AI LANGUAGES AND PROCESSING

Programming languages have been crucial in the development of the artificial intelligence (AI) branch of computer science, for at least two reasons. First, they allow convenient implementation and modification of programs that demonstrate and test AI ideas. Second, they provide vehicles of thought—they allow the user to concentrate on higher-level concepts. Frequently, new ideas in AI are accompanied by a new language in which it is natural to apply these ideas.

The process of programming a solution to a problem is inherently difficult. This has been recognized by conventional programmers for many years and has been one of the motivating forces behind both structured and object-oriented programming techniques. The problem seems to be that the human brain does not have the capacity to handle the complexity of the programming task for nontrivial problems. The solution has been to use first structured and then object-oriented techniques, which break problems up into manageable "chunks." However, this *divide et impera* technique did

J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*. Copyright © 1999 John Wiley & Sons, Inc.

not solve the problem of the imperative (procedural, commanding) description of the solution, that is, of the explicit ordering of the actions leading to the solution. Moreover, the sequence of statements in imperative language also implies the need to have explicit commands to alter the sequence, for example, control structures such as "while . . ., do," "repeat . . . until," or even "goto." Many errors in imperative languages are introduced because the specified sequencing is not correct. On the other hand, in declarative languages, used mainly for AI programming, we describe the problem itself rather than the explicit way to solve it or the order in which things must be done. The explicit ordering has been replaced by the implicit ordering, conditioned by the relationships between the objects. The avoidance of an explicit sequence of control relieves the user of the burden of specifying the control flow in the program.

Declarative programming is the umbrella term that covers both functional programing and relational programming. While the two approaches do have many superficial similarities—both classes of languages are nonprocedural and, in their pure forms, involve programming without side effects—they have different mathematical foundations. In writing functional programs, the programmer is concerned with specifying the solution to a problem as a collection of many-to-one transformations. This corresponds closely to the mathematical definition of a function. On the other hand, a relational program specifies a collection of many-to-many transformations. Thus in relational programming languages, there is a set of solutions to a particular application rather than the single solution that is produced from a functional application. Although the execution mechanisms that have been proposed for relational programming languages are radically different from the approaches for functional programming languages, both approaches have been widely used in AI programming.

To provide AI-related comparison, we have included two equally popular AI-language alternatives, a functional language Lisp and relational language Prolog. From the beginning, Lisp was the language of choice for US AI researchers. The reasons are many, but primarily result from the strong mathematical roots of the language, its symbolic rather than numeric processing, and its ability to treat its own code as data. Researchers have exploited this capability of Lisp programs to modify themselves at run time for research in machine learning, natural language understanding, and other aspects of AI. Moreover, AI programming requires the flexibility, the extensibility, the modularity, and the underlying data structures and data abstraction facilities that Lisp provides. Although Lisp is one of the older programming languages in use, it has remained the most widely used language in AI programming.

The logic programming language Prolog has been growing in popularity since it was originally introduced in Europe in the early 1970s. Prolog is most easily matched to tasks involving logic and prooflike activities. A Prolog program is essentially a description of objects and relations between them. A subset of formal logic (called Horn clause logic) is used to specify the desired conditions. Prolog's adherents believe that it is easier to learn and use than Lisp. They say that it uses less memory and is more easily moved from one computer to another. In the past, it has run with reasonable speed only on mainframes, but recent modifications are running satisfactorily even on personal computers.

Although both Lisp and Prolog have been supported with almost religious intensity by passionate advocates, the conflict between them has softened over the years, and many now believe in a combination of ideas from both worlds (see section on hybrid languages).

Before we discuss specific AI programming paradigms and languages, it will be useful to underline the specific features that facilitate the production of AI programs as distinct from other types of applications. Apart from the features that are now needed for building almost any kind of complex systems, such as possessing a variety of data types, a flexible control structure, and the ability to produce efficient code, the features that are particularly important in building AI systems are (1,2):

- Good symbol manipulation facilities, because AI is concerned with symbolic rather than numeric processing
- Good list-manipulating facilities, because lists are the most frequently used data structures in AI programs
- Late binding times for the object type or the data structure size, because in many AI systems it is not possible to define such things in advance
- Pattern-matching facilities, both to identify data in the large knowledge base and to determine control for the execution of production systems
- Facilities for performing some kind of automatic deduction and for storing a database of assertions that provide the basis for deduction
- Facilities for building complex knowledge structures, such as frames, so that related pieces of information can be grouped together and assessed as a unit
- Mechanisms by which the programmer can provide additional knowledge (metaknowledge) that can be used to focus the attention of the system where it is likely to be the most profitable
- Control structures that facilitate both goal-directed behavior (top-down processing or backward chaining) and data-directed behavior (bottom-up processing or forward chaining)
- The ability to intermix procedural and declarative knowledge in whatever way best suits a particular task
- A good programming environment, because AI programs are among the largest and most complex computer systems ever developed and present formidable design and implementation problems

No existing language provides all of these features. Some languages do well at one at the expense of others; some hybrid languages combine multiple programming paradigms trying to satisfy as many of these needs as possible. However, the main differentiator between various AI programming languages is their ability to represent knowledge clearly and concisely. Therefore, in the following section we present a summary of some of the basic knowledge representation paradigms. As each language is discussed, we look at some of the ways in which language represents various types of knowledge and satisfies other above-mentioned demands.

KNOWLEDGE REPRESENTATION

Knowledge representation is one of the most basic and actively researched areas of artificial intelligence. This research has thrown up a number of schemes for knowledge representation, each of which has both strong and weak points. The computational efficiency and the clarity of the representation are the most important aspects, both of which strongly depend on the nature of the AI application. Therefore, the choice of the representation formalism should be based on an analysis of the task to be performed with it, so as to ensure that the representation is well matched to the problem. The choice must not be based on any advocacy of a particular representational paradigm as adequate or natural, independent of the problem to be solved.

We may distinguish three types of adequacy of the representation of knowledge: metaphysical adequacy, epistemological adequacy, and heuristic adequacy. Metaphysical adequacy obtains if there are no contradictions between the facts we wish to represent and our representation of them. Epistemological adequacy is about the ability to express knowledge in our representation, and heuristic adequacy obtains if we can express in our representation the problem-solving process that we need to tackle a problem. Given a representation that is adequate on the above criteria, it is vital to check whether it is computationally tractable. For instance, natural language is an epistemologically adequate way of representing anything at all, but it fails on the computational criterion, since we cannot build AI systems that can make use of knowledge represented in this way in anything like an efficient manner.

Apart from the computational efficiency, we will need to consider a variety of other factors that are relevant to the desirability of a representation. One of the reasons for separating knowledge into a knowledge base is that by so doing it is possible to isolate the knowledge used in problem solving from the problem-solving strategies themselves, as well as to use the same problem-solving strategies in a variety of domains.

Another important factor is the clarity and comprehensibility of representation, because the builder of a system is rarely an expert in the field covered by the system, and both the knowledge engineer and the domain expert should understand the representation.

Related to the clarity is the conciseness of the representation. Other things being equal, the more concise a representation, the more likely it is to be easily understood. Conciseness can also have implications for computational efficiency.

Another factor that cannot be overlooked is the tools that will be available to support building of the knowledge base. In contrast to conventional database systems, AI systems require a knowledge base with diverse kinds of knowledge. These include, but are not limited to, knowledge about objects, knowledge about processes, and hard-to-represent commonsense knowledge about goals, motivations, causality, time, actions, etc. Attempt to represent this breadth of knowledge raise many questions:

- How should the explicit knowledge be structured in a knowledge base?
- How should rules for manipulating a knowledge base's explicit knowledge be encoded to infer knowledge contained implicitly within the knowledge base?
- When do we undertake and how do we control such inferences?
- How do we formally specify the semantics of a knowledge base?
- How do we deal with incomplete knowledge?
- How do we extract the knowledge of an expert to initially "stock" the knowledge base?
- How do we automatically acquire new knowledge as time goes on so that the knowledge base can be kept current?

In most early AI systems, knowledge representation was not explicitly recognized as an important issue in its own right, although most systems incorporated knowledge indirectly through rules and data structures. During the mid-1960s knowledge representation slowly emerged as a separate area of study. Several different approaches to knowledge representation began to manifest themselves and have resulted in the various formalisms in use today. The most important approaches are first-order logic, semantic networks, O–A–V triples, frames, and production systems. This is necessarily an oversimplification, since not all knowledge representation formalisms will fit into one of these approaches.

All of them have both strong and weak points. From our representational paradigm we want first computational efficiency and second clarity of represented knowledge, both of which depend on the nature of our application. Therefore we need to base our choice of representation on an analysis of the task to be performed with it. Also, all these knowledge representation paradigms have cross-fertilized each other. Currently popular are hybrid or multiparadigm languages and commercial products know as AI toolkits, which enable a wider variety of representational paradigms and therefore have been successful in a huge spectrum of applications.

The usefulness of *first-order logic* in a knowledge representation context became evident during the 1960s, primarily as an outgrowth of research into automatic theorem proving. In this paradigm, the knowledge is represented as a set of axioms, while the inference comprises the proving of theorems from these axioms. Much research was directed at investigating the use of the resolution principle as an inference technique in various applications. Other research attempted to recast logical formalisms in a more computationally oriented framework. This has led to intense discussion regarding the pros and cons of logic-based approaches to representation. Concern has been expressed about the lack of an explicit scheme to index into relevant knowledge, the awkwardness of handling changing or incomplete knowledge, and perceived limitations of deductive inference. However, logic advocates muster counterarguments to many of these concerns, and there is no doubt that the formal precision and interpretability of logic are useful and supply expressiveness that other knowledge representation schemes lack. This kind of representation has experienced a surge of popularity, largely because of the availability of Prolog, which effectively provides an efficient theorem prover for a subset of first-order logic.

One of the oldest and the most general representational schemes, which came to prominence during the mid- to late 1960s, is the *semantic network.* Such networks are usually thought of as graphs consisting of a set of *nodes* representing concepts, linked by *arcs* representing the relationships between the concepts and associated with specialized inference procedures that can operate on the structure. *Concepts* are used to represent either physical objects that can be seen or touched, or conceptual entities such as events, acts or abstract categories. A link or arc may represent any type of relationship. The most popular are:

- The IS-A link, used to represent the class–instance or superclass–subclass relationships (for instance, a relationship between the subclass "dog" and its superclass "mammal," i.e., "Dog IS-A mammal," or the instance Layka and the class "dog," i.e., "Layka IS-A dog"). The most popular kind of inference has involved the inheritance of information from the top levels of hierarchy downward, along these IS-A links. Such an organization allows information to be shared among many nodes and thus leads to a large-scale representational economies.
- The HAS-A link, which identifies nodes that are properties of another nodes and shows part–subpart relationships.

Flexibility is a major advantage of this representational scheme. New nodes and links can be defined as needed.

