report, Imperial College, London, February 1985.
48. A. L. Davis and S. V. Robison, "The FAIM-1 Symbolic Multiprocessing System," *Proc. COMPCON Spring*, pp. 370–375, IEEE, 1985.
49. R. Davis and B. Buchanan, "Meta-level Knowledge: Overview and Applications," *Proc. 5th Int'l Joint Conf. on Artificial Intelligence*, pp. 920–928, William Kaufmann, Los Altos, CA, 1977.
50. M. F. Deering, "Architectures for AI," *Byte*, pp. 193–206, McGraw-Hill, New York, April 1985.
51. D. DeGroot, "Restricted AND-Parallelism," *Proc. Int'l Conf. on Fifth Generation Computers*, pp. 471–478, ICOT and North-Holland, New York, November 1984.
52. D. DeGroot and G. Lindstrom (eds.), *Logic Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1985.
53. J. S. Denker, "Neural Network Models of Learning and Adaptation," *Physica*, pp. 216–232, 1986.
54. A. M. Despain and Y. N. Patt, "Aquarius—A High Performance Computing System for Symbolic/Numeric Applications," *Proc. COMPCON Spring*, pp. 376–382, IEEE, February 1985.
55. P. Deutsch, "Experience with a Microprogrammed Interlisp System," *Proc. MICRO*, vol. 11, ACM/IEEE, November 1978.
56. D. J. DeWitt, "DIRECT—A Multiprocessor Organization for Supporting Relational Database Management Systems," *Trans. on Computers*, pp. 395–406, IEEE, June 1979.
57. E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, "On-the-Fly Garbage Collection: An Exercise in Cooperation," *Comm. of the ACM*, vol. 21, no. 11, pp. 966–975, November 1978.
58. J. Doyle, "A Truth Maintenance System," *Artificial Intelligence*, vol. 12, no. 3, pp. 231–272, North-Holland, New York, 1979.
59. H. Dreyfus and S. Dreyfus, "Why Expert Systems Do Not Exhibit Expertise," *Expert*, vol. 1, no. 2, IEEE, Summer 1986.
60. R. O. Duda, P. E. Hart, and N. J. Nilsson, "Subjective Bayesian Methods for Rule-Based Inference Systems," *Proc. Nat'l Computer Conf.*, pp. 1075–1082, AFIPS Press, 1976.
61. L. D. Erman, F. Hayes-Roth, V. R. Lesser, and D. R. Reddy, "The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty," *Computing Surveys*, vol. 12, no. 2, pp. 213–253, ACM, June 1980.

62. F. Fagin, *How Far Can We Go With Proven Technology?* Special Session on Neural Networks, AAAI Conference, Seattle, WA, July 1987.

63. R. Fagin and J. Halpern, "Belief, Awareness, and Limited Reasoning: Preliminary Report," *Proc. Int'l Joint Conf. on Artificial Intelligence*, pp. 491–501, IJCAI Inc., 1985.

64. S. E. Fahlman, *NETL: A System for Representing and Using Real-World Knowledge*, MIT Press, Cambridge, 1979.

65. S. E. Fahlman, "Design Sketch for a Million-Element NETL Machine," *Proc. 1st Annual Nat'l Conf. on Artificial Intelligence*, pp. 249–252, AAAI, August 1980.

66. S. E. Fahlman and G. E. Hinton, "Massively Parallel Architectures for AI: NETL, THISTLE, and Boltzmann Machines," *Proc. Nat'l Conf. on Artificial Intelligence*, pp. 109–113, AAAI, 1983.

67. S. E. Fahlman, "Parallel Processing in Artificial Intelligence," *Parallel Computing*, vol. 2, pp. 283–286, 1985.

68. S. E. Fahlman and G. E. Hinton, "Connectionist Architecture for Artificial Intelligence," *Computer*, vol. 20, no. 1, pp. 100–109, IEEE, January 1987.

69. C. Faloutos, "Access Method for Text," *Computing Surveys*, ACM, March 1985.

70. R. D. Fennell and V. R. Lesser, "Parallelism in Artificial Intelligence Problem Solving: A Case Study of Hearsay-II," *Trans. on Computers*, vol. C-26, no. 2, pp. 98–111, IEEE, February 1977.

71. E. A. Feustel, "On the Advantages of Tagged Architecture," *Trans. on Computers*, vol. C-22, no. 7, pp. 644–656, IEEE, 1973.

72. R. E. Fikes and N. J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," *Artificial Intelligence*, vol. 2, no. 3 & 4, pp. 189–208, North-Holland, New York, 1971.

73. M. A. Fischetti, "A Review of Progress at MCC," *Spectrum*, IEEE, March 1986.

74. C. L. Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intelligence*, vol. 19, no. 1, pp. 17–37, North-Holland, New York, September 1982.

75. C. L. Forgy, A. Gupta, A. Newell, and R. Wedig, Initial Assessment of Architectures for Production Systems," *Proc. Nat'l Conf. on Artificial Intelligence*, pp. 116–120, AAAI, August 1984.

