RULE-BASED PROGRAMMING IN THE *UNIX*® SYSTEM

Gregg T. Vesonder

Gregg T. Vesonder *is in the Artificial Intelligence (AI) Systems Department at AT&T Bell Laboratories in Liberty Corners, New Jersey. He is technical supervisor of the AI technology group that is responsible for AI tool development, applied research in AI, and expert system prototypes. Mr. Vesonder, who joined the company in 1979, has a B.A. in psychology from the University of Notre Dame, and both an M.S. and a Ph.D. in cognitive psychology from the University of Pittsburgh.*

Rule-based tools are widely used for building expert systems, and rule-based programming is gaining acceptance as a general method. Our experience with building expert systems suggests that rule-based tools are most effectively used for developing expert systems when they are integrated with a general-purpose operating system such as the UNIX® system. This paper introduces the concepts of rule-based programming, discusses strengths and weaknesses of the method, and describes our current rule-based programming environment within the UNIX system.

Background

An expert system is a computer program that embodies the knowledge an expert uses when doing a task that requires reasoning and, using that knowledge, performs the task as well or nearly as well as the expert. Many expert systems have been built using rule-based tools (e.g., see reference 1). Rule-based tools are popular for building expert systems because expertise stated in the form—"*If* this is the situation, *then* take this action"—is readily coded into the IF-THEN format of rules. In 1981, we began using rule-based tools and the UNIX® system as a basis for developing the ACE expert system.[2,3] (Panel 1 defines acronyms used in this paper.) Since that time, the development environment that we have used for building expert systems within the UNIX system has evolved.

This paper introduces the concepts of rule-based programming, provides reasons why this method is appropriate for building expert systems (and maybe other programming tasks) and describes our current environment for building rule-based systems within the UNIX system.

Rule-Based Programming

Today, most expert systems are built using a rule-based approach. A rule-based methodology is useful for two reasons:

- It enables the developer to represent units of knowledge in chunks called rules.

- Because these chunks are independent, developers can easily change the knowledge that a rule represents without disturbing the rest of the system.

Several rule-based systems are available for the developer to use. One of the most popular is OPS5, a rule-based system that originated at Carnegie-Mellon University. (OPS stands for official production system.) OPS5 has become a de facto standard for a variety of knowledge-engineering groups who use rules as a representation method for building knowledge bases. (Knowledge engineers collect knowledge from experts, organize it, and develop the systems that mimic the expert in applying the knowledge.)

Before we examine the advantages and disadvantages of using a rule-based system and describe the OPS5 language, it will be useful to describe rule-based systems in more general terms.

Panel 1. Acronyms in This Paper

ACE	automated cable expertise
AI	artificial intelligence
BASIC	beginner's all-purpose symbolic instruction code
BLISS	a programming language
CL5	port of OPS5 to Common Lisp
EMYCIN	inference engine for MYCIN, expert system for diagnosing blood diseases
Fortran	formula translator
LHS	left hand side
Lisp	list processing
LOOPS	hybrid rule-based language
OD	OPS with database
OPS	official production system
PUFF	expert system for diagnosing pulmonary function disorders
RETE	modified hashing scheme algorithm
RHS	right hand side
SOAR	experimental system, uses general problem-solving techniques called *weak methods*
XCON	expert configuration system
YES/MVS	expert system that assists computer-center operators

The Rule-Based Approach

A rule-based system consists of three components:

- A data base, more commonly referred to as the *working memory*
- A set of rules, the *rule memory*
- The interpreter, referred to as the *inference mechanism.*

Each rule consists of a set of patterns that incoming data must match (similar to *if* clauses in the C programming language) and a set of actions that are executed if the patterns do match (similar to *then* clauses in C).