Some AI researchers use *object–attribute–value (O–A–V) triples* that each look like a link on a semantic net. However, an O–A–V scheme is sometimes used to represent known facts, rather than a particular logical structure as in the semantic net. In an expert system, a program may simply gather information before fitting it into the knowledge base. Alternatively, O–A–V triples may be used to create a data structure like a blank form. The blanks are said to be uninstantiated values. In exercising an expert system, general and case-specific information exist, and both can be represented using O–A–V triples. This representational scheme is used in MYCIN, the first well-known expert system, built at Stanford University at 1972.

Marvin Minsky (3) postulated that a useful way to organize a knowledge base was to break it into highly modular "almost decomposable" chunks called *frames* (sometimes also referred to as *schemata*). They associate an object with a collection of features, and are similar to a property list or record, used in conventional programming. Each feature is stored in a slot (frame variable). Slots may also contain default values, pointers to other frames, sets of rules, or procedures by which values may be obtained. Default values are quite useful when representing knowledge in domains where exceptions are rare. A *procedural attachment* is another way that a slot in a frame can be filled. In this case the slot contains instructions for determining an entry. These are essentially pieces of code (often called *demons*) associated with slots, which are invoked when the slots are accessed. The inclusion of procedures in frames joins together in a single representational strategy two complementary (and historically competing) ways to state and store facts: *procedural* and *declarative* representation. The two perspectives, considered as two complementary aspects of knowledge, are often referred to as *dual semantics.* Frames gain power, generality, and popularity by their ability to integrate both procedural and declarative semantics, and so they became the basis for another major school of knowledge representation. Dividing a knowledge base into frames has become common in a variety of applications, such as computer vision and natural language understanding. Frames are particularly useful when used to represent knowledge of certain stereotypical concepts or events. When one of these standard concepts or events is recognized, slots inside the appropriate frame can be filled in by tokens representing the actual actors or actions. After this step, much "precompiled" knowledge can be gleaned directly from frames or deduced via inference. Often a distinction is made between scripts with little capability for inference and more procedurally oriented frames.

Production system architectures are another way of representing knowledge. Proposed by A. Newell (4), production systems were originally presented as models of human reasoning. A set of production rules (each essentially a condition–action or a premise–conclusion pair) operate on a short-term memory buffer of relevant concepts. A basic control loop tries each rule in turn, executing the action part of the rule only if the condition part matches. Additionally, there will be some principle, known as a *conflict resolution principle,* that determines which rule fires when several rules match. Representing knowledge as condition–action pairs has proved to be a very natural way of extracting and encoding rule-based knowledge in many applications, and now production systems are widely used to construct special-purpose knowledge-based systems, so called *expert systems.* Some expert systems have rules that incorporate pattern-matching *variables.* In such systems, the variable rule allows the system to substitute many different facts into the same general format.

Given the diversity of these knowledge representation paradigms, we need to consider how we should approach the selection of one against other. Although people have been prepared to champion one formalism against another, in fact, as regards expressive power, they can all be viewed as equivalent to first-order logic or a subset thereof. However, the important point is that they are not all equivalent in terms of pragmatic considerations, most obviously that of computational efficiency. But the computational and other pragmatic benefits from one representation form to another will vary according to the problem at hand. There is therefore little point in arguing the merits of the various formalisms independently of an understanding of the work that we wish to do with the formalism in our system.

A serious shortcoming of all above-mentioned conventional approaches to knowledge representation is that they are based on bivalent logic and therefore do not provide an adequate model for representing and processing of uncertain and imprecise knowledge. Fuzzy logic, which may be viewed as an extension of classical logical systems, provides an effective conceptual framework for dealing with the problem of knowledge representation in an environment of uncertainty and imprecision.

LOGIC PROGRAMMING

Logic programming began in the early 1970s as a direct outgrowth of earlier work in automatic theorem proving and artificial intelligence. It can be defined as the use of symbolic logic for the explicit representation of problems, together with the use of controlled logical inference for the effective solution of those problems (5)

Constructing automatic deduction systems is central to the aim of achieving artificial intelligence. Building on work of Herbrands (6) in 1930, there was much activity in theorem proving in the early 1960s by Prawitz (7), Gilmore (8), Davis and Putnam (9), and others. This effort culminated in 1965 with the publication of the landmark paper by Robinson (10), which introduced the resolution rule. Resolution is an inference rule that is particularly well suited to automation on a computer.

The credit for the introduction of logic programming goes mainly to Kowalski and Kuehner (11,12) and Colmerauer et al. (13), although Green (14) and Hayes (15) should also be mentioned in this regard. In 1972, Kowalski and Colmerauer were led to the fundamental idea that logic can be used as a programming language. The name Prolog (for "programming in logic") was conceived and the first Prolog interpreter was implemented in the language Algol-W by Roussel in 1972.

The idea that first-order logic, or at least substantial subset *s* of it, can be used as a programming language was revolutionary, because until 1972, logic had been used only as a specification language in computer science. However, it has been shown that logic has a procedural interpretation, which makes it very effective as a programming language. Briefly, a program clause $A \Leftarrow B_1, \ldots, B_n$ is regarded as a procedure definition. If $\Leftarrow C_1, \ldots, C_k$ is a goal clause, then each C_j is regarded as a procedure call. A program is run by giving it an initial goal. If the current goal is $\Leftarrow C_1, \ldots, C_k$, a step in the computation involves unifying some C_j with the head A of a program clause $A \Leftarrow B_1, \ldots, B_n$ and thus reducing the current goal to the goal $\Leftarrow(C_1, \ldots, C_{j-1}, B_1, \ldots, B_n, C_{j+1}, \ldots, C_k)\theta$, where θ is the unifying substitution. Unification thus becomes a uniform mechanism for parameter passing, data selection, and data construction. The computation terminates when the empty goal is produced.

One of the main ideas of logic programming, which is due to Kowalski, is that an algorithm consist of two disjoint components, the logic and the control. The logic is the statement of *what* the problem is that has to be solved. The control is the statement of *how* it is to be solved. The ideal of logic programming is that the programmer should only have to specify the logic component of an algorithm. The logic should be exercised solely by the logic programming system. Unfortunately, this ideal has not yet been achieved with current logic programming systems, because of two broad problems. The first of these is the control problem. Currently, programmers need to provide a lot of control information, partly by the ordering of clauses and atoms in clauses and partly by extralogical control features, such as *cut.* The second problem is the negation problem. The Horn clause subset of logic does not have sufficient expressive power, and hence Prolog systems allow negative literals in the bodies of clauses.

Logic has two other interpretations. The first of these is the database interpretation. Here a logic program is regarded as a database. We thus obtain a very natural and powerful generalization of relational databases, which correspond to logic programs consisting solely of ground unit clauses. The concept of logic as a uniform language for data, programs, queries, views, and integrity constraints has great theoretical and practical potential.

The third interpretation of logic is the process interpretation. In this interpretation, a goal $\Leftarrow B_1, \ldots, B_n$ is regarded as a system of concurrent processes. A step in the computation is the reduction of a process to a system of processes. Shared variables act as communication channels between processes. There are now several Prologs based on the process interpretation. This interpretation allows logic to be used for operating-system applications and object-oriented programming.

It is clear that logic provides a single formalism for apparently diverse parts of computer science. Logic provides us with a general-purpose, problem-solving language, a concurrent language suitable for operating systems, and also a foundation for database systems. This range of applications, together with the simplicity, elegance, and unifying effect of logic programming, assures it of an important and influential future.

Deductive Databases

The last decade has seen substantial efforts in the direction of merging logic programming and database technologies for the development of large and persistent knowledge bases (see e.g. Refs. 16–21). The efforts differ in the degree of coupling between the two paradigms. A lot of pragmatic attempts fall into the *loose coupling* category, where existing logic programming and database environments (usually Prolog and relational databases) are interconnected through ad hoc interfaces. Although some interesting results have been achieved, recent research results have shown that simple interfaces are not efficient enough and that an enhancement in efficiency can be achieved by intelligent interfaces. It has become obvious that stronger integration is needed and that knowledge base management systems should provide direct access to data and should support rule-based interaction as one of the programming paradigms. Deductive databases and the Datalog language are the first steps in this direction, and will be discussed in detail later in this entry.

Deductive database systems are database management systems whose query language and storage structure are designed around a *logical model of data.* As relations are naturally thought of as the "values" of logical predicates, and relational languages such as SQL are syntactic sugarings of a limited form of logical expression, it is easy to see deductive database systems as an advanced form of relational system.

Compared with other extensions of relational systems (the object-oriented system, for instance), deductive databases have the important property of being *declarative,* that is, of allowing the user to query or update by saying what he or she wants, rather than how to perform the operation. Since declarativeness is a major peculiarity of relational systems and is now being recognized as an important driver of their success, deductive databases are nowadays considered the natural development of relational systems.

Even though deductive database systems have not yet obtained success on the database market, we see deductive database technology infiltrating other branches of database systems, especially the object-oriented world, where it is becoming increasingly important to interface object-oriented and logical paradigms in so-called declarative and object-oriented databases (DOODs).

A deductive database D consists of the following (16):

- A set P of *base* predicates and, for each predicate, an associated set of facts
- A set Q of *built-in* predicates (their associated sets of facts are assumed to be known)
- A set R of *derived* predicates, and for each predicate, an associated set of rules (each predicate is the head of each of its associated rules)
- A set S of *integrity constraints*

The predicates in P, Q, and R are disjoint. The first two sets are referred to as the *extensional database* (EDB), and the last two sets are referred to as the *intensional database* (IDB). The entire database is understood as collection of axioms (it must be consistent), and the resolution principle is established as a rule of inference. A *query* is a rule, whose head predicate is always called Q. The variables that appear only in its head are free. Assuming that Q has free variables $X = (X_1, \ldots, X_n)$, a tuple of constants $a = (a_1, \ldots, a_n)$ belongs to the (extensional) answer to Q if the substitution of a_i for X_i ($i = 1, \ldots, n$) yields a theorem. Relatively little effort (e.g., tabulation, sorting, grouping) is required for adequate presentation of an extensional answer to the user. This is because the extensional information is relatively simple, and all users may be assumed to be familiar with its form and meaning. Intensional information is more complex (e.g., rules, constants, hierarchies, views), and the user may not always be assumed to be familiar with its form and meaning. Hence, the presentation of intensional answers may require more effort.

Inductive Logic Programming

Inductive logic programming (ILP) is a research area formed at the intersection of machine learning and logic programming. ILP systems develop predicate descriptions from examples and background knowledge. The examples, background knowledge, and final descriptions are all described as logic programs. A unifying theory of ILP is being built up around lattice-based concepts such as refinement, least general generalization, inverse resolution, and most specific corrections. In addition to a well-established tradition of learning-in-the-limit results, some results within Valiant's PAC learning framework have been demonstrated for ILP systems. *U*-learnability, a new model of learnability, has also been developed.

Presently successful applications areas for ILP systems include the learning of structure–activity rules for drug design, finite-element mesh analysis design rules, primary–secondary prediction of protein structure, and fault diagnosis rules for satellites.