76. K. S. Fu, *Syntactic Methods in Pattern Recognition*, Academic Press, New York, 1974.

77. K. Fuchi, "The Direction the FGCS Project Will Take," *New Generation Computing*, vol. 1, no. 1, pp. 3–9, OHMSHA Ltd. and Springer-Verlag, New York, 1983.

78. J. L. Gaudiot, S. Lee, and A. Sohn, *Data-Driven Multiprocessor Implementation of the Rete Match Algorithm*, Technical Report, Department of Computer Science, University of Southern California, Los Angeles, 1987.

79. T. K. Gaylord and E. I. Verriest, "Matrix Triangularization Using Arrays of Integrated Optical Givens Rotation Devices," *Computer*, vol. 20, no. 12, pp. 59–67, IEEE, December 1987.

80. M. R. Genesereth, "An Overview of Meta-Level Architecture," *Proc. Nat'l Conf. on Artificial Intelligence*, pp. 119–124, AAAI, 1983.

81. A. J. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA, 1983.

82. A. Goto, H. Tanaka, and T. Moto-oka, "Highly Parallel Inference Engine PIE—Goal Rewriting Model and Machine Architecture," *New Generation Computing*, vol. 2, no. 1, pp. 37–58, OHMSHA Ltd. and Springer-Verlag, New York, 1984.
83. E. Goto, T. Ida, K. Hiraki, M. Suzuki, and N. Inada, "FLATS, A Machine for Numerical, Symbolic and Associative Computing," *Proc. 6th Int'l Joint Conf. on Artificial Intelligence*, pp. 1058–1066, William Kaufmann, Los Altos, CA, August 1979.
84. N. Greenfeld and A. Jericho, "A Professional's Personal Computer System," *Proc. 8th Int'l Symp. on Comp. Architecture*, pp. 217–226, IEEE/ACM, 1981.
85. M. Griss and M. Swanson, "MBALM/1700: A Microprogrammed Lisp Machine for the Burroughs B1726," *Proc. MICRO-10*, ACM/IEEE, 1977.
86. A. Gupta, C. L. Forgy, A. Newell, and R. Wedig, "Parallel Algorithms and Architectures for Rule-Based Systems," *Proc. 13th Int'l Symp. on Computer Architecture*, pp. 28–37, IEEE/ACM, 1986.
87. J. Halpern and Y. Moses, "A Guide to the Modal Logics of Knowledge and Belief: Preliminary Draft," *Proc. Int'l Joint Conf. on Artificial Intelligence*, pp. 480–490, IJCAI Inc., 1985.
88. R. H. Halstead, Jr., "Implementation of MULTILISP: LISP on a Multiprocessor," *Proc. Symp. on LISP and Functional Programming*, ACM, 1984.
89. R. Halstead, "Parallel Symbolic Computing," *Computer*, vol. 19, no. 8, pp. 35–43, IEEE, August 1986.
90. R. Halstead, Jr., "An Assessment of Multilisp: Lessons from Experience," *Int'l J. Parallel Programming*, vol. 15, no. 6, pp. 459–501, December 1986.
91. R. Halstead, Jr. and J. Loaiza, "Exception Handling in Multilisp," *Proc. Int'l Conf. on Parallel Processing*, pp. 822–830, August 1985.
92. R. Halstead, Jr., T. Anderson, R. Osborne, and T. Sterlig, "Concert: Design of a Multiprocessor Development System," *Proc. Int'l Symp. on Computer Architecture*, pp. 40–48, IEEE/ACM, June 1986.
93. P. B. Hawthorn and D. J. DeWitt, "Performance Analysis of Alternative Database Machine Architectures," *Trans. on Software Engineering*, vol. SE-8, no. 1, pp. 61–75, IEEE, January 1982.
94. H. Hayashi, A. Hattori, and H. Akimoto, ALPHA: "A High-Performance Lisp Machine Equipped with a New Stack Structure and Garbage Collection System," *Proc. 10th Annual Int'l Symp. on Computer Architecture*, pp. 342–348, IEEE/ACM, June 1983.
95. B. Hayes-Roth, "A Blackboard Architecture for Control," *Artificial Intelligence*, vol. 26, no. 3, pp. 251–321, North-Holland, New York, July 1985.
96. F. Hayes-Roth, D. A. Waterman, and D. B. Lenat, *Building Expert Systems*, Addison-Wesley, Reading, MA, 1983.
97. P. Henderson, *Function Programming, Application and Implementation*, Prentice-Hall, Englewood Cliffs, NJ, 1980.
98. C. Hewitt, "Viewing Control Structure as Patterns of Passing Messages," *Artificial Intelligence*, vol. 8, no. 3, pp. 323–364, North-Holland, New York, 1977.
99. C. Hewitt, "The Apiary Network Architecture for Knowledgeable Systems," *Conf. Record of Lisp Conf.*, pp. 107–117, Stanford University, Menlo Park, CA, 1980.
100. C. Hewitt and H. Lieberman, "Design Issues in Parallel Architectures for Artificial Intelligence," *Proc. COMPCON Spring*, pp. 418–423, IEEE, February 1984.
101. C. E. Hewitt, *Description and Theoretical Analysis of PLANNER: A Language for Proving Theorems and Manipulating Models in Robots*, MIT AI Lab. TR-258, 1972.