Panel 2 presents an example of a simple rule. Several things are evident in this example:

1. The rule assesses the current state, car approaching the intersection.
2. It looks for patterns in the environment—for example, the traffic light is red—and goals to be met, going straight or turning left.
3. The rule identifies actions to take to change the state, and conditions to be met before the state can change again.
4. Rule firing or execution, which represents the control of the system, depends only on the data about the state of the world and on the inference mechanism, not on other rules or programming constructs.

Notice that the process of dealing with a red light in this situation is contained in one rule.

The programmer could also change the rule if the knowledge changes. For example, if the "right on red" law is repealed, then the programmer can simply delete the third IF clause in Panel 2, and the system is modified to conform to this new requirement without affecting the rest of the system. (In this modified rule, a car would stop regardless of the direction—straight, left, or right.)

Panel 2. A Simple Rule

IF: the car is approaching an intersection
and the traffic light is red
and direction to go is straight or left
THEN: stop the car
and wait until the light turns green

This brief example stresses the interaction between the data in working memory and the rules in rule memory. What is not evident in the example is the role of the inference mechanism. The interactions among these three components and the advantages and disadvantages that they provide will be clearer after we examine each component a bit further. While describing these components, we will refer to an expansion of the preceding traffic light example that is presented in Panel 3.

Working Memory. The database or working memory describes the current state of the rule-based system, and moderates all communication between rules. If a rule needs to pass values to another rule, then it must do so through working memory.

The programmer declares the items in working memory using a format that is similar to type-declaration information in standard programming languages. For example, in Panel 3, the first element declared in working memory is of *class* `traffic light` and consists of one *attribute* `color`, which will have a value in working memory when the program is running. The *type* of value (e.g., integer, real, string) also is required in some rule-based dialects—for example, OPS83.[4]

Rule Memory. The rule memory is a collection of rules. Each rule consists of a set of conditions and a set of actions. The programmer constructs the rules so that each represents a functionally independent and meaningful chunk of the problem solution. The set of conditions is commonly referred to as the LHS (left hand side), and the set of actions is commonly referred to as the RHS (right hand side). The LHS/RHS terminology arises because an arrow often separates a rule's condition and action clauses. The condition clauses are to the left of the arrow, and the action clauses are to the right of the arrow. (In the rules in Panel 3, THEN has the same function as the "arrow.")

Each condition in the LHS must refer to a *legal* (previously declared) working-memory element. Even with this restriction, there is a fair amount of latitude in the form used on the LHS to specify conditions. The pattern on the LHS can be an exact replica of a working-memory element; such a pattern is called a *literal* (e.g., `my_car=me`). An LHS can also contain predicates that test for set, logical, or arithmetic relations. An example of this is the `member-of` predicate in the first if-clause of rule 1 in Panel 3.

Other legal forms for patterns on the LHS include the use of the negation operator and variables. Because the scope of a variable binding (where the variable has meaning) is the entire rule, the same variable may be referred to in several clauses on the LHS and on the RHS. A consistent set of bindings on the LHS is called an *instantiation* of the LHS.

The RHS of a rule is executed if the clauses on the LHS match working memory and the inference mechanism selects that rule for execution. When a rule is executed or *fired*, the actions in the RHS are taken. This may include adding, deleting, or modifying working-memory elements, or performing input or output to a specified stream. Rule 1 in Panel 3 illustrates an RHS that performs a modification to working memory.

In a rule-based system, the RHS of a rule is the only place where working memory can be modified. The LHS of the rule is used solely for pattern matching.

Inference Mechanism. In procedural languages, the sequence of the program statements and explicit control statements determines execution order. But in rule-based programming, the inference mechanism regulates the matching, selection, and execution of rules.