LOGIC LANGUAGES

One of the most important practical outcomes of the research in logic programming has been the language Prolog, based on the Horn clause subset of logic. The majority of logic programming systems available today are either Prolog interpreters or Prolog compilers. Most use the simple computation rule that always selects the leftmost atom in a goal. However, logic programming is by no means limited to Prolog. It is essential not only to find more appropriate computation rules, but also to find ways to program in larger subsets of logic, not just clausal subsets. In this entry we will also briefly cover a database query language based on logic programming, Datalog, and several hybrid languages supporting the logic programming paradigm (together with some other paradigms—functional, for instance).

Prolog

Prolog emerged in the early 1970s to use logic as a programming language. The early developers of this idea included Robert Kowalski at Edinburgh (on the theoretical side), Maarten van Emden at Edinburgh (experimental demonstration), and Alain Colmerauer at Marseilles (implementation). The present popularity of Prolog is largely due to David Warren's efficient implementation at Edinburgh in the mid 1970s.

Prolog has rapidly gained popularity in Europe as a practical programming tool. The language received impetus from its selection in 1981 as the basis for the Japanese Fifth Generation Computing project. On the other hand, in the United States its acceptance began with some delay, due to several factors. One was the reaction of the "orthodox school" of logic programming, which insisted on the use of pure logic that should not be marred by adding practical facilities not related to logic. Another factor was previous US experience with the Microplanner language, also akin to the idea of logic programming, but inefficiently implemented. And the third factor that delayed the acceptance of Prolog was that for a long time Lisp had no serious competition among languages for AI. In research centers with a strong Lisp tradition, there was therefore natural resistance to Prolog.

The language's smooth handling of extremely complex AI problems and ability to effect rapid prototyping have been big factors in its success, even in the US. Whereas conventional languages are procedurally oriented, Prolog introduces the descriptive, or declarative view, although it also supports the procedural view. The declarative meaning is concerned only with the *relations* defined by the program. This greatly alters the way of thinking about problem and makes learning to program in Prolog an exciting intellectual challenge. The declarative view is advantageous from the programming point of view. Nevertheless, the procedural details often have to be considered by the programmer as well.

Apart from this dual procedural–declarative semantics, the key features of Prolog are as follows (22,23):

- Prolog programming consists of defining *relations* and querying about relations.
- A program consists of *clauses.* These are of three types: *facts, rules,* and *questions.*
- A relation can be specified by *facts,* simply stating the *n*-tuples of objects that satisfy the relation, or by stating *rules* about the relation.
- A *procedure* is a set of clauses about the same relation.
- Querying about relations, by means of *questions,* resembles querying a database. Prolog's answer to a question consists of a set of objects that satisfy the question.
- In Prolog, to establish whether an object satisfies a query is often a complicated process that involves logical inference, exploring among alternatives, and possibly *backtracking.* All this is done automatically by the Prolog system and is, in principle, hidden from the user.

Different programming languages use different ways of representing knowledge. They are designed so that the kind of information you can represent, the kinds of statements you can make, and the kinds of operations the language can handle easily all reflect the requirements of the classes of problems for which the language is particularly suitable. The key features of Prolog that give it its individuality as a programming language are:

- Representation of knowledge as *relationships* between objects (the core representation method consists of relationships expressed in terms of a predicate that signifies a relationship and arguments or objects that are related by this predicate)
- The use of *logical rules* for deriving implicit knowledge from the information explicitly represented, where both the logical rules and the explicit knowledge are put in the *knowledge base* of information available to Prolog
- The use of *lists* as a versatile form of structuring data, though not the only form used
- The use of *recursion* as a powerful programming technique
- The assignment of values to variables by a process of *pattern matching* in which the variables are *instantiated,* or bound, to various values

The simplest use of Prolog is as a convenient system for retrieving the knowledge explicitly represented, i.e., for interrogating or querying the knowledge base. The process of asking a question is also referred to as setting a *goal* for the system to satisfy. One types the question, and the system searches the knowledge base to determine if the information one is looking for is there.

The next use of Prolog is to supply the system with part of the information one is looking for, and ask the system to find a missing part.

In both cases, Prolog works fundamentally by pattern matching. It tries to match the pattern of our question to the various pieces of information in the knowledge base.

The third case has a distinguishing feature. If a question contains variables (a word beginning with an uppercase letter), Prolog also has to find what are the particular objects (in place of variables) for which the goal are satisfied. The particular instantiations of variables to these objects are shown to the user.

One of the advantages of using Prolog is that a Prolog interpreter is in essence a built-in inference engine that draws logical conclusions using the knowledge supplied by the facts and rules.

To program in Prolog, one specifies some facts and rules about objects and relationships and then asks questions about the objects and relationships. For instance, if one entered the facts

```
likes (peter, mary)
likes (paul, mary)
likes (mary, john)
```

and then asked

```
?-likes (peter, mary)
```

Prolog would respond by printing

```
yes.
```

In this trivial example, the word `likes` is the *predicate* that indicates that such a relationship exists between one object, `peter`, and a second object, `mary`. In this case Prolog says that it can establish the truth of the assertion that Peter likes Mary, based on the three facts it has been given. In a sense, computation in Prolog is simply controlled logical deduction. One simply states the facts that one knows, and Prolog can tell whether or not any specific conclusion could be deduced from those facts. In knowledge engineering terms, Prolog's control structure is logical inference.

Prolog is the best current implementation of logic programming, given that a programming language cannot be strictly logical, since input and output operations necessarily entail some extralogical procedures. Thus, Prolog incorporates some basic code that controls the procedural aspects of its operations. However, this aspect is kept to a minimum, and it is possible to conceptualize Prolog as strictly a logical system.

Indeed, there are two Prolog programming styles: a *declarative* style and a *procedural* style. In declarative programming, one focuses on telling the system what it should know and relies on the system to handle the procedures. In procedural programming, one considers the specific problem-solving behavior the computer will exhibit. For instance, knowledge engineers who are building a new expert system concern themselves with procedural aspects of Prolog. Users, however, need not to worry about procedural details and are free simply to assert facts and ask questions.

One of the basic demands that an AI language should satisfy is good list processing. The *list* is virtually the only complex data structure that Prolog has to offer. A list is said to have a head and a tail. The head is the first list item. The tail is the list composed of all of the remaining items. In Prolog notation, the atom on the left of vertical bar is the list head, and the part to the right is the list tail.

The following example illustrates the way the list appending operation is performed in Prolog:

```
append ([], L,L).
append ([X|L1],L2,[X|L3]).
  :-append (L1,L2,L3).
```

This simple Prolog program consists of two relations. The first says that the result of appending the empty list (`[]`) to any list `L` is simply `L`. The second relation describes an inference rule that can be used to reduce the problem of computing the result of an append operation involving a shorter list. Using this rule, eventually the problem will be reduced to appending the empty list, and the value is given directly in the first relation. The notation `[X|L1]` means the list whose first element is `X` and the rest of which is `L1`. So the second relation says that the result of appending `[X|L1]` to `L2` is `[X|L3]` provided that it can be shown that the result of appending `L1` to `L2` is `L3`.

Datalog

Datalog (24) is a database query language based on the logic programming paradigm, and in many respects represents a simplified version of general logic programming. In the context of general logic programming it is usually assumed that all the knowledge (facts and rules) relevant to a particular application is contained within a single logic program *P*. Datalog, on the other hand, has been developed for the applications that use a large number of facts stored in a relational database. Therefore, two different sets of clauses should be considered—a set of ground facts, called the *extensional database* (EDB), physically stored in a relational database, and a Datalog program *P* called the *intensional database* (IDB). The predicates occurring in the EDB and in *P* are divided into two disjoint sets: the *EDB predicates,* which are all those occurring in the extensional database, and the *IDB predicates,* which occur in P but not in the EDB. It is necessary that the head predicate of each clause in P be an IDB predicate. EDB predicates may occur in P, but only in clause bodies.

Ground facts are stored in a relational database; it is assumed that each EDB predicate r corresponds to exactly one relation R of the database such that each fact $r(c_1, \ldots, c_n)$ of the EDB is stored as a tuple $\langle c_1, \ldots, c_n \rangle$.

Also, the IDB predicates of P can be identified with relations, called *IDB relations,* or *derived relations,* defined through the program P and the EDB. IDB relations are not stored explicitly, and correspond to relational *views.* The materialization of these views, i.e. their effective computation, is the main task of a Datalog compiler or interpreter.

The semantics of a Datalog program P can be defined as a mapping from database states (collections of EDB facts) to result states (IDB facts). A more formal definition of the logical semantics of Datalog a be found in Ref. 24, p. 148:

> Each Datalog fact F can be identified with an atomic formula F^* of First-Order Logic. Each Datalog rule R of the form L_0: -$L_1, \ldots, L_n$ represents a first-order formula R^* of the form $\forall X_1, \ldots, \forall X_m$ $(L_1 \wedge \cdots \wedge L_n \Rightarrow L_0)$, where $X_1, \ldots, X_m$ are all variables occurring in R. A set S of Datalog clauses corresponds to the conjunction S^* of all formulas C^* such that $C \in S$.

The *Herbrand base* HB is the set of all facts that we can express in the language of Datalog, i.e., all literals of the form $P(c_1, \ldots, c_k)$ such that c_i are constants. Furthermore, let *EHB* denote the extensional part of the Herbrand base, i.e., all literals of *HB* whose predicate is an EDB predicate, and, accordingly, let *IHB* denote the set of all literals of *HB* whose predicate is an IDB predicate. If S is a finite set of Datalog clauses, we denote by *cons*(S) the set of all facts that are logical consequences of S^*.

The semantics of a Datalog program P can be described as a mapping $\mathscr{M}_P$ from *EHB* to *IHB* which to each possible extensional database $E \subseteq EHB$ associates the set $\mathscr{M}_P(E)$ of intensional *result facts* defined by $\mathscr{M}_P(E) = cons(P \cup E) \cap IHB$.

When a goal `?-G` is given, then the semantics of the program P with respect to this goal is a mapping $\mathscr{M}_{PG}$ from *EHB* to *IHB* defined as follows:

$$\forall E \subseteq EHB \quad \mathscr{M}_{PG}(E) = \{H | H \in \mathscr{M}_P(E) \wedge \Gamma > H\}$$

The semantics of Datalog is based on the choice of a specific model, the least Herbrand model, while first-order logic does not prescribe a particular choice of a model.

Pure Datalog syntax corresponds to a very restricted subset of first-order logic. To enhance its expressiveness, several extensions of pure Datalog have been proposed in the literature. The most important of these extensions are built-in predicates, negation, and complex objects. For instance, the objects handled by pure Datalog programs are tuples of relations made of attribute values. Each attribute value is atomic, so the model is both mathematically simple and easy to implement. On the other hand, more complex contemporary applications require the storage and manipulation of structure objects of higher complexity. Therefore, the relational model has been extended to allow a concise representation of complex structured objects. One of the best known extensions of Datalog is LDL (logic data language) from MCC, Austin, TX (25).

FUNCTIONAL PROGRAMMING LANGUAGES

Historically, the most popular AI language, Lisp, has been classified as a functional programming language in which simple functions are defined and then combined to form more complex functions. A function takes some number of arguments, binds those arguments to some variables, and then evaluates some forms in the context of those bindings.