102. Y. Hibino, "A Practical Parallel Garbage Collection Algorithm and Its Implementations," *Proc. 7th Annual Symp. on Computer Architecture*, pp. 113–120, IEEE/ACM, May 1980.

103. W. D. Hillis, *The Connection Machine*, MIT Press, Cambridge, 1985.

104. B. K. Hillyer and D. E. Shaw, "Execution of OPS5 Production Systems on a Massively Parallel Machine," *J. Parallel and Distributed Computing*, vol. 3, no. 2, pp. 236–268, Academic Press, New York, 1986.

105. J. Hintikka, "Impossible Possible World Vindicated," *J. Philosophical Logic*, pp. 475–484, 1975.

106. G. Hinton, J. L. McClelland, and D. D. Rumelhart, "Distributed Representations," *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, eds. D. E. Rumelhart, J. L. McClelland, and the PDP Research Group, vol. 1, MIT Press, Cambridge, 1986.

107. G. Hinton and T. J. Sejnowski, "Learning and Relearning in Boltzmann Machines," *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, eds. D. E. Rumelhart, J. L. McClelland, and the PDP Research Group, vol. 1, MIT Press, Cambridge, 1986.

108. G. E. Hinton, *Connectionist Learning Procedures*, Technical Report CMU-CS-87-115, Carnegie Mellon University, Pittsburgh, PA, June 1987.

109. J. J. Hopfield and D. W. Tank, "Neural Computation of Decisions in Optimization Problems," *Biological Cybernetics*, vol. 52, no. 3, pp. 1–25, July 1985.

110. J. J. Hopfield and D. W. Tank, *Disordered Systems and Biological Organization*, Springer-Verlag, New York, 1986.

111. J. J. Hopfield and D. W. Tank, "Computing with Neural Circuits: A Model," *Science*, pp. 625–633, August 1986.

112. H. A. H. Ibrahim, J. R. Kender, and D. E. Shaw, "Low-Level Image Analysis Tasks on Fine-Grained Tree-Structured SIMD Machines," *J. Parallel and Distributed Computing*, pp. 546–574, Academic Press, New York, December 1987.

113. N. Ichiyoshi, T. Miyazaki, and K. Taki, "A Distributed Implementation of Flat GHC on the Multi-PSI," *Int'l Conf. on Logic Programming*, 1987.

114. N. Ito et al., "The Architecture and Preliminary Evaluation Results of the Experimental Parallel Inference Machine PIM-D," *Proc. 13th Int'l Symp. on Computer Architecture*, pp. 149–156, IEEE/ACM, 1986.

115. N. Ito, H. Shimizu, M. Kishi, E. Kuno, and K. Rokusawa, "Data-Flow Based Execution Mechanisms of Parallel and Concurrent Prolog," *New Generation Computing*, vol. 3, pp. 15–41, OHMSHA Ltd. and Springer-Verlag, New York, 1985.

116. P. Kanerva, *Parallel Structures in Human and Computer Memory*, RIACS Technical Report TR-86.2, NASA Ames Research Center, Moffett Field, CA, January 1986.

117. R. M. Keller, F. C. H. Lin, and J. Tanaka, "Rediflow Multiprocessing," *Proc. COMPCON Spring*, pp. 410–417, IEEE, 1984.

118. C. Kellogg, "Intelligent Assistants for Knowledge and Information Resources Management," *Proc. 8th Int'l Joint Conf. on Artificial Intelligence*, pp. 170–172, William Kaufmann, Los Altos, CA, 1983.

119. D. F. Kibler, "Parallelism in AI Programs," *Proc. Int'l Joint Conf. on Artificial Intelligence*, pp. 53–56, IJCAI, 1985.

120. M. King and M. Rosner, "Scanning the Issue: The Special Issue on Knowledge Representation," *Proc. IEEE*, vol. 74, no. 10, pp. 1299–1303, October, 1986.

121. J. de Kleer, "An Assumption-Based TMS," *Artificial Intelligence*, vol. 28, pp. 127–161, North-Holland, New York, 1986.

122. J. de Kleer, "Extending the ATMS," *Artificial Intelligence*, vol. 28, pp. 163–196, North-Holland, New York, 1986.

123. J. de Kleer, "Problem Solving with the ATMS," *Artificial Intelligence*, vol. 28, pp. 197–224, North-Holland, New York, 1986.

124. T. Knight, *The CONS Microprocessor*, AI Working Paper 80, MIT, Cambridge, November 1974.

125. R. Kowalski, "Predicate Logic as a Programming Language," *IFIP Information Processing*, pp. 569–574, North-Holland, New York, 1974.