The inference mechanism is similar to an interpreter executing the following four-step loop:

1. In the *match* phase, the inference mechanism

Panel 3. The Stoplight Example

Working Memory Elements

```
traffic_light  color

car_motion     starting_point
               my_car
               current_direction
               desired_direction
               ending_point
```

Working Memory Example

```
traffic_light  color=red

car_motion     starting_point=home
               my_car=me
               current_direction=approach_intersection
               desired_direction=straight
               ending_point=airport
```

Rules

Rule 1:

```
IF   car_motion     my_car=me
                    current_direction=approach_intersection
                    desired_direction=(member-of straight, left)
 and traffic_light color=red

THEN change current_direction to equal stopped
```

Rule 2:

```
IF   car_motion      my_car=me
                     current_direction=stopped
                     desired_direction=straight
 and traffic_light   color=green

THEN change current_direction to equal straight
```

collects all rules that match the current state in working memory.

2. During the *select* stage, the inference mechanism selects the one rule that will execute, if there is more than one rule that matches working memory. Usually, a process called *conflict resolution*, which specifies how rule priority is determined, moderates this selection.
3. In the *act* stage, the RHS clauses of the selected rule are executed (also referred to as *firing* the rule). When a rule executes, usually data is deleted or added in working memory.
4. After the rule's actions are executed, the inference mechanism again begins the match stage (step 1). This continues until no more rules match, or an explicit halt terminates the loop.

Most rule-based languages provide their own inference mechanism, and the rules that govern conflict resolution are carefully explained in the documentation. (An exception to this is OPS83—which, besides providing a default inference mechanism, also gives the user the capability to change the inference mechanism. Other dialects—e.g., EMYCIN[5]—include the ability to assign confidence factors or weights to rules or working-memory elements. Because these weights are used during conflict resolution to decide what rule should fire, the programmer can directly influence the outcome.)

The inference mechanism's operation is where rule-based programming differs from the standard procedural methodology. A procedural program operates by continuously executing code. What usually determines the order of code execution is the programmer's ordering of the code or the value of several local state variables.

In a rule-based system, the program oscillates between a match/select phase and an execute phase. The contents of working memory and the conflict-resolution phase determine what actions (comparable to program statements) are executed next. In contemporary rule-based languages, where the programmer places rules has **no** influence on what actions are executed next. (However, in earlier systems such as Waterman's rule-based system,[6] rule ordering did affect rule execution.) But in procedural languages, the organization of the program usually is a strong determiner of what statement is executed.

This order independence has an important effect on the development process of rule-based systems. Unlike the program statements in procedural systems, rules can usually be added or deleted anywhere in the system without affecting the program's execution. This flexibility makes it easier for the programmer to modify the system—an important attribute where requirements continually change because of an increased understanding of the task requirements or a changing environment that makes aspects of the current program obsolete. In situations of rapid change or continuous refinement of requirements, a rule-based programmer has a distinct advantage over the standard procedural programmer.

A useful analogy (attributed to Ted Kowalski) emphasizes the power of this flexibility in rule-based systems. Consider a program represented as a deck of cards. Each card represents a chunk of the program, e.g., a line of code in a C program delineated by a semicolon or a rule in a rule-based language. In the C language case, if one dropped the deck of cards and reassembled them randomly, then the program would be unlikely to work as intended. (Those of us who grew up in the age of punch cards may find this an all too painfully true example!) On the other hand, a programmer who uses rules could reassemble the cards randomly, and the program would function as intended if the inference mechanism could deal with unordered rules.

Strengths and Weaknesses of Rule-Based Programming. Most of the strengths of rule-based languages are a result of two characteristics: the format of the rules, and the "separation" of control from the program.

The rule's IF-THEN representation often matches well with the expert's description of the problem-solving process. (If this condition exists, then I take this action.) Thus, the programmer can create independent units of the program (rules) that are meaningfully related to what the expert claims is the solution process. This independence and modularity of rules make it easier to change the program as our understanding increases and also explain the system's behavior in relation to these meaningful units.

The separation of control from the rules contributes to this independence because the rule content can concentrate on meaningful chunks of the solution process rather than aspects of control.