Functional languages became popular within the AI community because they are much more problem-oriented than conventional languages. Moreover, the jump from formal specification to a functional program is much shorter and easier, so the research in the AI field was much more comfortable.

Functional programming is a style of programming that emphasizes the evaluation of expressions, rather than execution of commands. The expressions in this language are formed by using functions to combine basic values. A functional language is a language that supports and encourages programming in a functional style.

For example, consider the task of calculating the sum of the integers from 1 to 10. In an imperative language such as C, this might be expressed using a simple loop, repeatedly updating the values held in an accumulator variable `total` and a counter variable `i`:

```
total = 0;
for (i=1; i<=10; ++i)
  total += i;
```

In a functional language, the same program would be expressed without any variable updates. For example, in Haskell, nonstrict functional programming language, the result can be calculated by evaluating the expression

```
sum [1..10]
```

Here `[1..10]` is an expression that represents the list of integers from 1 to 10, while `sum` is a function that can be used to calculate the sum of an arbitrary list of values.

The same idea could be used in strict functional languages such as SML or Scheme, but it is more common to find such programs with an explicit loop, often expressed recursively. Nevertheless, there is still no need to update the values of the variables involved:

SML:
```
let fun sum i tot = if i=0 then tot else
      sum (i−1) (tot+i)
in sum 10 0
end
```
Scheme:
```
(define sum
  (lambda (from total)
  (if (= 0 from)
    total
    (sum (− from 1) (+ total from))
(sum 10 0)
```

It is often possible to write functional-style programs in an imperative language, and vice versa. It is then a matter of opinion whether a particular language can be described as functional or not. It is widely agreed that languages such as Haskell and Miranda are "purely functional," while SML and Scheme are not. However, there are some small differences of opinion about the precise technical motivation for this distinction. One definition that has been suggested says that purely functional languages perform all their computations via function application. This is in contrast to languages, such as Scheme and SML, that are predominantly functional, but also allow computational effects caused by expression evaluation that persist after the evaluation is completed. Sometimes, the term "purely functional" is also used in a broader sense to mean languages that might incorporate computational effects, but without altering the notion of "function" (as evidenced by the fact that the essential properties of functions are preserved). Typically, the evaluation of an expression can yield a "task," which is then executed separately to cause computational effects. The evaluation and execution phases are separated in such a way that the evaluation phase does not compromise the standard properties of expressions and functions. The input/output (I/O) mechanism of Haskell, for example, is of this kind.

There is also much debate in the functional programming community about the distinction and the relative merits of strict and nonstrict functional programming languages. In a strict language, the arguments to a function are always evaluated before it is invoked, while in a nonstrict language, the arguments to a function are not evaluated until their values are actually required. It is possible, however, to support a mixture of these two approaches, as in some versions of the functional language Hope.

It is not possible to discuss the mathematical foundation of functional programming without a formal notation for function definition and application. The usual notation that is used in applicative functional languages is so-called λ (lambda) calculus. It is a simple notation and yet powerful enough to model all of the more esoteric features of functional languages. The basic symbols in the λ calculus are the variable names, λ, dot (.), and open and close brackets. The general form for a function definition is

$$\lambda x.M$$

which denotes the function F such that for any value of x, $F(x) = M$, and the value of F can be computed on an argument N by substituting N into the defining equation. A valid λ expression, described in Backus normal form (BNF) notation, is as follows:

```
Expression :: = Variable_name |
             Expression Expression |
             , Variable_name_list .
              Expression |
             ( Expression )
```

The primary relevance of the λ calculus to AI is through the medium of Lisp. Lisp's creator McCarthy used λ calculus as the basis of Lisp's notation for procedures. Since that time other programming languages have used the λ calculus in a more pervasive way. However, from the point of view of AI, the most important among functional languages is definitely Lisp, which will be given more attention in the rest of this section.

Common Lisp

Common Lisp originated in an attempt to focus the work of several implementation groups, which had begun to diverge because of the differences in the implementation environments.

This section provides enough detail for the reader to get a general understanding of the Lisp programming language and to follow basic programming examples.

The History. Lisp (from "list processing") is a family of languages with a long history. Early key ideas in Lisp were developed by John McCarthy during the Dartmouth Summer Research Project on Artificial Intelligence in 1956. Of the major programming languages still in use, only Fortran is older then Lisp. Since then it has grown to be the most commonly used language for AI and expert systems programming. McCarthy's motivation was to develop an algebraic list-processing language for AI work.

Implementation efforts for early dialects of Lisp were undertaken on the IBM 704, the IBM 7090, and the Digital Equipment Corporation (DEC) PDP-1, PDP-6, and PDP-10. The primary dialect of Lisp between 1960 and 1965 was Lisp 1.5. By the early 1970s there were two predominant dialects of Lisp, both arising from these early efforts: MacLisp and Interlisp.

MacLisp (26,27), improved on the Lisp 1.5 notion of special variables and error handling. It also introduced the concept of functions that could take a variable number of arguments, macros, arrays, nonlocal dynamic exits, fast arithmetic, the first good Lisp compiler, and an emphasis on execution speed.

Interlisp (28) introduced many ideas into Lisp programming environments and methodology. One of the Interlisp ideas that influenced Common Lisp was an iteration construct implemented by Warren Teitelman that inspired the `loop` macro used both on the Lisp Machines and in MacLisp, and now in Common Lisp.

The concept of a Lisp machine was developed in the late 1960s. In the early 1970s, Peter Deutsch and Daniel Bobrow implemented a Lisp on the Alto, a single-user minicomputer, using microcode to interpret a byte-code implementation language. Shortly thereafter, Richard Greenblatt began work on a different hardware and instruction set design at MIT. An upward-compatible extension of MacLisp called Lisp Machine Lisp became available on the early MIT Lisp machines. Commercial Lisp machines from Xerox, Lisp Machine Incorporated (LMI), and Symbolics were on the market by 1981.

During the late 1970s, Lisp Machine Lisp began to expand toward a much fuller language. Sophisticated λ lists, sets, multiple values, and structures like those in Common Lisp are the results of early experimentation with programming styles by the Lisp Machine group.

Around 1980, Scott Fahlman and others at CMU began work on a Lisp to run on the Scientific Personal Integrated Computing Environment (SPICE) workstation. One of the goals of the project was to design a simpler dialect than Lisp Machine Lisp.

The Macsyma group at MIT began a project during the late 1970s called the New Implementation of Lisp (NIL) for the VAX. One of the stated goals of the NIL project was to fix many of the historical, but annoying, problems with Lisp. At about the same time, a research group at Stanford University and Lawrence Livermore National Laboratory began the design of a Lisp to run on the S-1 Mark IIA supercomputer. S-1 Lisp, never completely functional, was the test bed for adapting advanced compiler techniques to Lisp implementation.

One of the most important developments in Lisp, occurring during the second half of the 1970s, was Scheme, designed by Gerald J. Sussman and Guy L. Steele, Jr. The major contributions of Scheme were lexical scoping, lexical closures, first-class continuations, and simplified syntax.

In April 1981, after a DARPA-sponsored meeting concerning the splintered Lisp community, Symbolics, the SPICE project, the NIL project, and the S-1 Lisp project joined together to define Common Lisp. Initially spearheaded by White and Gabriel, the driving force behind this grassroots effort was provided by Scott Fahlman and Guy Steele (CMU), Daniel Weinreb and David Moon (Symbolics), Richard Greenblatt (LMI), Jonl White (MIT), and Richard Gabriel (LLNL). Common Lisp was designed as a description of a family of languages. The primary influences on Common Lisp were Lisp Machine Lisp, MacLisp, NIL, S-1 Lisp, and Scheme. It was defined in the book written by Guy Steele (who helped invent Common Lisp and Scheme) *Common LISP: The Language* (1st ed., Maynard, MA: Digital Press, 1984). Its semantics were intentionally underspecified in places where it was felt that a tight specification would overly constrain Common Lisp research and use. The first edition of Steel's book was eagerly adopted by language vendors, third-party developers, and programmers as a de facto Lisp standard.

In 1986 X3J13 was formed as a technical working group to produce a draft for an ANSI Common Lisp standard. Because of the acceptance of Common Lisp, the goals of this group differed from those of the original designers. These new goals included stricter standardization for portability, an object-oriented programming system, a condition system, iteration facilities, and a way to handle large character sets. To accommodate those goals, a new language specification was developed. The specification of the Common Lisp Object System (CLOS) alone took nearly two years and seven of the most talented members of X3J13.

In 1988, the Institute of Electrical and Electronic Engineers (IEEE) Scheme working group was also formed to produce an IEEE standard, and in the year 1990 the group completed its work, producing relatively small and clean standard Scheme. This, however, was only of limited commercial interest. Common Lisp is used internationally and serves as de facto standard.

As G. Steele points out in his book, Common Lisp is intended to meet the following goals:

- *Commonality.* Common Lisp serves as a common dialect to which each implementation makes any necessary extension.
- *Portability.* Common Lisp intentionally excludes features that cannot be implemented easily on a broad class of machines.
- *Consistency.* The definition of Common Lisp avoids inconsistencies by explicitly requiring the interpreter and compiler to impose identical semantics on correct programs as far as possible.
- *Expressiveness.* Common Lisp culls what experience has shown to be the most useful and understandable constructs from not only MacLisp but also Interlisp, other Lisp dialects, and other programming languages.
- *Compatibility.* Unless there is a good reason to the contrary, Common Lisp strives to be compatible with Lisp Machine Lisp, MacLisp, and Interlisp, roughly in that order.
- *Efficiency.* Common Lisp has a number of features designed to facilitate the production of high-quality compiled code in those implementations whose developers care to invest effort in an optimizing compiler.
- *Power.* Common Lisp is a descendant of MacLisp, which has traditionally placed emphasis on providing system-building tools. Although these tools are not part of the Common Lisp core specification, they are expected to be built on top of it.
- *Stability.* It is intended that Common Lisp will change only slowly and with due deliberation. Any extension will be added to Common Lisp only after careful examination and experimentation.

Lisp Basics

John McCarthy, the language's creator, describes the key ideas in Lisp as follows (29):

- Computing with symbolic expressions rather than numbers (that is, bit patterns in a computer's memory and registers can stand for arbitrary symbols, not just those of arithmetic)
- List processing, that is, representing data as linked-list structures in the machine and as multilevel lists on paper
- Control structure based on the composition of functions to form more complex functions
- Recursion as a way to describe processes and problems
- Representation of Lisp programs internally as linked lists and externally as multilevel lists, that is, in the same form as all data are represented.
- The function `EVAL`, written in Lisp itself, serves as an interpreter for Lisp and as a formal definition of language.