126. R. Kowalski, *Logic for Problem Solving*, North-Holland, New York, 1979.

127. H. Kung and S. Song, *An Efficient Parallel Garbage Collection System and Its Correctness Proof*, Technical Report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, September 1977.

128. R. Kurzweil, "What is Artificial Intelligence Anyway?" *American Scientist*, vol. 73, no. 3, pp. 258–264, May–June, 1985.

129. T. H. Lai and S. Sahni, "Anomalies in Parallel Branch-and-Bound Algorithms," *Comm. of the ACM*, vol. 27, no. 6, pp. 594–602, June 1984.

130. L. E. Larson, J. F. Jensen, and P. T. Greiling, "GaAs High-Speed Digital IC Technology: An Overview," *Computer*, vol. 19, No. 10, pp. 21–28, IEEE, October 1986.

131. T. F. Lehr and R. G. Wedig, "Toward a GaAs Realization of a Production-System Machine," *Computer*, vol. 20, no. 4, pp. 36–49, IEEE, April 1987.

132. D. B. Lenat and J. McDermott, "Less Than General Production System Architectures," *Proc. 5th Int'l Joint Conf. on Artificial Intelligence*, pp. 923–932, William Kaufmann, Los Altos, CA, 1977.

133. D. B. Lenat, "The Ubiquity of Discovery," *Proc. 5th Int'l Joint Conf. on Artificial Intelligence*, pp. 1093-1105, William Kaufmann, Los Altos, CA, 1977.

134. D. B. Lenat, "The Nature of Heuristics," *Artificial Intelligence*, vol. 19, no. 2, pp. 189–249, North-Holland, New York, 1982.

135. D. B. Lenat, "Computer Software for Intelligent Systems," *Scientific American*, vol. 251, no. 3, pp. 204–213, Scientific American Inc., September 1984.

136. G.-J. Li and B. W. Wah, "Computational Efficiency of Parallel Approximate Branch-and-Bound Algorithms," *Proc. Int'l Conf. on Parallel Processing*, pp. 473–480, IEEE, August 1984.

137. G.-J. Li and B. W. Wah, "The Design of Optimal Systolic Algorithms," *Trans. on Computers*, vol. C-34, no. 1, pp. 66–77, IEEE, January 1985.

138. G.-J. Li and Benjamin W. Wah, "MANIP-2: A Multicomputer Architecture for Evaluating Logic Programs," *ICPP*, pp. 122-130, IEEE, June 1985.

139. G.-J. Li and B. W. Wah, "Multiprocessing of Logic Programs," *Proc. Int'l Conf. on Systems, Man and Cybernetics*, pp. 563–567, IEEE, October 1986.

140. G.-J. Li and B. W. Wah, "Optimal Granularity of Parallel Evaluation of AND-Trees," *Proc. Fall Joint Computer Conf.*, pp. 297–306, ACM/IEEE, November 1986.

141. G.-J. Li and B. W. Wah, "How Good are Parallel and Ordered Depth-First Searches?" *Proc. Int'l Conf. on Parallel Processing*, pp. 992–999, IEEE, August 1986.

142. H. Lieberman and C. Hewitt, "A Real-Time Garbage Collector Based on the Lifetimes of Objects," *Comm. of the ACM*, vol. 26, no. 6, pp. 419–429, June 1983.

143. G. Lindstrom and P. Panangaden, "Stream-Based Execution of Logic Programs," *Proc. Int'l Symp. on Logic Programming*, pp. 168–176, IEEE, February 1984.

144. R. P. Lippmann, "An Introduction to Computing with Neural Nets," *IEEE ASSP Magazine*, pp. 4–22, April, 1987.

145. F. J. Malabarba, "Review of Available Database Machine Technology," *Proc. Trends and Applications*, pp. 14–17, IEEE, 1984.

146. Y. Malachi, Z. Manna, and R. Waldinger, "TABLOG: A New Approach to Logic Programming," *Logic Programming*, ed. D. DeGroot and G. Lindstrom, Prentice-Hall, Englewood Cliffs, NJ, 1985.

147. G. Matthews, R. Hewes, and S. Krueger, "Single-Chip Processor Runs Lisp Environments," *Computer Design*, pp. 69–76, PennWell, May 1, 1987.

148. A. D. McAulay, "Spatial-Light-Modulator Interconnected Computers," *IEEE Computer*, vol. 20, no. 10, pp. 45–58, October 1987.

149. G. McCalla and N. Cercone, (eds.), "Special Issue on Knowledge Representation," *Computer*, vol. 16, no. 10, IEEE, October 1983.

150. J. McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I," *Comm. of the ACM*, vol. 3, no. 4, pp. 184–195, 1960.

151. J. McCarthy and P. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence," *Machine Intelligence 4*, pp. 463–502, Elsevier, NY, 1969.