The weaknesses of rule-based languages are directly related to the characteristics that are considered to be its strengths. While useful for describing knowledge related to the solution process, the format of the rules is restrictive for expressing other types of knowledge.[7] The modularity and independence of rules and that they only interact through the working memory make it difficult to represent dependent subtasks, such as the subroutines in procedural languages. Because rules interact indirectly through working memory, Davis and King[8] have characterized rule-based programming as: of necessity, programming by side effects! Such reliance on side effects is considered harmful in some quarters, most notably by advocates of functional programming.[9]

The format of the rules, separation of control from the program, and single channel of interaction (working memory) also make it difficult to represent other commonly used programming constructs, such as iteration and recursion. In addition, these three aspects of rule-based programming contribute to the opacity of control that many investigators have attributed to rule-based languages.[8,10,11] This opacity results from the implicit control regimen of rule-based languages—to analyze the behavior of such programs, one must monitor the interaction between rule memory and working memory. But working memory changes on every cycle, so it is difficult to recapture the state of the system. Even when languages permit the programmer to freeze the state of the interpreter, the large number of items in working memory sometimes makes troubleshooting a Herculean task.

Therefore, there is an important contrast between rule-based and procedural languages. Rule-based languages explicitly represent problem-solving knowledge (*domain knowledge* in expert system terminology), yet implicitly represent control through the inference mechanism and the interaction of rule memory with working memory. On the other hand, procedural languages explicitly represent control through the program's organization, yet implicitly represent problem-solving knowledge, which is far less localized than in rules. This contrast is important in determining what tasks are appropriate and inappropriate for rule-based languages. Tasks that would benefit from the encapsulation of problem-solving knowledge are good candidates for rule-based languages.

Panel 4. Declaring Pattern Types in OPS5

```
(literalize car_motion
                        starting_point
                        my_car
                        current_direction
                        desired_direction
                        ending_point)

(literalize traffic_light
                        color)
```

Appropriate tasks for rule-based languages are ones that:

- Have a large number of distinct states (e.g., diagnostic problems, such as diagnosing an ailment)
- Can separate knowledge from control (e.g., classification problems)
- Can be viewed as a sequence of transitions to a variety of subproblems (e.g., synthesis, such as the design of a microchip[12])
- Are ill-defined; that is, have vague requirements
- Will require extensive modification and maintenance throughout their life cycle.

Inappropriate tasks for rule-based languages are ones that:

- Are algorithmic

Panel 5. Rules Written in OPS5

```
(p red_light_1
   (car_motion
       ^my_car <instance>
       ^current_direction approach_intersection
       ^desired_direction << straight left >>
   )
   (traffic_light
       ^color red
   )
   -->
   (modify 1  ^current_direction stopped)
)
(p green_light_1
   (car_motion
       ^my_car <instance>
       ^current_direction stopped
       ^desired_direction straight
   )
   (traffic_light
       ^color green
   )
   -->
   (modify 1  ^current_direction straight)
)
```

- Are well understood
- Merge representation and control (e.g., recipes).

A notable example of where rule-based programming succeeded when procedural languages were "not completely successful" was Digital Equipment Corporation's XCON system,[13] used for configuring Digital Equipment Corporation VAX™ computer systems. (Here, we use the term "procedural languages" loosely, because the two "procedural" languages were Fortran and BASIC.) The XCON project succeeded using OPS5, a rule-based language, whereas previous attempts had used Fortran and BASIC. Although Kraft[13] does not speculate directly on why OPS succeeded where Fortran and BASIC failed, the text suggests two possible reasons: the requirements were vague, and the environment was changing rapidly.

Programs have also followed the opposite route. PUFF,[14] an expert system for diagnosing pulmonary function disorders, was originally coded in EMYCIN. After the knowledge needed was well defined, it was recoded into BASIC for efficiency and portability.