One of the major differences between Lisp and conventional programming languages (such as Fortran, Pascal, Ada, C) is that Lisp is a language for symbolic rather than numeric processing. Although it can manipulate numbers as well, its strength lies in being able to manipulate *symbols* that represent arbitrary objects from the domain of interest. Processing pointers to objects and altering data structures comprising other such pointers is the essence of symbolic processing. Symbols, also called *atoms* because of the analogy to the smallest indivisible units, are the most important data types in Lisp. Their main use is as a way of describing programs and data for programs. A symbol is a Lisp object. It has a name associated with it, and a number of aspects or uses. First, it has a value, which can be accessed or altered using exactly the same forms that access or alter the value of a lexical variable. In fact, the methods of naming symbols are the same as those used for naming a lexical variable. In addition to a value, a symbol can have a property list, a package, a print name, and possibly a function definition associated with it. A property list is simply a list of indicators and values used to store properties associated with some objects that the symbol is defined by the programmer to represent. A print name is usually the string of characters that constitutes the identifier. A package is a structure that establishes a mapping between an identifier and a symbol. It is usually a hash table containing symbols. A function is normally associated with a lexical variable or a symbol. The printed representation of a symbol is as a sequence of alphabetic, numeric, pseudoalphabetic, and special characters.

Other typical data types are lists, trees, vectors, arrays, streams, structures, etc. Out of these data structures can be built representations for formulas, real-world objects, natural-language sentences, visual scenes, medical concepts, geographical concepts, and other symbolic data (even other Lisp programs). It is important to note that in Lisp it is data objects that are typed, not variables. Any variable can have any Lisp object as its value.

Historically, list processing was the conceptual core of Lisp, as its name suggests. Lists in Lisp are reprinted in two basic forms. The external, visible, form of a list is composed of an opening parenthesis followed by any number of symbolic expressions followed by a closing parenthesis. A symbolic expression can be a symbol or another list. Internally, a list is represented as a chain of `CONS` cells. The `CONS` cell is the original basic building block for Lisp data structures. Each `CONS` cell is composed of a `CAR` (the upper half, the "data" part) and a `CDR` (the lower half, the "link" part). Lists are represented internally by linking `CONS` cells into chains by using the `CDR` of each cell to point to the `CAR` of the next cell. `CONS` cells can be linked together to form data structures of any desired size or complexity. `NIL` is the Lisp symbol for an empty list, and it is also used to represent the Boolean value "false."

The list-appending function in Lisp is as follows:

```
(DE APPEND (L1 L2)
  (COND (( NULL L1) L2)
    (( ATOM L1) (CONS L1 L2)
    ( TRUE (CONS (CAR L1) (APPEND (CDR L1)
      L2)0000
```

The Lisp function returns a list that is the result of appending `L1` to `L2`. It uses the Lisp function `CONS` to attach one element to the front of a list. It calls itself recursively until all of the elements of `L1` have been attached. The Lisp function `CAR` returns the first element of the list it is given, and the function `CDR` returns the list it is given minus the first element. `ATOM` is true if its argument is a single object rather than a list.

Lisp relies on dynamic allocation of space for data storage. Memory management in Lisp is completely automatic, and the application programmer does not need to worry about assigning storage space. To change or extend a data structure in a Lisp list, for example, one need only change a pointer at a `CONS` cell (see Fig. 1). In Fig. 1 one can see that we can add Mary's name to a list without otherwise changing the program. If we wanted to rearrange a Fortran array to insert Mary's name, we would need to change everything after Mark, which could easily turn out to be a very time-consuming procedure. To make Mary's name occur in several lists,we would simply point to it from each list in which we wanted to include it. Mary and her associated property list, however, would occur in memory only once. Also, a list could contain another list within it, i.e., lists can be nested with arbitrary depth. Elements of lists need not be adjacent in memory—it is all done with pointers. This not only means that Lisp is very modular, it also means that it manages storage space

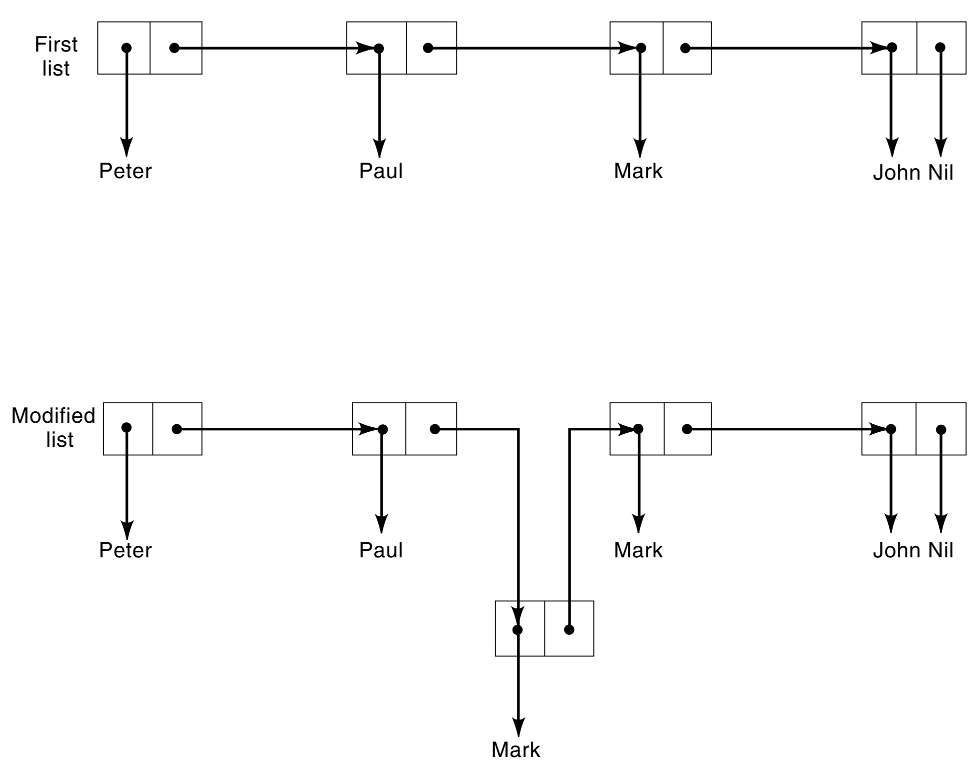

Figure 1. List processing in Lisp.

very efficiently and frees the programmer to create complex and flexible programs.

Multidimensional *arrays,* with general as well as specialized elements, are also defined. An array can have any nonnegative number of dimensions and can have any Lisp object as a component (a general array), or it may be a specialized array, meaning that each element must be of a given, restricted type.

One-dimensional arrays are called *vectors.* General vectors may contain any Lisp object. Vectors whose elements are restricted to character type are called *strings.* Vectors whose elements are restricted to bit type are called *bit vectors.* Strings, like all vectors, may have fill pointers. String operations generally operate only on the active portion (below the fill pointer).

Common Lisp provides for a rich *character set,* including ways to represent characters of various type styles. A variety of string operations is provided.

The *numeric* data types defined in Common Lisp are integers, ratios, floating-point numbers, and complex numbers. Many numeric functions will accept any kind of number (generic functions), while others will accept only specific kinds.

Hash tables provide the means for mapping any Lisp object to an associated object. Each hash table has a set of entries, each of which associates a particular key with a particular value. Finding the value is very fast, even if there are many entries, because hashing is used. This is an important advantage of hash tables over property lists.

Structures are user-defined record structures, objects that have named components. The user can define a new data type and to take advantage of type checking, modularity, and convenience of user-defined record data types. Constructor, access, and assignment constructs are automatically defined when the data type is defined.

A *package* is a data structure that establishes a mapping from print names (strings) to symbols. At any given time one package is current, and the parser recognizes symbols by looking up character sequences in the current package.

Streams represent sources or sinks of data, typically characters or bytes. Character streams produce or absorb characters, while binary streams produce or absorb integers. They are used to perform I/O and for internal purposes (string parsing, for instance). Common Lisp provides a rich set of facilities for performing I/O. All I/O operations are performed on streams of various kinds. A frequent use of streams is to communicate with a file system to which groups of data (files) can be written and from which files can be retrieved.

Pathnames represent names of files, used to interface to the external file system.

Random states are data structures used to encapsulate the state of a built-in random-number generator.

Conditions are objects used to affect control flow in certain conventional ways by means of signals and handlers that intercept those signals.

X3J13 voted in June 1988 to adopt CLOS, thereby introducing new categories of data objects. This object-oriented extension of Lisp, now one of the most important aspects of the ANSI X3J13 standard, has been designed to provide a portable, flexible, and extensible object-oriented programming interface to standard Common Lisp. Many Lisp users regard CLOS as an extremely important feature of any Common Lisp implementation. The most important categories of objects introduced by CLOS are classes, methods, and generic functions.

Classes determine the structure and behavior of other objects (their *instances*). Every Common Lisp data object belongs to some class.

Methods are chunks of code that operate on arguments satisfying a particular pattern of classes. Methods are not functions; they are not invoked directly on arguments, but instead are bundled into generic functions.

Generic functions are functions that contain, among other information, a set of methods. When invoked, a generic function executes a subset of its methods. The subset chosen for execution depends in a specific way on the classes or identities of the arguments to which it is applied.

Most Lisp programs are in the form of functions or collections of functions, which return values, and programming in Lisp is very much like functional composition. Historically, Lisp has been classified as a functional programming language in which simple functions are defined and then combined to form more complex functions. A function takes some number of arguments, binds those arguments to some variables, and then evaluates some forms in the context of those bindings.

The *evaluator* is the mechanism that executes Lisp programs. The evaluator accepts a form and performs the computation specified by the form. This mechanism is made available to the user through the function `EVAL`. The evaluator is typically implemented as an interpreter that traverses the given form recursively, performing each step of the computation as it goes. An alternative approach is for the evaluator first to completely compile the form into machine-executable code and then invoke the resulting code. Various mixed strategies are also possible. However, the implementors should document the evaluation strategy for each implementation.

After the evaluation takes place, a value or values are returned. The argument-passing convention is *call by value*—every argument to a function is first evaluated, and then Lisp pointers to those values are passed to the function. Obviously, the values that are passed are de facto pointers to values, so that if the value is a complex data structure, the data structure is not copied, but a pointer to it is passed.

The concept of evaluation, i.e., the Lisp terminology for executing a procedure and returning the result, is important to understanding the nature of Lisp. The values of simple expressions are computed, and those values are passed on to other functions, which use them to evaluate further values. An *expression* is either a constant, a variable, a symbol, a combination, or a special form. A *constant* is a number, a string, or a quoted object. The value of a constant is a constant itself. The value of a symbol is the contents of the value cell of the symbol, and the value of a variable is obtained from its associated location. The value returned by a function is simply a transient result that is presented back to the higher-level function. A *combination* can be a function invocation, a macro invocation, or a special form. Special forms look like combinations.

Conventional programming languages normally consists of sequential statements and associated subroutines. Lisp consists of a group of modules, each of which specializes in performing a particular task. This makes it easy for programmers to subdivide their efforts into numerous modules, each of which can be handled independently. Also, it is easily possible to see in Lisp's CONS cells the germs of the if–then rules that are so popular in expert systems. Likewise, it is possible to see the beginnings of frames in the property lists that are attached to atoms (objects).