152. J. McCarthy, "History of Lisp," *SIGPLAN Notices*, vol. 13, no. 8, pp. 217–223, ACM, 1978.

153. J. L. McClelland, D. D. Rumelhart, and G. Hinton, "The Appeal of Parallel Distributed Processing," *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, eds. D. E. Rumelhart, J. L McClelland, and the PDP Research Group, vol. 1, MIT Press, Cambridge, 1986.

154. J. F. McDonald, H. J. Greub, R. H. Steinvorth, B. J. Donlan, and A. S. Bergendahl, "Wafer Scale Interconnections for GaAs Packaging Applications to RISC Architecture," *Computer*, vol. 20, no. 4, pp. 21–35, IEEE, April 1987.

155. J. R. McGraw, "Data Flow Computing: Software Development," *Trans. on Computers*, vol. C-29, no. 12, pp. 1095–1103, IEEE, 1980.

156. C. S. Mellish, "An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter," *Logic Programming*, ed. K. Clark and S. A. Tarnlund, pp. 99–106, Academic Press, New York, 1982.

157. R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, *Machine Learning: An Artificial Intelligence Approach*, Tioga, Palo Alto, CA, 1983.

158. M. Minsky, "A Framework for Representing Knowledge," *The Psychology of Computer Vision*, ed. P. H. Winston, McGraw-Hill, New York, 1975.

159. D. I. Moldovan and Y. W. Tung, "SNAP: A VLSI Architecture for Artificial Intelligence Processing," *J. Parallel and Distributed Computing*, pp. 109–131, Academic Press, New York, May 1985.

160. L. Monier and P. Sidhu, "The Architecture of the Dragon," *Proc. COMPCON*, pp. 118–121, Spring 1985.

161. D. A. Moon, "Symbolics Architecture," *Computer*, vol. 20, no. 1, pp. 43–52, IEEE, 1987.

162. T. Moto-oka, "Overview to the Fifth Generation Computer System Project," *Proc. 10th Annual Int'l Symposium on Computer Architecture*, pp. 417–422, IEEE/ACM, June 1983.

163. T. Moto-oka, H. Tanaka, H. Aida, K. Hirata, and T. Maruyama, "The Architecture of a Parallel Inference Engine (PIE)," *Proc. Int'l Conf. on Fifth Generation Computer Systems*, pp. 479–488, ICOT and North-Holland, New York, 1984.

164. T. Moto-oka and H. S. Stone, "Fifth-Generation Computer Systems: A Japanese Project," *Computer*, vol. 17, no. 3, pp. 6–13, IEEE, March 1984.

165. A. Mukhopadhyay, "Hardware Algorithms for Nonnumeric Computation," *Transactions on Computers*, vol. C-28, no. 6, pp. 384–394, IEEE, June 1979.

166. K. Murakami, T. Kakuta, R. Onai, and N. Ito, "Research on Parallel Machine Architecture for Fifth-Generation Computer Systems," *Computer*, vol. 18, no. 6, pp. 76–92, IEEE, June 1985.

167. K. Murakami, "Research on Parallel Machine Architecture for Fifth Generation Computing Systems," *Computer*, p. 92, IEEE, vol. 18, no. 6, 1985.

168. M. Nagao, J. I. Tsujii, K. Nakajima, K. Mitamura, and H. Ito, "Lisp Machine NK3 and Measurement of Its Performance," *Proc. 6th Int'l Joint Conf. on Artificial Intelligence*, pp. 625–627, William Kaufmann, Los Altos, CA, August 1979.

169. H. Nakagawa, "AND Parallel Prolog with Divided Assertion Set," *Proc. Symp. on Logic Programming*, pp. 22–28, 1984.

170. R. Nakazaki et al., "Design of a High Speed Prolog Machine (HPM)," *Proc. 12th Int'l Symp. on Computer Architecture* , pp. 191–197, IEEE/ACM, 1985.

171. P. M. Neches, "Hardware Support for Advanced Data Management Systems," *Computer*, vol. 17, no. 11, pp. 29–40, IEEE, November 1984.

172. A. Newell, J. C. Shaw, and H. A. Simon, "Programming the Logic Theory Machine," *Proc. 1957 Western Joint Computer Conf.*, pp. 230–240, IRE, 1957.

173. A. Newell and H. A. Simon, in *Human Problem Solving*, Prentice-Hall, Englewood Cliffs, NJ, 1972.

174. A. Newell and H. Simon, "ACM Turing Award Lecture: Computer Science as an Empirical Inquiry: Symbols and Search," *Communications of the ACM*, vol. 19, no. 3, March 1975.

175. A. Newell, "Production Systems: Models of Control Structures," *Visual Information Processing*, ed. W. G. Chase, Academic Press, New York, 1975.

176. H. P. Nii, "Blackboard Systems, Blackboard Application Systems, Blackboard Systems from a Knowledge Engineering Perspective," *AI Magazine*, pp. 82–106. AAAI, August 1986.