Schor[15] also has contrasted rule-based and procedural programming styles during the development of YES/MVS, an expert system that assists computer center operators. Here, programmers found that initially they tended to write rules in a procedural style but, after several iterations, wrote rules more effectively. (Programmers who try object-oriented programming for the first time make a similar observation.)

An Example: The OPS5 Language

The OPS5 programming language is a popular version of a rule-based language first designed and built by Charles Forgy[16] of Carnegie-Mellon University. A main feature of this language is the RETE algorithm,[17] a modified hashing scheme—designed by Forgy—that provides performance improvement for pattern matching, a bottleneck in rule-based system performance.

The OPS5 architecture is a standard one consisting of working memory, rule memory, and the inference mechanism. We will examine the syntax of OPS5 by using it to code a portion of our traffic-light example (Panel 3).

First, we will use the `literalize` command (Panel 4) to declare the types of patterns—called classes—that can exist in working memory. This command declares a set of pattern types or templates that the LHS of a rule must match in working memory so that the rule is eligible to execute. After the working memory patterns are declared using the `literalize` command, rules are added. To build rules in OPS5, we nest the patterns (LHS) and actions (RHS) in parentheses. An arrow delineates the LHSs and RHSs.

Panel 5 presents an example of rules for the traffic light example. The rule is embedded in a set of parentheses as is each pattern on the LHS and action on the RHS. The first character in a rule is always `p`, which calls the function for creating one rule. The symbol after the `p` is the name of the rule (e.g., `red_light` or `green_light` in Panel 5.)

Patterns begin with their class name and symbols that are prefaced by "^" are attributes of the class. Each attribute is followed by a variable, designated by "< >"; a list of possible values, designated by "<< >>" (similar to our `member-of` function in Panel 3); or a literal, which is the value to be matched. It is not necessary to list all the attributes in a class. For instance, in the `car_motion` template (Panel 5), we did not include `starting_point` and `ending_point` because they have no bearing on the knowledge that these rules represent.

If the rule matches a pattern in working memory and the inference mechanism—using the conflict-resolution mechanism—selects the rule, then the actions on the RHS of the rule are taken. In both rules listed in Panel 5, the action is to modify working memory. The number after the term `modify` specifies what pattern on the LHS to modify, and the rest of the action specifies the attributes and their new values. This process continues until no more rules match the current state of working memory.

Working-memory elements can be added using the `make` command and are deleted using the `remove` command. These two actions can be included in the RHSs of rules in a way similar to the `modify` command. Usually, a rule-based program in OPS5 begins with a series of `makes` that the developer has either included with the rules or read from another file. A complete description of rule-based programming using OPS5 can be found in the textbook by Brownston et al.[10] or in Forgy's original user manual.[16]

Rule-Based Programming in the UNIX System

When we began the ACE project, we settled on a programming environment that was essentially UNIX system: the Lisp programming language and OPS4 (an earlier version of the OPS programming language). We needed the Lisp language because OPS4 was written in Lisp. As the ACE system's development proceeded, we also used the C programming language and the UNIX system's AWK language,[18] shell programming, and other development tools such as a local database language.

We used these programming languages and tools because the Lisp and OPS4 languages—while convenient for expressing much of the knowledge about the domain—were not convenient for doing other required tasks.

When we first began building the expert system, we gave more thought to encoding the knowledge for the expert system than to constructing the entire information-processing system. As more of the system came together, it was clear that we faced many of the problems that more mainstream projects experienced. We required more long-term storage, communication with other systems, user interfaces, and system maintenance and administrative capabilities including backups and crash recovery. The Lisp and OPS4 languages were not the best for doing most of those chores (which were mostly procedural), and we found ourselves relying more on tools and languages that the UNIX system provided. However, the expert system component had to interact with these tools and languages, and this interaction was sometimes awkward.

For future systems, we thought that we could improve the interactions between UNIX system and the expert system component. The common denominator had to be C, the base language of UNIX system. That the earlier versions of OPS were written in Lisp added a layer of complexity to our interactions with UNIX system tools. But we wanted to maintain the ease of representation that the OPS language provided for encoding knowledge, and we felt no need to "invent" yet another rule-based language. We, therefore, decided to base our language on OPS5.