Good Environments. Today's Lisp environment support windowing, fancy editing (where the text editor can also be accessed by means of Lisp functions), good debugging (including text-based stepper, trace facility, inspector, and break-loop debuggers), graphically examining data structures, and even automatic testing, automatic cross-referencing, and so on (30). As Richard Gabriel states in his 1991 article (31), a lot of modern development environment features originated from the Lisp world:

- Incremental compilation and loading
- Symbolic debuggers
- Source-code-level single stepping
- Help on built-in operators
- Window-based debugging
- Symbolic stack backtraces
- Structure editors

In the same article, Gabriel pointed out that Lisp provides a means for multiparadigm programming owing to its ability to coexist with other languages (C, Pascal, Fortran, etc.). These languages can be invoked from Lisp and in general can reinvoke Lisp. Such interfaces allow the programmer to pass Lisp data to foreign code, pass foreign data to Lisp code, manipulate foreign data from Lisp code, manipulate Lisp data from foreign code, dynamically load foreign programs, and freely mix foreign and Lisp functions. Therefore, if the user wants to program low-level functions in C but write the program logic in Lisp, it is easily accomplished. However, it is not necessary to go to C++ or to Smalltalk to switch to the new object-oriented paradigm, since it is fully supported by the CLOS extension.

Lisp and Artificial Intelligence. Lisp is a very good choice for an AI project programming for several reasons. Most AI projects involve manipulation of symbolic rather than numeric data, and Lisp provides primitives for manipulating symbols and collections of symbols. Lisp also provides automatic memory management facilities, eliminating the need to write and debug routines to allocate and reclaim data structures. Lisp is extensible and contains a powerful macro facility that allows layers of abstraction.

Lisp's lists can be of any size and contain objects of any data types (including other lists), so that programmers can create very complex data structures for representing abstract concepts such as object hierarchies, natural language parse trees, expert-system rules, etc. A collection of facts about an individual object can easily be represented in the property list that is associated with the symbol representing the concept. The property list is simply a list of attribute–value pairs. The fact that both data and procedures are represented as lists makes it possible to integrate declarative and procedural knowledge into a single data structure such as a property list.

Although symbols and lists are central to many AI programs, other data structures such as arrays and strings are also often necessary.

The most natural Lisp control structure is recursion, which often represents the most appropriate control strategy for many problem-solving tasks.

Moreover, Lisp possesses the ability to treat its own code as data. Researchers have exploited this capability of Lisp programs to modify themselves at run time for research in machine learning, natural language understanding, and other aspects of AI. Lisp implementation also encourages an interactive style of development ideally suited to exploring solutions for difficult or poorly specified problems. This is of crucial importance in AI application areas where the problems are too hard to be solved without human intervention.

Perhaps the most successful AI application in the business world is expert-system technology. Lisp is tailored to expert-system creation because the language is rich, with its flexible list data types and excellent support for recursion; because it is extensible; and because it has facilities for rapid prototyping, which lets the implementor experiment with design and customize the expert system. Programmers can use the built-in list data type for easy creation of the data structures necessary to represent parameters, rules, premise clauses, conclusion actions, and other objects that constitute the knowledge base. Even the expert systems that could process a wealth of expertise about such esoteric disciplines as chemistry, biology, avionics, and handle complex and rapidly changing process in real time have been built in Lisp. A Lisp-based expert-system shell G2, manufactured by Gensym of Cambridge, MA, has been used all around the world for building real time expert systems. Even the most complex applications, such as Space Shuttle fault diagnosis or launch operation support, have been Lisp-based (32).

Also, representing knowledge using *frames* conforms nicely with object-oriented programming ideas, so thinking about representing frames in CLOS is only natural. Using CLOS as a foundation for knowledge representation provides a layer of system support that the implementer of frame-based systems can use effectively (33).

CLOS supports features such as classes, call hierarchies, slots, and instances, as well as generic functions and methods. A *generic function* is a function that can have instances of different classes as an argument (called overloading). The code that runs is dependent on the classes of the arguments; each piece of code is called a *method.* Since CLOS is designed as an object-oriented programming language and not as a frame-based knowledge representation language, it does not directly support every feature of frame-based systems.

However, CLOS can be customized straightforwardly using the *metaobject protocol* (33) to support most of the frame-based system features, including different inheritance schemes, multiple-valued slots, facets (ability to represent information about the value in the slot), demons (functions that run whenever a slot is accessed), etc. The basis of the metaobject protocol is an object-oriented implementation of CLOS itself that makes CLOS adjustable using object-oriented programming. CLOS classes are themselves implemented as CLOS objects, and hence each CLOS class is an instance of a class. Each such object is called a *class metaobject,* to distinguish it from other CLOS objects. These objects have one or more classes, which are called *class metaobject classes.* Each of these can be thought of as the class of a set of objects, each of which describes a class. Because CLOS itself is an object-oriented program, a class hierarchy and a set of methods are applicable to class metaobject classes. This class hierarchy, the names of the methods, and what the methods do provide a framework for the behavior of CLOS and constitute the metaobject protocol, which amounts to a customization protocol for users. The metaobject protocol makes it possible to customize inheritance (both for slots and methods), initialize classes and instances, allocate new instances, and add new classes dynamically. These tasks are achieved by specializing members of the set of generic functions defined by the metaobject protocol and by subclassing from the set of classes defined by the protocol.

To summarize, the metaobject protocol allows its users to develop their own object-oriented paradigms. A lot of AI developers already use the metaobject protocol. Artificial Intelligence Technologies Corporation sells a product based on CLOS extensions called Mercury KBE. Other companies such as Systems, which developed a frame-based knowledge representation system called CLOS-XT, have developed in-house expert systems using the metaobject protocol.

It is important to remember that Lisp can be an exploratory language as well as a product-producing one. Lisp is a marvelous research language that gives a programmer the ability to create and experiment without paying attention to the data types of variables or the way memory is allocated. Unfortunately, this latitude incurs a penalty. Because Lisp is an extremely high-level language, it is fairly well insulated from the machine it runs on and thus tends to run rather slowly. In addition, the ability to create recursive lists of heterogeneous objects dramatically increases pointer overhead. However, the user can try new AI ideas in Lisp and then to convert to other language. Some cross-compilers will even automatically convert Lisp into C or a similar language.

Lisp has strengths and weaknesses. It has had some real successes but also some real problems that still have to be solved (31). Nevertheless, it should be part of the toolkit of any professional AI programmer, particularly of those who routinely construct very large and complex expert systems.

HYBRID LANGUAGES

Various knowledge representation paradigms have cross-fertilized each other. Currently popular are commercial and other products known as AI toolkits [Loops, ART (Inference Corporation), KEE (IntelliCorp Inc.), BEST, OPS5 (Production Systems Technology), Nexpert Object, EXSYS, etc.]. Typically, this type of product can be considered as a semantic net with one kind of link, representing something like "subclass IS-A class" or "subclass IS-A-KIND-OF class," with the concepts at the nodes having features of frames (see Fig. 2) and some system for production rules or logical representation of rules added on top to enable a wider variety of inferences to be performed. The Loops (34) knowledge programming environment integrates function-oriented, object-oriented, rule-oriented, and access-oriented programming. Another rich amalgamation of multiple programming paradigms is KEE (35), which integrates object-oriented and rule-oriented programming with a database management system. ART (36) successfully combines rules and frames (schemata) and provides the means for hypothetical and time-state reasoning.

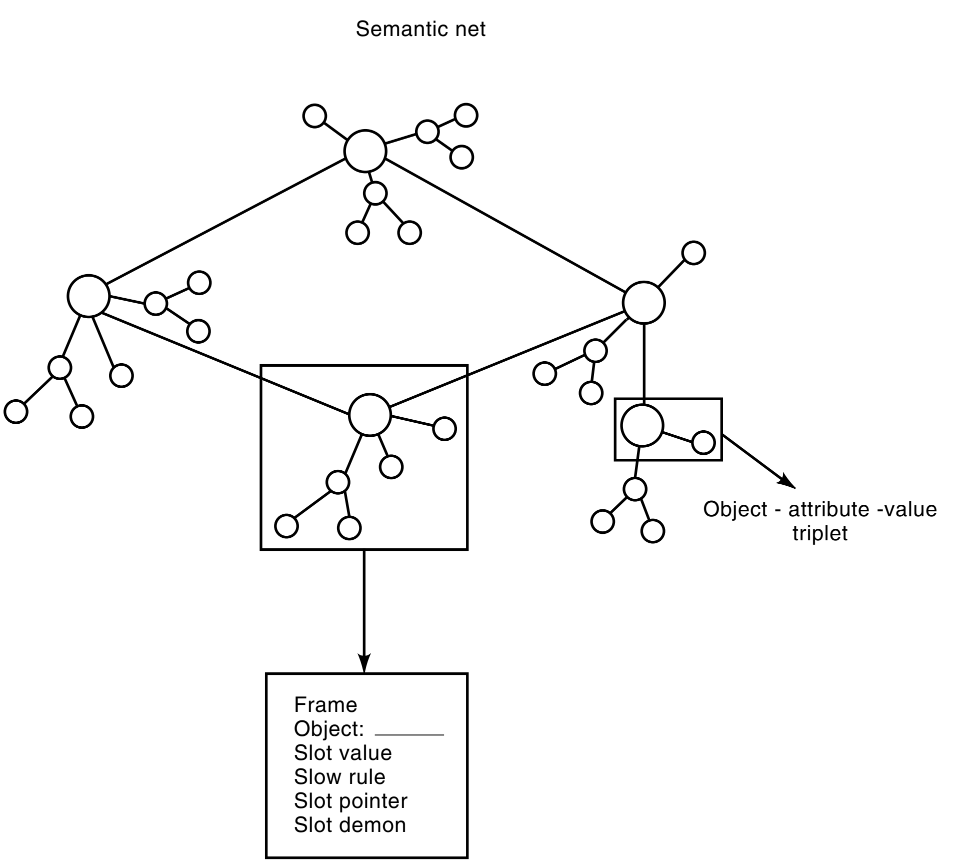

Figure 2. Hybrid knowledge representation.

Within the basic, knowledge-based programming paradigm, BEST (37) offers a multiparadigm language for representing complex knowledge, including incomplete and uncertain knowledge. Its problem-solving facilities include truth maintenance, inheritance over arbitrary relations, temporal and hypothetical reasoning, opportunistic control, automatic partitioning and scheduling, and both blackboard and distributed problem-solving paradigms.

All the above-mentioned toolkits have been fairly successful in building a variety of applications. However, although they are commonly used AI tools and allow many programming-like constructs, strictly speaking, they probably should not count as AI programming languages. We will discuss here only true hybrid programming languages, particularly those that resulted from the cross-fertilization between the two programming paradigms presented here (logic programming and functional programming) and the languages built on top of the two most popular AI languages discussed in this entry (Prolog and Lisp).