177. K. Oflazer, "Partitioning in Parallel Processing of Production Systems," *Proc. Int'l Conf. on Parallel Processing*, pp. 92–100, IEEE, 1984.

178. E. A. Ozkarahan, S. A. Schuster, and K. C. Smith, "RAP—An Associative Processor for Database Management," *Proc. Nat'l Computer Conf.*, pp. 379–388, AFIPS Press, 1975.

179. D. A. Patterson, "Reduced Instruction Set Computers," *Comm. of the ACM*, vol. 28, no. 1, pp. 8–21, January 1985.

180. J. Pearl, *Heuristics—Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, Reading, MA, 1984.

181. J. Pearl, "Some Recent Results in Heuristic Search Theory," *Trans. on Pattern Analysis and Machine Intelligence*, vol. PAMI-6, no. 1, pp. 1–13, IEEE, January 1984.

182. E. von Puttkamer, "A Microprogrammed Lisp Machine," *Microprocessing and Microprogramming*, vol. 11, no. 1, pp. 9–14, North-Holland, New York, January 1983.

183. M. R. Quillian, "Word Concepts: A Theory and Simulation of Some Basic Semantic Capabilities," *Behavioral Science*, pp. 410–430, 1967.

184. C. V. Ramamoorthy, J. L. Turner, and B. W. Wah, "A Design of a Fast Cellular Associative Memory for Ordered Retrieval," *Transactions on Computers*, vol. C-27, no. 9, pp. 800–815, IEEE, September 1978.

185. C. V. Ramamoorthy, A. Prakash, W. T. Tsai, and Y. Usuda, Software Engineering, *Computer*, vol. 17, pp. 191–210, IEEE, October 1984.

186. C. V. Ramamoorthy, S. Shekhar, and V. Garg, "Software Development Support for AI Programs," *Computer*, pp. 30–42, IEEE, January 1987.

187. U. S. Reddy, "On the Relationship Between Logic and Functional Languages," *Logic Programming*, ed. D. DeGroot and E. G. Lindstrom, Prentice-Hall, Englewood Cliffs, NJ, 1985.

188. P. Robinson, "The SUM: An AI Co-Processor," *Byte*, pp. 169–180, McGraw-Hill, New York, June 1985.

189. F. Rosenblatt, *Principles of Neurodynamics*, Spartan Books, New York, 1962.

190. D. D. Rumelhart, G. Hinton, and J. L. McClelland, "A General Framework for Parallel Distributed Processing," *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, ed. D. E. Rumelhart, J. L. McClelland, and the PDP Research Group, vol. 1, MIT Press, Cambridge, 1986.

191. D. E. Rumelhart, J. L. McClelland, and the PDP Research Group (eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, MIT Press, Cambridge, 1986.

192. E. Sandewall, "An Approach to the Frame Problem, and its Implementation," *Machine Intelligence 7*, pp. 195–204, John Wiley and Sons, New York, 1972.

193. J. Sansonnet, D. Botella, and J. Perez, "Function Distribution in a List-Directed Architecture," *Microprocessing and Microprogramming*, vol. 9, no. 3, pp. 143–153, North-Holland, New York, 1982.

194. J. P. Sansonnet, M. Castan, and C. Percebois, "M3L: A List-Directed Architecture," *Proc. 7th Annual Symp. on Computer Architecture*, pp. 105–112, IEEE/ACM, May 1980.

195. J. P. Sansonnet, M. Castan, C. Percebois, D. Botella, and J. Perez, "Direct Execution of Lisp on a List-Directed Architecture," *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, pp. 132–139, ACM, March 1982.

196. D. Schaefer and J. Fischer, "Beyond the Supercomputer," *Spectrum*, vol. 19, no. 3, pp. 32–37, IEEE, March 1982.

197. M. Schor, "Declarative Knowledge Programming: Better Than Procedural," *Expert*, vol. 1, no. 1, pp. 36–43, IEEE, Spring 1986.

198. T. J. Sejnowski and C. R. Rosenberg, *NETtalk: A Parallel Network That Learns to Read Aloud*, Tech. Rep., Johns Hopkins University, Baltimore, January 1986.

199. G. Shafer, *A Mathematical Theory of Evidence*, Princeton Univ. Press, Princeton, NJ, 1976.

200. E. Shapiro and A. Takeuchi, "Object Oriented Programming in Concurrent Prolog," *New Generation Computing*, vol. 1, no. 1, pp. 25–48, OHMSHA Ltd. and Springer-Verlag, New York, 1983.

201. S. M. Shatz and J. P. Wang, "Introduction to Distributed-Software Engineering," *Computer*, pp. 23–32, IEEE, October 1987.

202. D. E. Shaw, *On the Range of Applicability of an Artificial Intelligence Machine*, Technical Report, Columbia University, New York, January 1985.

203. Y. Shih and K .B. Irani, "Large Scale Unification Using a Mesh-Connected Array of Hardware Unifiers," *Proc. Int'l Conf. on Parallel Processing*, pp. 787–794, Pennsylvania State University Press, 1987.