C5: A Rule-Based Language for Building Expert Systems. Our version of OPS5 is the C5 language (developed by James Rowland of AT&T Bell Laboratories[19]) and is written in the C programming language. It is completely compatible with the OPS5 programming language. Therefore, any rule-based program that is written in OPS5 using only rules will run unmodified in C5. However, OPS5—as defined in the original technical report[14]—also interacted extensively with Lisp. Written in C language, C5 could not maintain this capability. Instead, we created a similar interface using C language that gave C5 a procedural representation to augment its rule-based representation. (Versions of OPS5 based on other languages are available from other sources. For example, Digital Equipment Corporation offers a version of OPS5 written in the BLISS programming language.)

C5's relationship with the C programming language adds a great deal of flexibility for the programmer who develops rule-based systems in the UNIX system. Using C5, a developer can build a program's rule-based component using an interpreter supplied with the C5 programming system. This interpreter provides a versatile environment for debugging a rule-based program.

The user can single-step the rule execution, and examine the state of working memory and the *conflict set* (the set of rules that match the current situation). A user can also *retract* rule firings, thereby backing up the execution and discovering why rules fired in the sequence that they did. (Remember—because working memory moderates the rule firing—to determine why particular actions were taken, one must be able to examine the state of working memory at each step.) The user may also add C language functions that are called by the RHS of the rules.

After the developer is satisfied with the system's execution, the code can then be compiled in a larger C program. The interpreter can be triggered by a C main program or a C function, run its course, and then return to the C program. In this mode, the code for the C5 interpreter is loaded as a C library.

Such tight integration with the C programming language makes it possible for knowledge engineers to develop expert systems with straightforward access to the UNIX system's capabilities, including procedural languages. Also, developers can include a bit of an expert system in their programs or, when convenient, simply represent some of their task in a rule-based formalism without drastically changing their programming environment. Therefore, the programmer can draw on the strengths of the rule-based methodology that OPS5 provides and the strengths of the procedural methodology that C and the UNIX system provide and can embed the code in an existing system.

A Supporting Framework for C5. As we have stressed throughout, the complete implementation of an expert system is more than the knowledge base. An expert system must interact with humans, other computer systems, and databases. Currently, the facilities of the UNIX system and C provide tools for the human interface and communication with remote systems when using C5. For dealing with databases, Douglas Gordin of AT&T Bell Laboratories has developed a collection of routines known as OD[20] (for OPS with database). OD permits the expert system to interact with the databases using straightforward commands from the OPS RHS functions. (It currently interacts with an internal AT&T database tool, with AT&T's Tuxedo™ database system and Relational Database Systems' Informix® database system to follow.) As an example of its power, OD permits a developer to load either an entire database or selected parts of it into working memory with one command. The developer can also update a record in the database using a RHS command. This gives C5 users a convenient mechanism for interacting with databases.

Future of Rule-Based Systems and Other AI Techniques

Initially, most rule-based languages were developed and deployed in Lisp. As the methodologies and languages became more stable and proved useful in several areas, their features began appearing in more standard languages. Continuing in this framework, new versions of rule-based languages, such as SOAR,[21] are experimenting with *weak methods* (general problem-solving techniques) to augment the application-specific knowledge that is represented in most rule-based systems.

In addition, there is a growing movement to develop hybrid languages, such as LOOPS,[22] that merge the capabilities of rule-based languages with other programming methodologies such as objects and frames.[7] As these new approaches are investigated in Lisp-based implementations, their strengths will be incorporated in future versions of tools, such as C5, that are based on more standard languages.