A programming paradigm can be thought of as a basis for a class of programming languages, as an underlying computational model, as a primitive set of execution facilities, or as a powerful way of thinking about a computer system. The purpose of multiparadigm programming is to let us build a system using as many paradigms as we need, each paradigm handling those aspects of the system for which it is best suited. Researchers in the AI field have recently been taking an increased interest in multiparadigm representation and reasoning systems. The first-generation AI systems were mostly monolithic, isolated, and standalone, which prevented them from adequately addressing the complexity, diversity, and performance challenges of complex, heterogeneous, large-scale applications. However, most real-world problems and situations are sufficiently complex to demand more than one reasoning technique for solution, each technique attacking one characteristic of the problem domain. There is also a need to integrate AI solution techniques with conventional techniques. AI researchers have for most of these early years been guilty of ignoring a significant amount of research done in traditional fields such as decision sciences and operations research.

Furthermore, just as there is no one omnipotent general reasoning technique, there is little consensus on a general architecture for integrating diverse reasoning techniques, so that should be among the main research topics within the AI community in the future. A special part of the Fifth International Symposium on AI held in Cancun, Mexico, was Marvin Minsky's keynote address on "The Near Future of AI," which set the tone of the conference in encouraging AI researchers to use multiparadigm approaches to solving problems. Minsky stressed that during the next decade, AI researchers need to move toward a hybrid approach to handling knowledge representation and solving problems.

Though a number of multiparadigm AI systems have been designed and studied, little effort has been devoted to comparing the systems or searching for common principles underlying them. A characterization of the multiparadigm AI systems along two dimensions has been suggested (38)—systems that employ multiple representations and those with multiple reasoning paradigms. The same knowledge can be represented in different media, as in the VIVID reasoning systems, presented by Etherington et al. (39), and the multiple reasoners in a single layer of the CAKE architecture (40); or it can have different representations for different kinds of knowledge, as in KRYPTON (41) and many-sorted logic (42–44).

An interesting example of a language with multiple paradigms in the object framework is Orient/84 (45), which has been designed to describe both knowledge systems and systems of more general application. It has the metaclass–class–instance hierarchy and multiple inheritance from multiple superclasses. A knowledge object consists of a behavior part, a knowledge base part, and a monitor part.

The behavior part is a collection of procedures (or methods) that describe actions and attributes of the object in Smalltalk-like syntax and semantics. The knowledge base part is the local knowledge base of the object, containing rules and facts. The Prolog-like predicate logic is employed to describe the knowledge base. The monitor part is the guardian and demon for the object and is described in a declarative manner.

Along the reasoning dimension, a system can have different reasoners for the same representations, as in KRYPTON, KL-TWO (46), theory resolution (47) and many-sorted logic; or the same reasoner for different kinds of knowledge, as in the Patel-Schneider's hybrid logic (48). RAL (rule-extended algorithmic language) from Production Systems Technology (developer of OPS5 and OPS83) adds rule-based and object-oriented capabilities to C programs.

The Loom knowledge representation language, developed by Robert McGregor and John Jen at the University of Southern California's Information Sciences Institute, combines object-oriented programming, rules, and logic programming in a set of tools for constructing and debugging declarative models (49). Loom uses a description classifier to enhance knowledge representation and to extend the class of useful inference beyond simple inheritance (found in most frame systems). It supports the description language (the frame component) and a rule language, and uses its classifier to bridge the gap between the two. The classifier gives Loom the additional deductive power to provide inference capabilities not often found in current knowledge-representation tools.

Two modern paradigms—logical programming and functional programming—are also an active area of research in multiparadigm languages and environments. Their great similarities include applicative nature, reliance on recursion for program modularity, and providing execution parallelism in a natural manner. Their differences include radically different variable concepts, availability of higher-order program entities, and fundamental support for nondeterministic execution. A successful combination would be the notational leverage and execution directness of functional programming with search guided through constraint accumulation. Many proposals have been offered on how to combine these two paradigms. One of the most interesting proposals has come from the University of Utah (50,51).

Tablog [IBM Research, Stanford University, Weizmann Institute (Israel), and SRI International] is a logic programming language that combines functional and relational programming into a unified framework. It incorporates advantages of two of the leading programming languages for symbolic manipulation—Prolog and Lisp—by including both relations and functions and adding the power of unification and binding mechanisms.

RELFUN (University of Aachen, Germany) is a relational–functional programming language with call-by-value expressions for nondeterministic, nonground functions. Its clauses define operations (relations and functions) permitting (apply-reducible) higher-order syntax with arbitrary terms (constants, structures, and variables) as operators. The language explicitly distinguishes structures (passive) and expressions (active). All structures and expressions, not only lists and tuples, can have varying arities. Mercury is another new logic–functional programming language, which combines the clarity and expressiveness of declarative programming with advanced static analysis and error detection features. ALF is yet another logic–functional language, and there are a lot more of them in development. LIFE (logic, inheritance, functions, and equations) is an experimental programming language that integrates three orthogonal programming paradigms: logic programming, functional programming, and object-oriented programming.

The programming language Nial (52) supports several styles of programming including imperative, procedural, applicative, and λ-free. Nial tools can be built to illustrate relational and object-oriented styles of programming.

Prolog-Based Hybrid Languages

Although the logic programming language Prolog, with its coherent declarative and procedural semantics, seems to be a very good candidate for AI applications (23,53), predicate calculus has been widely criticized for its lack of certain facilities that are important in knowledge representation and processing, such as:

- the expressive power needed to represent complex knowledge
- the facility to modularize a knowledge base effectively
- the facility to construct a hierarchy of *concepts*
- the facility to control inheritance of properties through a concept hierarchy
- the facility to pursue alternative pathways to a goal (hypothetical reasoning)
- a time-state reasoning facility
- the facility to deal with incomplete knowledge
- the facility to customize the inference control strategy
- the possibility to express the external control

Over the past fifteen years, great efforts have been invested in the study of techniques in overcoming the mentioned representational (54) and inferential (55–57) inadequacies and drawbacks of standard Prolog as a tool for knowledge-based system development. Prolog/Rex (58) integrates these efforts and experience into a single uniform and flexible hybrid environment for knowledge-based system development, which further provides the user with hypothetical and time-state reasoning, truth maintenance, and uncertainty management facilities. Prolog/Rex aims at realizing both hybrid knowledge representation and hybrid control of knowledge, whereas other similar systems realize either hybrid knowledge representation or hybrid control mechanisms. In KORE (54), for instance, different knowledge control paradigms are distributed to subsystems, each of which provides a unique paradigm for controlling knowledge. Prolog/Rex (58), on the other hand, provides the deep and soft integration of different knowledge representation and processing paradigms, not only a collection of single paradigm subsystems. Rules use frames extensively as a working memory, whereas frames can contain behavioral information in the form of rules as well as procedures. In addition to this, forward and backward inference paradigms can be used alternately during the same reasoning cycle. This conceptual integration of the different paradigms lets the user bypass the major gaps in pure rule-based representation languages—their inadequate description of entities and the static relationships among them. KORE's subsystem KORE/IE (59) fails to provide any kind of frame language, uncertainty management, hypothetical time-state reasoning, or even backward-chaining reasoning.

In the FROG system (60), which is similar to Prolog/Rex, a hybrid knowledge representation formalism and a modest amount of control flexibility (in the frame system only) are provided. The different representation paradigms (frames, rules, procedures) are not fully integrated (rules and procedures can be called from the frames only). While FROG's frame system implements only standard inheritance relations, Prolog/Rex offers the facility to create any kind of relations among domain entities. All FROG's flexibility is due to its Prolog-based design. In particular, different control strategies can be obtained by simply combining the basic components of the knowledge base (expressed as Prolog predicates) in those Prolog clauses that constitute a high-level description.

Among the five main knowledge representation approaches (semantic networks, O–A–V triplets, rules, frames, predicate calculus), Prolog/Rex attempts to combine the last three approaches, through the common basis of Prolog. A great deal of success has been achieved by integrating frames (Prolog/Rex concepts) and production rules to form hybrid representation facilities that combine the advantages of both component representation techniques, as the major inadequacies of production rules are in areas that are effectively handled by frames and vice versa (61). Moreover, since Prolog/Rex allows the user to define arbitrary relations among concepts (not only standard inheritance relations) so that they can be connected into the semantic network, while the concept confined to one slot represents nothing but an O–A–V triplet, we can also claim the incorporation of the former two paradigms. This multiple representation language helps its user to address a range of required applications, since no single formalism is sufficient for effectively representing the diversity of knowledge required in real-life applications. This also helps with the efficiency goal, since an application can be constructed using an appropriate combination of representations, and it does not have to be distorted to fit a single representation approach.

Lisp-Based Hybrid Languages

KRL (62) is a hybrid language, built on top of Interlisp, that facilitates the representation of knowledge in frame structures (slot-and-filler structures). Its design was motivated by the following assumptions about knowledge representation (p. 5 of Ref. 62):

- Knowledge should be organized around conceptual entities with associated description and procedures.

- A description must be able to represent partial knowledge about an entity and accommodate multiple descriptors that can describe the associated entity from different viewpoints.
- An important method of description is comparison with a known entity, with further specification of the described instance with respect to the prototype.
- Reasoning is dominated by a process of recognition in which new objects and events are compared with stored sets of prototypes, and in which specialized reasoning strategies are keyed to these prototypes.
- Intelligent programs will require multiple active processes with explicit user-provided scheduling and resource allocation heuristics.
- Information should be clustered to reflect use in processes whose results are affected by resource limitation and differences in information accessibility.
- A knowledge representation language must provide a flexible set of underlying tools, rather than embody specific commitments about either processing strategies or the representation of specific areas of knowledge.

Each object (entity) in KRL is represented by means of data abstraction called a UNIT. Some UNITS represent abstract concepts, or classes of objects, while others represent specific instances of those concepts. A single UNIT can describe an object from several viewpoints. The SELF relation corresponds to the IS-A and INSTANCE-OF relations. Slot fillers in concept UNITS describe restrictions on the values that may fill the slot for an instance of the concept, or they may represent a default value to the slot. Slot fillers in UNITS representing specific objects are the actual values for the particular object.

Frame-based languages such as KRL (62), FRL (63), and KL-One (64) have mechanisms to construct hierarchies of domain objects and to retrieve information in the hierarchy. However, complex reasoning based on that information is not supported by these languages. Most of them have an escape to other languages (usually Lisp) for procedural attachment. This makes the meaning of data structures unclear, since it depends on the behavior of these procedures for handling them. The situation is worse in traditional semantic networks, which provide no procedural semantics at all.

Another Lisp-based hybrid language is PLANNER (65), designed for representing both traditional, data-driven (forward) reasoning and goal-driven (backward) reasoning, encourages the encoding of procedural knowledge, and offers utilities for using that kind of knowledge in a particular style of problem solving.

In PLANNER, the programmer expresses his program in terms of collection of statements. There are two types of statements:

- *Assertions,* which simply state known facts
- *Theorems,* which describe how to achieve a goal given certain preconditions and what to do should certain situations arise in the process. There are special theorems used for goal-directed reasoning (*consequent theorems*), for data-directed reasoning (*antecedent theorems*), and for deleting assertions from the database (*erasing theorems*).