204. Y. Shobatake and H. Asio, "A Unification Processor Based on Uniformly Structured Cellular Hardware," *Proc. 13th Int'l Symp. on Computer Architecture*, pp. 140–148, IEEE/ACM, 1986.

205. H. A. Simon, "Search and Reasoning in Problem Solving," *Artificial Intelligence*, vol. 21, pp. 7–29, North-Holland, New York, 1983.

206. H. A. Simon, "Whether Software Engineering Needs to Be Artificially Intelligent," *Trans. on Software Engineering*, vol. SE-12, no. 7, IEEE, July 1986.

207. R. G. Smith, "The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver," *Trans. on Computers*, vol. C-29, no. 12, IEEE, 1980.

208. A. Synder, *Object-Oriented Programming for Common Lisp*, Report ATC-85-1, Software Technology Lab., Hewlett-Packard Lab., Palo Alto, CA, 1985.

209. C. Stanfill and D. Waltz, "Toward Memory-Based Reasoning," *Comm. of the ACM*, pp. 1213–1228, December 1986.

210. M. Stefik and D. G. Bobrow, "Object-Oriented Programming: Themes and Variations," *AI Magazine*, pp. 40–62, AAAI, Spring 1986.

211. S. J. Stolfo and D. E. Shaw, *DADO: A Tree-Structured Machine Architecture for Production Systems*, Technical Report, Columbia University, New York, March 1982.

212. S. J. Stolfo and D. P. Miranker, "The DADO Production System Machine," *J. Parallel and Distributed Computing*, vol. 3, no. 2, pp. 269–296, Academic Press, New York, 1986.

213. S. J. Stolfo, "Initial Performance of the DADO-2 Prototype," *Computer*, pp. 75–84, IEEE, January 1987.

214. H. S. Stone, "Parallel Querying of Large Databases: A Case Study," *Computer*, pp. 75–84, IEEE, January 1987.

215. S. Y. W. Su, "Associative Programming in CASSM and Its Applications," *Proc. 3rd Int'l Conf. on Very Large Databases*, pp. 213–228, Morgan Kaufmann, Los Altos, CA, 1977.

216. P. A. Subrahmanyam and J. H. You, "FUNLOG: A Computational Model Integrating Logic Programming and Functional Programming," *Logic Programming*, ed. D. DeGroot and G. Lindstrom, Prentice-Hall, Englewood Cliffs, NJ, 1985.

217. S. Sugimoto, K. Agusa, K. Tabata, and Y. Ohno, "A Multi-Microprocessor System for Concurrent Lisp," *Proc. Int'l Conf. on Parallel Processing*, pp. 135–143, IEEE, 1983.

218. G. J. Sussman and D. V. McDermott, "From PLANNER to CONNIVER–A Genetic Approach," *Fall Joint Computer Conf.*, vol. 41, pp. 129–137, AFIPS Press, 1972.

219. G. J. Sussman, J. Holloway, G. L. Steel, Jr., and A. Bell, "Scheme-79–Lisp on a Chip," *Computer*, vol. 14, no. 7, pp. 10–21, IEEE, July 1981.

220. K. Takahasi, H. Yamada, H. Hagai, and K. Matsumi, "A New String Search Hardware Architecture for VLSI," *Proc. 13th Int'l Symp. on Computer Architecture*, pp. 20–27, IEEE/ACM, 1986.

221. A. Takeuci and K. Fukukuwa, "Parallel Logic Programming Languages," *Proc. 3rd Int'l Conf. on Logic Programming*, Springer-Verlag, New York, 1986.

222. K. Taki, Y. Kaneda, and S. Maekawa, "The Experimental Lisp Machine," *Proc. 6th Int'l Joint Conf. on Artificial Intelligence*, pp. 865–867, William Kaufmann, Los Altos, CA, August 1979.

223. K. Taki, M. Yokota, A. Yamamoto, H. Nishikawa, S. Uchida, H. Nakashima, and A. Mitsuishi, "Hardware Design and Implementation of the Personal Sequential Interface Machine (PSI)," *Proc. Int'l Conf. on Fifth Generation Computer Systems*, pp. 398–409, ICOT and North-Holland, New York, 1984.

224. W. Teitelman and L. Masinter, "The Interlisp Programming Environment," *Computer*, vol. 14, no. 4, pp. 25–33, IEEE, April 1981.

225. M. F. M. Tenorio and D. I. Moldovan, "Mapping Production Systems into Multiprocessors," *Proc. Int'l Conf. on Parallel Processing*, pp. 56–62, IEEE, 1985.

226. E. Tick and D. H. D. Warren, "Towards a Pipelined Prolog Processor," *New Generation Computing*, vol. 2, no. 4, pp. 323–345, OHMSHA Ltd. and Springer-Verlag, New York, 1984.