What makes this exploration possible is the flexibility and expressiveness of the Lisp language and the quality of its debugging environment. Until recently, competing, incompatible dialects have hampered programming efforts in Lisp. But in the last five years, a new dialect, Common Lisp,[23] has received wide acceptance as an emerging standard in both the academic and commercial artificial-intelligence communities.

Within AT&T, Elia Weixelbaum and Ilan Caron have developed a version of Common Lisp called Portable Lisp, or Plisp, that has three goals:

- Portability
- Compactness
- Adherence to the Common Lisp standard.

They achieve portability in several ways. First, they wrote the kernel of Plisp's interpreter and some Common Lisp functions entirely in C. Second, Plisp's compiler was written in Common Lisp and produces C code that is compiled by the standard C compiler. The object code is loaded into the running Lisp image, using an incremental loader developed by John Puttress.[24] Because Plisp relies on C, it runs on a variety of machines, including AT&T 3B2, Digital Equipment Corporation VAX, and Hewlett-Packard 9000 computers.

Compactness is related to portability. Current versions of the Common Lisp language are large, with running process sizes measured in megabytes. (Current versions of Franz Extended Common Lisp and Kyoto Common Lisp that run on our 3B2 computer have process sizes of over 4 megabytes.) On most current hardware (VAX or 3B computers), processes this large can slow system performance for other users, even with today's paging systems. In contrast, Plisp's initial process runs on less than a half megabyte, producing a more hospitable multiprocess environment.

When one adheres to the Common Lisp standard, programs developed on other systems can be easily transferred to Plisp. This is especially important for users of Lisp workstations such as the Texas Instruments Explorer™ workstation.

Plisp provides an environment for experimenting with sophisticated artificial-intelligence techniques. For example, CL5—a version of OPS5 written in Common

Lisp—will run in Plisp. This combination of CL5 and Plisp will be used for prototypes of potential enhancements to C5. We will then test the prototype language in applications and incorporate useful features into C5. This gives us a convenient environment to experiment with more sophisticated artificial-intelligence software techniques without incurring extensive up-front development costs.

The Plisp-CL5 environment also enables developers to develop their product on AI workstations with extensive debugging environments, and then deliver the product on more conventional hardware using Plisp and CL5.

We expect Plisp to become increasingly integrated with the UNIX system in a way similar to C5. Therefore, we hope to overcome some of the difficulties that we encountered when using the Lisp and OPS4 languages to build ACE. Plisp will be able to call C language functions from Lisp. Plisp will also exist as a C library so that programmers can use Lisp in their C programs when appropriate. Because it emphasizes compactness, Plisp will keep the resulting process size within reasonable limits.

Therefore, Plisp enables UNIX system-based developers to use Lisp for quick prototyping of applications that require artificial-intelligence techniques and, in the future, will permit them to use Lisp with C. It also provides a migration path for AI-workstation users, permitting them to develop the code on their workstations and deliver it as a product on more conventional hardware.

Summary

Rule-based languages are widely used for building expert systems. Some of the attributes that make rule-based languages useful for expert system programming may also apply to other programming tasks. Tasks that are amenable to rule-based languages have many distinct states (diagnostic problems), are ill-defined, and require extensive modification throughout their development cycle. Tasks that are algorithmic are not good candidates for rule-based languages.

Major impediments to the widespread use of rule-based languages have been that they rely on Lisp and cannot interact conveniently with conventional languages. C5 is a rule-based language that overcomes these impediments. It is written in C and can be used as an interpreter or loaded as a C library. In addition, C5 can call C functions, and C functions can invoke C5.

C5's interaction with conventional systems is further enhanced by OD, which serves as a bridge between C5 and popular database management packages. Also available to UNIX system programmers is a version of Common Lisp, Plisp, which is closely integrated with C and the UNIX system. These tools permit developers to use rule-based and other artificial-intelligence techniques in their projects without relinquishing the power, flexibility, and familiarity of the UNIX system.