One of the main difficulties that arose with PLANNER is its rigid backtracking strategy, which was automatic, rather than being under the control of the programmer. In reaction to this rigidity, a new language, CONNIVER, was developed (66). It retained many of the ideas of PLANNER, but at a lower level, so that fewer of the mechanisms were imposed on the user. However, the user could explicitly direct the control flow of a CONNIVER program.

BIBLIOGRAPHY

1. A. Barr and E. Feigenbaum, *The Handbook of Artificial Intelligence,* London: Pitman, 1983.
2. E. Rich, *Artificial Intelligence,* New York: McGraw-Hill, 1983.
3. M. Minsky, A framework for representing knowledge, in P. Winston (ed.), *The Psychology of Computer Vision,* New York: McGraw-Hill, 1975.
4. A. Newell, Production systems: Models of control structures, in W. G. Chase (ed.), *Visual Information Processing,* New York: Academic Press, 1973.
5. B. Kowalski and L. Stipp, Object processing for knowledge based systems, *AI Expert,* **5** (10): 34–42, 1990.
6. J. Herbrands, Research in the theory of demonstration, in J. van Heijenoort (ed.), *From Frege to Gödel: A Source Book in Mathematical Logic,* Cambride, MA: Harvard University Press, 1967.
7. D. Prawitz, An improved proof procedure, *Theoria,* **26**: 102–139, 1960.
8. P. C. Gilmore, A proof method for quantification theory, *IBM J. Res. Develop.,* **4**: 28–35, 1990.
9. M. Davis and H. Putnam, A computing for quantification theory, *J. Assoc. Comput. Mach.,* **7**: 201–215, 1960.
10. J. A. Robinson, A machine-oriented logic based on the resolution principle, *J. Assoc. Comput. Mach.,* **12** (1): 23–41, 1965.
11. R. A. Kowalski and D. Kuehner, Linear resolution with selection function, *Artif. Intell.,* **2**: 227–260, 1971.
12. R. A. Kowalski, *Logic for Problem Solving,* New York: Elsevier North Holland, 1979.
13. A. Colmerauer et al., *Un Système de Communication Homme–Machine en Français,* Groupe de Recherche en Intelligence Artificielle, Universite d'Aix-Marseille, 1973.
14. C. Green, Applications of theorem proving to problem solving, in *Proc. IJCAI '69 Conf.,* 1969, pp. 219–239.
15. P. J. Hayes, Computation and deduction, in *Proc. MFCS Conf.,* 1973.
16. Amihai Motro, Intensional answers to database queries, *IEEE Trans. Knowl. Data Eng.,* **6**: 444–454, 1994.
17. T. Imieliski, Intelligent query answering in rule-based systems, *J. Logic Programming,* **4** (3): 229–257, 1987.
18. L. Cholvy and R. Demolombe, Querying a rule base, in *Proc. 1st Int. Workshop Expert Database Syst.,* 1984, pp. 326–341.
19. A. Pirotte and D. Roelants, Constraints for improving the generation of intensional answers in deductive database, in *Proc. IEEE Comput. Soc. 5th Int. Conf. Data Eng.,* 1989, pp. 652–659.
20. J. Han, Chain-split evaluation in deductive databases, *IEEE Trans. Knowl. Data Eng.,* **7**: 261–274, 1995.
21. S. H. Lee and L. J. Henschen, Evaluation of recursive queries with extended rules in deductive databases, *IEEE Trans. Knowl. Data Eng.,* **7**: 328–332, 1995.
22. W. F. Clocksin and C. S. Mellish, *Programming in Prolog,* New York: Springer-Verlag, 1984.
23. I. Bratko, *Prolog Programming for Artificial Intelligence,* Reading, MA: Addison-Wesley, 1986.

24. S. Ceri, G. Gottlog, and L. Tanca, What you always wanted to know about Datalog (and never dared to ask), *IEEE Trans. Knowl. Data Eng.,* **1**: 146–166, 1989.

25. S. Tsur and C. Zaniolo, LDL: A logic-based query language, in *Proc. 12th Int. Conf. Very Large Data Bases,* 1986.

26. D. Moon, *MacLISP Reference Manual,* Cambridge, MA: MIT Project MAC, 1974.

27. K. M. Pitman, *The Revised MacLISP Manual,* MIT/LCS/TR 295, Cambridge, MA: MIT Laboratory for Computer Science, 1983.

28. W. Teitelman, *InterLISP Reference Manual,* Palo Alto, CA: Xerox Palo Alto Research Center, 1978.

29. J. McCarthy, History of Lisp, in D. Wexelblat (ed.), *History of Programming Languages,* New York: Academic Press, 1978.

30. M. Brinsmead et al., Common Lisp product roundup, *AI Expert,* **6** (6): 48–57, 1991.

31. R. Gabriel, LISP: Good news, bad news, how to win big, *AI Expert,* **6** (6): 31–40, 1991.

32. J. Keyes, LISP: The great contender, *AI Expert,* **7** (1): 24–28, 1992.

33. Jim Veitch, Frames in CLOS, *AI Expert,* **6** (6): 31–40, 1991.

34. M. J. Stefik, D. G. Bobrow, and K. M. Kahn, Integrating access-oriented programming into a multiparadigm environment, *IEEE Software,* **3** (1): 10–18, 1986.

35. IntelliCorp Inc., *KEE Software Development System User's Manual,* Mountain View, CA, 1990.

36. Inference Corporation, *ART-IM (Automated Reasoning Tool for Information Management) User's Manual,* Los Angeles, CA, 1991.

37. S. Vranes and M. Stanojevic, Integrating multiple paradigms within the blackboard framework, *IEEE Trans. Softw. Eng.,* **21**: 244–262, 1995.

38. A. Frisch and A. Cohn, 1988 Workshop on Principles of Hybrid Reasoning, *AI Mag.,* **11** (5): 77–84, 1991.

39. D. W. Etherington et al., Vivid knowledge and tractable reasoning: Preliminary report, in *Proc. 11th Int. Conf. Artificial Intell.,* 1989, pp. 1146–1152.

40. C. Rich, The layered architecture of a system for reasoning, in *Proc. 6th National Conf. Artificial Intell.,* 1987, pp. 48–52.

41. R. J. Brachman, R. E. Fikes, and H. J. Levesque, KRYPTON: A functional approach to knowledge representation, *IEEE Computer,* **16** (10): 67–73, 1983.

42. A. G. Cohn, A more expressive formulation of many sorted logic, *J. Automated Reason.,* **3** (2): 113–200, 1987.

43. A. M. Frisch, A general framework for sorted deduction: Fundamental results on hybrid reasoning, in R. J. Brachman, H. J. Levesque, and R. Reiter (eds.), *Proc. 1st Int. Conf. Principles Knowl. Representation Reasoning,* Toronto, 1989.

44. C. Walther, *A Many-Sorted Calculus Based on Resolution and Paramodulation,* Los Altos, CA: Morgan Kaufman, 1987.

45. M. Tokoro and Y. Ishikawa, Orient/84: A language with multiple paradigms in the object framework, in *Proc. Hawaii Int. Conf. Syst. Sci.,* 1986.

46. M. Vilain, The restricted language architecture of a hybrid representation system, in *Proc. Int. Joint Conf. Artificial Intell.,* 1985, pp. 547–551.

47. M. E. Stickel, Automated deduction by theory resolution, in *Proc. 9th Int. Joint Conf. Artificial Intell.,* 1985, pp. 455–458.

48. P. Patel-Schneider, Decidable first-order logic for knowledge representation, Ph.D. Thesis, University of Toronto, 1987.

49. R. McGregor and M. H. Burstein, Using a description classifier to enhance knowledge representation, *IEEE Expert,* **6** (3): 41–46, 1991.

50. G. Lindstrom and P. Panangaden, Stream-based execution of logic programs, in *Proc. 1984 Int. Symp. Logic Programming,* 1984, pp. 168–176.

51. G. Lindstrom, Functional programming and the logical variable, in *Proc. Symp. Principles Programming Languages,* 1985, pp. 266–280.

52. M. A. Jenkins, J. I. Glasgow, and C. D. McCrosky, Programming styles in Nial, *IEEE Software,* **3** (1): 46–55, 1986.

53. K. Weiskamp and T. Hengl, *Artificial Intelligence Programming with Turbo Prolog,* New York: Wiley, 1988.

54. T. Shintani et al., KORE: A Hybrid Knowledge Programming Environment for Decision Support Based on a Logic Programming Language, in *Proc. 5th Conf. Symp. Logic Programming,* 1986.

55. P. Devanbu, M. Freeland, and S. Naqvi, A procedural approach to search control in Prolog, in *Proc. Int. Conf. ECAI'86,* 1986.

56. M. Dincbas and J.-P. La Pape, Metacontrol of Logic Programs in METALOG, in *Proc. Int. Conf. 5th Generation Comput. Syst. 1984,* 1984.

57. H. Gallaire, J. Minker, and J. Nicolas (eds.), *Advances in Database Theory,* Vol. 1, New York: Plenum Press, 1981.

58. S. Vranes and M. Stanojevic, Prolog/Rex—a way to extend Prolog for better knowledge representation, *IEEE Trans. Knowl. Data Eng.,* **6**: 22–37, 1994.

59. T. Shintani, A fast Prolog based production system KORE/IE, in *Proc. 5th Conf. Symp. Logic Programming,* 1986.

60. L. Console and G. Rossi, Using Prolog for building FROG, a hybrid knowledge representation system, *New Generation Comput.,* **6**: 361–388, 1989.

61. J. S. Aikins, A representation scheme using both frames and rules, in B. G. Buchanan and E. Shortlife (eds.), *Rule Based Expert Systems,* Reading, MA: Addison-Wesley, 1984.

62. D. G. Bobrow and T. Winograd, An overview of KRL-0, a knowledge representation language, *Cognitive Sci.,* **1**, 1977.

63. R. B. Roberts and I. P. Goldstein, *The FRL manual,* MIT AI-Memo 409, Cambridge, MA: MIT, 1977.

64. R. J. Brachman and J. G. Schmolze, An overview of the KL-ONE knowledge representation system, *Cognitive Sci.,* **9**: 171–216.

65. C. Hewitt, Description and theoretical analysis (using schemas) of PLANNER: A language for proving theorems and manipulating models in a robot, Doctoral Dissertation, AI Laboratory, Massachusetts Institute of Technology, 1971.

66. G. J. Sussman and D. McDermott, *Why conniving is better than planning,* Technical Report, AI Memo 2655A, Cambridge, MA: Massachusetts Institute of Technology, 1972.

SANJA J. VRANES
The Mihailo Pupin Institute

AIR-CORE MAGNETS. See SUPERCONDUCTING ELECTROMAGNETS.

AIRCRAFT CONTROL. See ATTITUDE CONTROL.

AIRCRAFT ENGINES. See JET ENGINE CONTROL, IMPLEMENTATIONS.

AIRCRAFT, JET TRANSPORT MAINTENANCE. See JET TRANSPORT MAINTENANCE.