227. S. Torii et al., "A Database System Architecture Based on a Vector Processing Method," *Proc. 3rd Int'l Conf. on Data Engineering*, IEEE, February 1987.

228. P. C. Treleaven and I. G. Lima, "Japan's Fifth-Generation Computer Systems," *Computer*, vol. 15, no. 8, pp. 79–88, IEEE, August 1982.

229. A. B. Tucker, Jr., "A Perspective on Machine Translation: Theory and Practice," *Comm. of the ACM*, vol. 27, no. 4, pp. 322–329, April 1984.

230. S. Uchida, "Inference Machines in FGCS Project," *Proc. VLSI Int'l Conf.*, IFIP TC-10, WG 10.5, August 1985.

231. K. Ueda, *Guarded Horn Clauses*, Technical Report TR-103, ICOT, Tokyo, Japan, 1985.

232. L. M. Uhr, "Parallel-Serial Production Systems," *Proc. 6th Int'l Joint Conf. on Artificial Intelligence*, pp. 911–916, William Kaufmann, Los Altos, CA, August 1979.

233. D. Unger, "Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm, "*ACM. Sigsoft*, vol. 9, no. 3, pp. 157–167, May 1984.

234. D. Unger, R. Blau, P. Foley, D. Samples, and D. A. Patterson. "Architecture of SOAR: Smalltalk on RISC," *Proc. 11th Annual Int'l Symp. on Computer Architecture*, pp. 188–197, IEEE/ACM, 1984.

235. D. Ungar and D. Patterson, "What Price Smalltalk?" *Computer*, pp. 67–74, IEEE, January 1987.

236. S. R. Vegdahl, "A Survey of Proposed Architectures for the Execution of Functional Languages," *Trans. on Computers*, vol. C-33, no. 12, pp. 1050–1071, IEEE, December 1984.

237. J. S. Vitter and R. A. Simons, "New Classes for Parallel Complexity: A Study of Unification and Other Complete Problems," *Trans. on Computers*, vol. C-35, no. 5, pp. 403–418, IEEE, May 1986.

238. B. W. Wah, G. J. Li, and C. F. Yu, "Multiprocessing of Combinatorial Search Problems," *IEEE Computer*, vol. 18, no. 6, pp. 93–108, June 1985. Also in Tutorial: Computers for Artificial Intelligence Applications, ed. B. W. Wah, IEEE Computer Society, 1986, pp. 173–188.

239. B. W. Wah, "Guest Editor's Introduction: New Computers for Artificial Intelligence Processing," *Computer*, vol. 20, no. 1, pp. 10–15, IEEE, January 1987.

240. D. L. Waltz, "Applications of the Connection Machine," *Computer*, vol. 20, no. 1, IEEE, January 1987.

241. D. H. Warren, L. M. Pereira, and F. Pereira, "Prolog—The Language and Its Implementation Compared with Lisp," *Proc. Symp. on Artificial Intelligence and Programming Languages*, also *SIGART Newsletter*, vol. 64, pp. 109–115, ACM, August 1977.

242. P. Wegner and B. Shriver (eds.), "Special Issue on Object-Oriented Programming Workshop," *SIGPLAN Notices*, vol. 21, no. 10, ACM, October 1986.

243. M. Weiser, S. Kogge, M. McElvany, R. Pierson, R. Post, and A. Thareja, "Status and Performance of the ZMOB Parallel Processing System," *Proc. COMPCON Spring*, pp. 71–73, IEEE, February 1985.

244. L. C. West, "Picosecond Integrated Optical Logic," *Computer*, vol. 20, pp. 34–47, IEEE, December 1987.

245. T. Winograd, "Frame Representations and the Declarative/Procedural Controversy," *Representation and Understanding: Studies in Cognitive Science*, pp. 185–210, Academic Press, New York, 1975.

246. P. H. Winston and B. Horn, *Lisp*, Second Edition, Addison-Wesley, Reading, MA, 1984.

247. Y. Yamaguchi, K. Toda, and T. Yuba, "A Performance Evaluation of a Lisp-Based Data-Driven Machine (EM-3)," *Proc. 10th Annual Int'l Symp. on Computer Architecture*, pp. 363–369, IEEE/ACM, June 1983.

248. Y. Yamaguchi, K. Todo, J. Herath, and T. Yuba, "EM-3: A Lisp-Based Data-Driven Machine," *Proc. Int'l Conf. on Fifth Generation Computer Systems*, pp. 524–532, ICOT and North-Holland, New York, 1984.

249. C. F. Yu, *Efficient Combinatorial Search Algorithms*, Ph.D. Thesis, School of Electrical Engineering, Purdue University, West Lafayette, IN, December 1986.

250. L. A. Zadeh, "Fuzzy Sets," *Information and Control*, pp. 338–353, 1965.

251. L. A. Zadeh, "Approximate Reasoning Based on Fuzzy Logic,"*Proc. 6th Int'l Joint Conf. on Artificial Intelligence*, pp.1004–1010, William Kaufmann, Los Altos, CA, August 1979.