References

1. J. McDermott, "R1: A rule-based configurer of computer systems," *Artificial Intelligence*, 1982, pp. 39-88.
2. G. T. Vesonder et al., "ACE: An expert system for telephone cable maintenance," *IJCAI*, Vol. 8, 1983, pp. 116-120.
3. J. R. Wright and E. M. Siegfried, "ACE: Going from prototype to product," *Expert Systems and Knowledge Engineering: Essential Elements of Advanced Information Technology*, Thomas Bernold (ed.), North Holland, New York, 1985, pp. 121-131.
4. C. L. Forgy, *The OPS83 Manual*. Productions Systems Technologies, 1985.
5. B. G. Buchanan and E. H. Shortliffe, *Rule-based Expert Systems*, Addison-Wesley, Reading, Massachusetts, 1984.
6. D. A. Waterman, "Adaptive production systems," *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, 1974, pp. 296-303.
7. R. J. Brachman, "The Basics of Knowledge Representation and Reasoning," *AT&T Technical Journal*, Vol. 67, No. 1, January/February 1988, pp. 7-24.
8. R. Davis and J. King, "The origin of rule-based systems in AI," *Rule-based expert systems: The MYCIN experiments of the Stanford heuristic programming project*, Buchanan and Shortliffe (eds.), Addison-Wesley, Reading, Massachusetts, 1984.
9. P. Henderson, *Functional programming: Application and implementation*, Prentice-Hall, Englewood Cliffs, New Jersey, 1980.
10. L. Brownston et al., *Programming Expert Systems in OPS5: An Introduction to Rule Based Programming*, Addison-Wesley, Reading, Massachusetts, 1986.
11. A. Barr and E. A. Feigenbaum, *The Handbook of Artificial Intelli-*

gence, Volume 1, William Kaufmann Inc., 1981, pp. 190-199.

12. T. J. Kowalski, "A VLSI Design Automation Assistant: A Synthesis Expert," *AT&T Technical Journal*, Vol. 67, No. 1, January/February 1988, pp. 81-92.
13. A. Kraft, "XCON: An expert configuration system at Digital Equipment Corporation," *The AI business: The commercial uses of artificial intelligence*, P. H. Winson and K. A. Prendergast (eds.), MIT Press, 1984.
14. J. S. Aikins, J. C. Kunz, and E. H. Shortliffe, "PUFF: An expert system for interpretation of pulmonary function data," *Computers and biomedical research*, 1983, pp. 199-208.
15. M. I. Schor, "Declarative knowledge programming: Better than procedural?," *IEEE Expert*, 1986, pp. 36-43.
16. C. L. Forgy, *OPS5 User's Manual*, Technical Report CMU-CS-81-135, Department of Computer Science, Carnegie-Mellon University, July 1981.
17. C. L. Forgy, "RETE: A fast algorithm for the many pattern/many object match problem," *Artificial Intelligence*, Vol. 19, No. 1, 1982, pp. 17-37.
18. A. V. Aho, B. W. Kernighan, and P. J. Weinberger, *The AWK Programming Language*, Addison-Wesley, Reading, Massachusetts, 1988
19. J. R. Rowland and G. T. Vesonder, "C5 User Manual, Release 1.0," AT&T Bell Laboratories, unpublished manuscript, July 1987.
20. D. Gordin, "OD: A tool to interface C5 with a database," AT&T Bell Laboratories, unpublished manuscript, January 1988.
21. J. Laird, J. Rosenbloom, and A. Newell, *Universal subgoaling and chunking*, Kluwer Academic Publishers, 1986.
22. D. G. Bobrow and M. Stefik, *The LOOPS manual*, XEROX Corporation, December 1983.
23. *Common LISP the language*, Digital Equipment Press, 1984.
24. J. J. Puttress and H. H. Goguen, "Incremental loading of subroutines at runtime," AT&T Bell Laboratories, unpublished manuscript, April 1986.

(Manuscript received November 2, 1987)